Proceedings, The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)

Portfolio Online Evolution in StarCraft

Che Wang, Pan Chen, Yuanda Li, Christoffer Holmgård, Julian Togelius
New York University
Brooklyn, NY 11201
{cw1681, pchen, yuanda.li, holmgard}@nyu.edu, julian@togelius.com

Abstract

Portfolio Online Evolution is a novel method for playing real-time strategy games through evolutionary search in the space of assignments of scripts to individual game units. This method builds on and recombines two recently devised methods for playing multi-action games: (1) Portfolio Greedy Search, which searches in the space of heuristics assigned to units rather than in the space of actions, and (2) Online Evolution, which uses evolution rather than tree search to effectively play games where multiple actions per turn lead to enormous branching factors. The combination of both ideas lead to the use of evolution to search the space of which script/heuristic is assigned to which unit. In this paper, we introduce the ideas of Portfolio Online Evolution and apply it to StarCraft micro, or individual battles. It is shown to outperform all other tested methods in battles of moderate to large size.

Introduction

Real-time strategy games often involve AI problems with huge branching factors. For instance, *StarCraft* (Blizzard Entertainment 1998) is a popular RTS game by Blizzard Entertainment where multiple players battle together, collect resources, build their economies, establish bases, train armies, and try to eliminate the opposing player(s) with a variety of strategies. Among these aspects of the game, combat is a challenging one. In an encounter between two armies of each 12 units (24 total), assuming each unit can attack a opponent unit or move in 4 directions, the branching factor for each move can be as high as 16^{12}. And such a move has to be decided within 40 ms in order to ensure a 24 frames per second game speed. If multiple AI players are present in a game or the graphics get more demanding, this time constraint may be even harsher. It's very challenging for a traditional search algorithm to tackle this problem.

In commercial RTS games, AI players are mostly implemented by scripts. A script is a fixed policy that tells units what to do under certain conditions. A very simple script used in the StarCraft AI competition (Churchill and Buro 2016; Ontanón et al. 2013; Churchill et al. 2016) is *AttackClosest*. The AttackClosest script simply instructs a unit to attack the enemy unit that is closest to it, or if no enemy units are in range, move towards the closest enemy unit. More complicated scripts are designed in RTS games like StarCraft to provide interesting AI opponents for human players. These scripts do not give very strong AI players, but they are very efficient and can be used without slowing down the game. Although scripted AI players still dominate commercial RTS games, research has been done to develop better methods. The current best published algorithms for StarCraft micro are Portfolio Greedy Search (Churchill and Buro 2013) and Script- and Cluster-based UCT (Justesen et al. 2014). Both of these approaches are built on the idea of selecting not which action to take for each unit, but which script to use to select an action for each unit. This significantly decreases the search space (if there are fewer scripts than action) to the cost of requiring some domain knowledge (in find good scripts to select among). The Script- and Cluster-based UCT is also built on the very popular Monte Carlo Tree Search algorithm (Kocsis and Szepesvári 2006; Coulom 2006; Browne et al. 2012). These algorithms outperform purely scripted players significantly and have shown promising efficiency in medium to large scale unit battles compared to other search algorithms. (However, it should be noted that for the complete StarCraft AI challenge, which includes several layers of strategy, the best-performing bots include significant amounts of hand-coding with simulation-based planning usually relegated to a peripheral role. (Churchill and Buro 2016; Ontanón et al. 2013; Churchill et al. 2016))

Recently, another method called *Online Evolution* (Justesen, Mahlmann, and Togelius 2016) was proposed as a fundamentally different way of dealing with game-playing problems with very large branching factor. The Online Evolution algorithm was implemented for the turn-based multi-action game *Hero Academy* (Robot Entertainment 2012), where a player can control multiple units and can do multiple moves each turn. Thus, this game also has a huge branching factor and cannot be tackled by traditional search methods. Online Evolution takes an evolutionary approach: Instead of searching through all possible moves, it evolves a sequence of unit moves. Essentially, the action selection problem is seen as an optimization problem rather than a planning problem. This outperforms all other algorithms implemented for the game. Online Evolution is inspired by

Rolling Horizon Evolution (Perez et al. 2013), where evolution is used in real-time to search for sequences of actions for a single agent. This approach has been shown to work well for the Physical Traveling Salesman Problem, which is a blend of a TSP with a racing game (Perez, Rohlfshagen, and Lucas 2012). However, evolution has not been used for action selection in a multi-agent, real-time adversarial scenario such as an RTS game before.

In this paper we combine the ideas of Portfolio Greedy Search and Online Evolution to build a new algorithm for RTS games: Portfolio Online Evolution (POE). POE keeps a collection of scripts, called a portfolio, and keeps mutating and selecting in a population where each individual is defined as a sequence of scripts that are used to generate moves for corresponding units. This paper will focus on the implementation, performance and efficiency analysis of this new algorithm. Performances of different algorithms are compared in a StarCraft combat simulator *JarCraft* (Tillman 2014) under different army size settings and a harsh time limit.

Related Work

In this section, we briefly review the StarCraft Combat Simulator used for the experiments, JarCraft, scripts used in StarCraft AI competitions, as well as the other search algorithms that are compared to Online Evolution.

JarCraft

JarCraft is an open-source combat simulator, implemented in Java, for StarCraft (Tillman 2014). It is a translation of another StarCraft combat simulator, *SparCraft*, which is implemented in C++ (Churchill 2013). In JarCraft we can efficiently simulate unit combat with different army sizes, while other parts of the StarCraft game, such as economy and construction, are not considered. In JarCraft, a few game elements are simplified to ensure experimental efficiency e.g. there is no collision among units. A more detailed explanation on the features of the simulator can be found at (Churchill, Saffidine, and Buro 2012) and (Churchill 2013).

Scripts

In previous research and StarCraft AI competitions, various scripts have been designed as AI controllers for combat units. Among them the two most popular ones are No-OverKill-Attack-Value script (NOK-AV) and the Kiter script. NOK-AV instructs a unit u to target the enemy in range with highest $\frac{dpf(u)}{hp(u)}$ value, where $dpf(u)$ represents the damage the enemy unit can inflict per game frame and $hp(u)$ the health point of the enemy. If no enemy is in range, the unit will move towards the closest enemy. NOK-AV avoids overkill and lets friendly units cooperate to inflict the highest total damage to the enemy. The Kiter script is unique in that it instructs a unit that is in attack range but currently cannot attack to move away from the enemy and then move forward again to attack. This feature allows units to constantly move back in order to evade enemy melee units or force the enemy to change their focus of fire. The Kiter script used in our experiments also targets the enemy unit with the highest $\frac{dpf(u)}{hp(u)}$ value.

State Evaluation

A state evaluation method called LTD2 is used in JarCraft to decide the quality of a certain state. LTD2 computes the value of a state s using unit sets U_1 and U_2 that are controlled by Player 1 and Player 2 respectively, as well as health points $hp(u)$ and damage capability $dpf(u)$ of each unit u. The score of current game state for Player 1 is calculated as:

$$\text{LTD2}(s) = \sum_{u \in U_1} \sqrt{hp(u)} \cdot dpf(u) - \sum_{u \in U_2} \sqrt{hp(u)} \cdot dpf(u)$$

Note that the LTD2 score can be used to evaluate terminal states, where all units of one player are eliminated as well as non-terminal states. It can be used alone or combined with a playout. In a playout the state is forwarded under a certain policy. Churchill et al. found that among all scripts NOK-AV has the best performance for playouts. Such a playout will generate moves for both players using NOK-AV for all units until a terminal state or until a round limit is reached. The final state is then evaluated using LTD2. This is the method used by UCT related algorithms. Portfolio Greedy Search and Portfolio Online Evolution utilize more complicated playout policies that we will elaborate in later sections.

Portfolio Greedy Search

Portfolio Greedy Search (PGS) is a novel hill climbing greedy search algorithm proposed by Churchil and Buro (Churchill and Buro 2013). PGS uses a set of scripts P which is called the *Portfolio* to generate moves for units. There are 3 major procedures:

GetSeedPlayer at the beginning, a set of scripts, all set to a default script which is NOK-AV in (Churchill and Buro 2013), is returned to the player. So with a seed set of scripts all units of this player are assigned NOK-AV script. Both the player's scripts and a set of enemy scripts are initialized using GetSeedPlayer.

Improve PGS iterates through all units that the player controls, and for a unit u assign each script in the portfolio to it and do a playout. For the duration of this playout all the player's units will follow their currently assigned scripts and the enemy's units follow the enemy's current scripts. The resulting state will be evaluated and a script with the highest value will be assigned to unit u. The algorithm then goes to the next unit. PGS finds a better set of scripts with this hill-climbing method.

Response after the player's choice of scripts are improved, the enemy's scripts then get improved, using the player's scripts as its enemy. The idea is to let the player and the enemy scripts get improved in turn multiple times before giving the final scripts to use. By using a greedy hill-climbing method, PGS has a reduced search space and has shown better performance than best algorithms before it, such as the UCTCD.

Script- and Cluster-based UCT

Upper Confidence Bound (UCT) is a very popular member of the Monte Carlo Tree Search (MCTS) family (Kocsis and Szepesvári 2006; Coulom 2006; Browne et al. 2012). UCT Considering Durations (UCTCD) is an extension of UCT, designed specifically for a RTS scenario where units move simultaneously instead of in turns (Churchill and Buro 2013). For a StarCraft scenario, when unit moves are generated using UCTCD, it constructs a tree from the current state, reducing its search complexity by using methods such as random sampling and giving priority to search on more promising child nodes. At leaf nodes a playout is run and the final state is evaluated to get a score value and then back-propagates this to the root. The algorithm ends by giving the best unit moves it can find. Such a search algorithm does not search through the complete search space but has shown good performance in StarCraft combat simulations. Script- and Cluster-based UCT (Justesen et al. 2014) are two newer algorithms that can be seen as enhanced versions of UCTCD. Script-based UCT is different from UCTCD in that UCTCD searches for a vector of particular moves for units while Script-based UCT searches for a vector of scripts. This approach reduces the branching factor considerably when only a small amount of scripts are used, that is, the number of scripts used are lower than the number of particular moves a unit can take. Cluster-based UCT also uses a script-based approach so it also searches for a vector of scripts. In addition to that, it also groups units that are at similar positions into clusters. Then all the units in a cluster will use the same script to move.

Online Evolution

Evolutionary algorithms have been used widely for evolving AI controllers (Lucas and Kendall 2006; Miikkulainen et al. 2006; Risi and Togelius 2014). In most cases parameters of a controller are evolved over a number of generations, where the fitness function is defined by the controller's success in playing a game. This is known as an offline evolutionary approach because the controller first goes through evolution and then is used for the actual task. Online Evolution is novel in the sense that it does not evolve a controller, but evolves a sequence of actions for units to use for each turn in the actual game. In the implementation of Online Evolution for Hero Academy, at each turn, an initial population is created by repeatedly selecting a random action. The population has size 100, in each generation the individuals in the population are evaluated with a heuristic, then the best 50% individuals are selected. The selected ones go through a uniform crossover each with another random best individual and produce a new individual. When applying the idea of Online Evolution to StarCraft combat simulations, these parameters are changed in order to fit the new context.

Method

In (Justesen, Mahlmann, and Togelius 2016), the Online Evolution algorithm implemented for the turn-based multi-action game Hero Academy (Robot Entertainment 2012) is designed to evolve a sequence of particular unit moves and tries to find the best sequence of moves for a turn. One of the problems in this implementation is the existence of illegal moves. In Hero Academy a player can issue 5 commands each turn and these commands can be issued to the same unit or to multiple units. So for instance, it can happen that the evolutionary algorithm evolves to a move that instructs a unit to move backwards, then instructs the unit to attack an enemy that the unit used to be able to attack but cannot after it moved. Such problems can be tackled in Hero Academy specifically by disallowing the selection of illegal movement from a parent action. However, during our implementation, we found that in an RTS game such as StarCraft, too many illegal moves can be included, making it hard to fix the sequence, reducing the direct applicability of online evolution. The solution is to use a script-based method. That is where Portfolio Greedy Search comes in.

Combining the advantages of Online Evolution and Portfolio Greedy Search, we arrive at Portfolio Online Evolution, which is essentially an evolutionary algorithm that tries to evolve an as good set/sequence of scripts as possible to use for units in several future steps from the current game state. We use the word "genome" to denote such a sequence of scripts. Similar to Portfolio Greedy Search, Portfolio Online Evolution first initializes all scripts to be used to NOKAV. Afterwards the mutation and selection process on the population is similar to that in Online Evolution.

There are several key differences between Portfolio Greedy Search (PGS) and Portfolio Online Evolution (POE): 1. Where PGS uses hill-climbing, POE uses evolution; 2. PGS's evaluation method is to do a playout from the root node and during the playout each unit will always use the script it is assigned to. POE's genome has the scripts for all units for several upcoming steps, see Figure 1; 3. PGS's playout will try to play until the end of game. But POE will only do a shallow playout. POE's playout often will not reach the end of game and is controlled by a parameter.

Move/Unit	1	2	3	4
1	NOKAV	Kiter	NOKAV	Kiter
2	Kiter	NOKAV	NOKAV	Kiter
3	Kiter	NOKAV	Kiter	Kiter

Table 1: An $m \times n$ matrix is used to represent an individual, or genome in portfolio online evolution. n is the number of units for a player and m is a constant set to be the future steps that the evolution is going to explore. This graph shows an indiviual when $n = 4$ and $m = 3$. During evaluation, the "Kiter" at row 3, column 1 means from the current state on, unit 1 will be using Kiter script for its 3rd move.

Algorithm Structure

A simplified description of POE is given below: A POE player has a $portfolio$, which can contain any number of scripts used for evolving. When the evolution time limit is reached, the scripts in the current best genome in the population are used to generate unit moves. To be consistent with

previous research, the NOK-AV and Kiter scripts that we mentioned in the related work section are used for all script-based algorithms in our main experiments. But to showcase how POE scales with more scripts, we also compare the performance of POE with 2 scripts (POE-2) and POE with 6 scripts (POE-6). POE-2 uses NOK-AV and Kiter, while POE-6 uses NOK-AV and the following 5 scripts: Back, which lets a unit move back from the closest enemy when reloading; BackFar, which lets a unit move towards the farthest ally unit when reloading; BackClose, which lets a unit move towards the closest ally unit when reloading; Forward, which lets a unit move to closest enemy when reloading; ForwardFar, which lets a unit move to the farthest enemy when reloading. All of these scripts command a unit to move towards enemy when they are not in attack range and attack whenever they can. All their attacks have the feature of NOK-AV, so they will choose the enemy with the highest dpf/hp value to attack and will not overkill. These scripts essentially enables a unit to move in one of 5 directions or stand still during battle, but these scripts are also quite simple and have no fancy tactics. Note that the scripts used in POE-6 forbids a unit to move back before it engages an enemy, making unit behavior more stable. $Init$ function will initialize all the genomes in the population with all NOK-AV scripts. This is to ensure that all NOK-AV, which is a baseline strategy is always available for POE. $populationSize$ is a parameter that decides the number of genomes in a population. These genomes are evaluated with a playout, the resulting value is stored with the genome. $Mutate$ function will add mutation into the population. For each gene in each genome, a $mutationrate$ is set to indicate how likely a gene will mutate into another gene. Crossover is not implemented for POE-2 because the scripts are first initialized to be all NOK-AV and for the first few generations, crossover does not make much sense for several all NOK-AV genomes and experiment results confirmed our assumption. But for POE-6, we found that implementing crossover can increase the performance of POE. So there is a group of experiments dedicated to show how crossover can change performance of POE when using more scripts. $Select$ function basically sorts the genomes in the population according to their evaluation value then select the best ones. A parameter b is set to decide how many best genomes are selected. $Eval$ function takes in a State s, a Genome g, creates a copy of State s and does a playout from this state using the scripts in g. After the playout, the final state is evaluated using the LTD2 method. $Playout$ function takes in a State s, a Genome g, and a $playoutLimit$ to evaluate a state using playout. Assume we have 4 units and a genome that is the same as in Table 1. During a playout the enemy will all use NOK-AV script for all times, and ally units will first follow the script in the genome. So the 4 units will use NOK-AV, Kiter, NOK-AV, Kiter respectively for their 1st next move; Kiter, NOK-AV, NOK-AV, Kiter for their 2nd move, and so on, until all the scripts in the genome are used, which in this case is after the 3rd move. Assume the playout limit is 25, then for the next 25-3 = 22 moves, all ally units will use NOK-AV. After 25 moves, the final state is returned for evaluation. Interested readers should consult (Churchill 2013) and (Churchill and Buro 2013) when implementing this in the SparCraft or JarCraft simulator since they simulate real-time play with an asynchronous unit action model which can be a little tricky.

```
1:  function PortfolioOnlineEvolution(State s)
2:      Genome[] pop
3:      Init(pop, s)
4:      while currentTimeUsed < timeLimit do
5:          mutate(pop)
6:          select(pop)
7:      return best genome in pop
8:
9:  function Init(Genome[] pop, State s)
10:     for x = 1 to populationSize do
11:         N ← number of units
12:         M ← number of future steps
13:         Genome g ← new Genome(N, M)
14:         pop.add(g)
15:         fill(g, NOKAV)
16:
17: function Mutate(Genome[] pop)
18:     for Genome g in pop do
19:         for i in M do
20:             for j in N do
21:                 if random(0,1) < mutateRate then
22:                     g[M][N] ← random(portfolio.size())
23:
24: function Select(Genome[] pop, State s)
25:     for Genome g in pop do
26:         Eval(g, s)
27:     pop.sort()
28:     pop ← Best b ones in pop, b is a constant
29:
30: function Eval(Genome g, State s)
31:     State s_new ← s
32:     s_new ← Playout(s_new, g, playoutLimit)
33:     return s_new.LTD2()
34:
35: function Playout(Genome g, State s, Playout Limit limit)
36:     for timestep=1 to limit do
37:         if more scripts available in g then
38:             use scripts in g for ally units, use NOK-AV for all enemy units, forward s
39:         else
40:             use NOK-AV for all units, forward s
41:     return s
```

Results

All the experiments in this paper are 2-player battles conducted in JarCraft, with a map size 25*20 tiles and a tile size of 32 pixels. A unit can move on any tile of this map. Each player in a battle can control a total of n units, half of them are Dragoons (ranged unit), the other half are Zealots (melee unit). The units' original positions are generated on vertical lines, with 16 units on a line, symmetrically. In between the two player's units, a distance of 225 pixels is set to create

a realistic battle where units can change formations before engaging with the other side. After that, a random shuffling process give a slight random change (0 to 20 pixel) to the x and y position of each unit. We set this random change to a small value to make experiment results more stable. Algorithms are tested with army size n = 4, 8, 16, 32, 64, 96.

All experiments are performed on an Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz running Windows 10 with 12 GB of DDR3 1600MHz RAM. All algorithms are implemented to run in a single thread. We use a harsh time limit of 20 ms as it better reflects the computation environment in an actual RTS game, where multiple AI players might be run on a single machine and graphics takes considerate computation power. In earlier tests we also tested on a 40 ms time limit and results are similar. Except for the time limit, the configurations for the script- and cluster-based UCTCD are the same as in (Justesen et al. 2014). The configuration for portfolio greedy search looks very basic but it is the same as in (Churchill and Buro 2013), Churchill explains that higher settings do not yet run within the time limit. We only used the cluster-based UCTCD that cluster all units since (Justesen et al. 2014) found that this generally yielded better performance. The configurations are shown in Table 2 and the win rates are shown in Figure 1.

Configuration for all algorithms:
- Time limit: 20ms
- Scripts used: NOK-AV, Kiter
Configuration for all UCT algorithms:
- Max. children: 20
- Evaluation: NOK-AV vs. NOK-AV playout
- Final move selection: Most valuable
- Exploration constant: 1.6
Script-based UCTCD:
- Child generation: One-at-leaf
Cluster-based UCTCD:
- Child generation: One-at-leaf
- Cluster max-distance-to-mean: 30 pixels
- Opponent clustering: No
- Units to cluster: All
Portfolio Greedy Search:
- Improvement iterations: 1
- Response iterations: 0
- Initial enemy script: NOK-AV
- Evaluation: NOK-AV Playout
Portfolio Online Evolution:
- Evaluation: NOK-AV Playout
- Future steps: 3
- Mutation rate: 0.2
- Number of genomes selected in each mutation: 4
- Offspring of each selected superior member: 3
- Playout round limit: 25

Table 2: An overview of the algorithm configurations.

POE compared with NOK-AV The Portfolio Online Evolution player is able to win against NOKAV with a 1.0 rate in all battles with army size $>= 8$, and loses only a very small fraction with army size of 4.

POE compared with PGS Portfolio online evolution is able to win over portfolio greedy search with all army sizes. The win rate for POE is slightly higher with 16 units or fewer. Average win rate is around 75%.

POE compared with Script- and Cluster-based UCT Portfolio Online Evolution is able to outperform both Script-based UCT and Cluster-based UCT significantly in large-scale combats where there are more than 32 units. In fact, POE gets a higher win rate when more units are in the battle. When it comes to fewer units, POE still wins but with a lower win rate. But when controlling very few units, POE cannot outperform script-based UCT. However, win rate is very near 0.5 which means their performance are very similar at a very small-scale battle. The non-deterministic nature of POE decides that it sometimes will give a sub-optimal move, and that might be the main reason why it cannot win over Script-based UCT in a very small sized battle.

PGS compared with Script- and Cluster-based UCT Since in (Justesen et al. 2014), no comparison was done on the performance between PGS and Script- and Cluster-based UCT, we implemented PGS according to the specifications of Churchill in (Churchill and Buro 2013) and did this comparison. The results show that both UCT-based methods outperform Portfolio Greedy Search in all combat sizes. It's interesting to see that PGS seems to perform best against the two UCT methods at an army size of 16 units for each player.

POE with 6 scripts compared with POE with 2 scripts Experiment results show that when POE uses more scripts, its performance in RTS battle can be not just maintained, but even further improved. Here we compared the performance of POE with 6 different scripts against POE with only 2 scripts. Results show that POE-6 outperforms POE-2 in all combat configurations.

POE with crossover implemented vs POE with uniform mutation only Experiments show that when POE has only 2 scripts, the performance of POE-2 with crossover implemented is about the same as POE-2 using only uniform mutation, no significant improvement is observed. However, when POE has 6 scripts, a considerable higher performance is observed for combats where army size is 16 or higher. This might indicate that crossover can make better use of the diversity of scripts in the portfolio.

Running time test on POE Experiments results show that the average time needed for POE is about proportional to the number of units on the battlefield, as shown in figure 2. This is mainly due to the fact that the scripts in the portfolio iterate through all enemy units to find the nearest enemy or apply the no-overkill policy. For scripts that do not iterate through units, then the running time for a generation will not change significantly as the number of units changes.

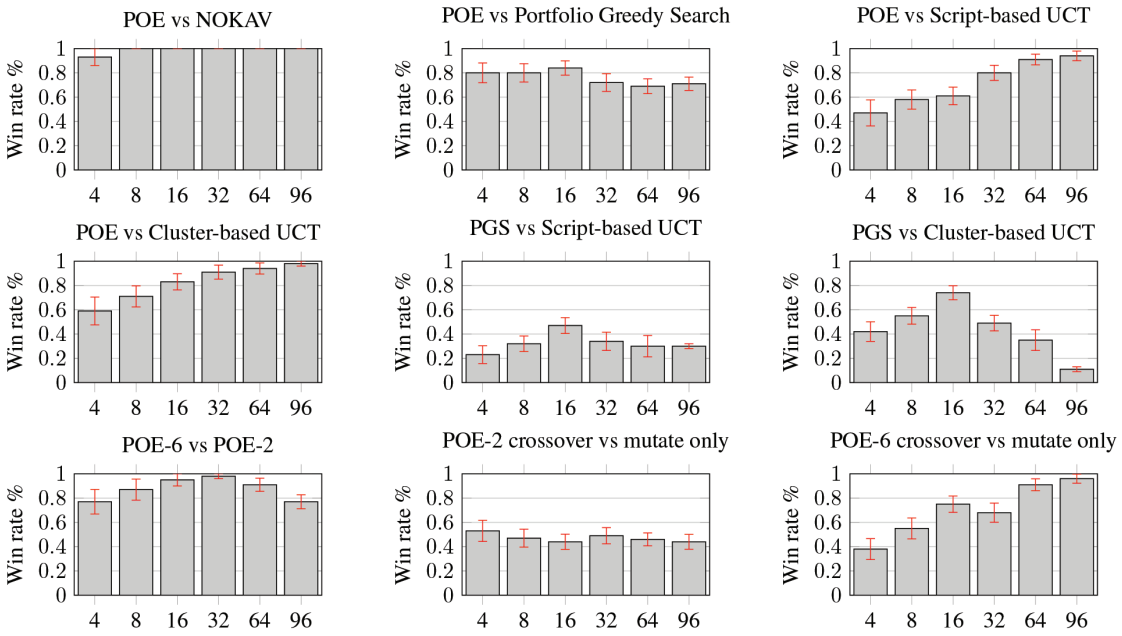

Figure 1: Win rate of the different Portfolio Online Evolution variations against other types of controllers. X-axes show the number of units on each side. 100 games were played for each army size. Error bars show 95% confidence intervals.

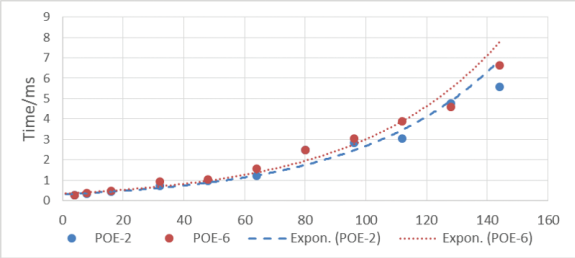

Figure 2: Average time for evolving one generation using POE with respect to army size

Discussion

Although Portfolio Online Evolution has shown good performance against other algorithms, it currently has some limitations. Due to its randomized nature, some of the moves output by POE does not make great sense. This is especially obvious when using scripts of high freedom with a large army size. From time to time, some units will showcase a random-looking behavior that is far from optimal. In some tests where a large army size is selected and POE player is given 6 scripts that allow all directions of unit moves at all times, its performance can be reduced greatly. The two popular scripts NOKAV and Kiter do not allow units to move back before they are very close to the enemy thus avoid most of this problem. But optimization is clearly required if we want to use more diverse and unpredictable strategies with POE. More scripts should be developed, possibly the scripts themselves could be evolved.

The parameters currently used for POE are arbitrarily chosen and can be optimized, possible via off-line evolution. Other enhancements such as clustering can also be applied to POE. Although it might not reduce the running time of POE since its complexity does not depend on the number of units, it can reduce the number of random moves and thus increase its stability. Parallelism is another simple optimization with potential for large speedups.Since our experiments were conducted in a combat simulator where unit collision is not modeled, it would also be of great value to see how POE works in actual StarCraft games.

Conclusions

In this paper we presented a new algorithm called Portfolio Online Evolution, which combines the ideas of Portfolio Greedy Search and Online Evolution. We tested it against 3 state-of-the-art algorithms for playing different StarCraft micro scenarios. We found that Portfolio Online Evolution outperforms all three of them in combat experiments carried out within the JarCraft simulator. The results further indicate that Portfolio Online Evolution performs relatively better the more units are involved in the battle. This suggests that this approach should scale very well to more complex scenarios.

References

Blizzard Entertainment. 1998. *StarCraft*. Blizzard Entertainment.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on* 4(1):1–43.

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

Churchill, D., and Buro, M. 2016. AIIDE Starcraft AI Competition.

Churchill, D.; Preuss, M.; Richoux, F.; Synnaeve, G.; Uriarte, A.; Ontanón, S.; and Certickỳ, M. 2016. Starcraft bots and competitions.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for rts game combat scenarios. In *AIIDE*.

Churchill, D. 2013. Sparcraft: open source starcraft combat simulation.

Coulom, R. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, 72–83. Springer.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script-and cluster-based uct for starcraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8. IEEE.

Justesen, N.; Mahlmann, T.; and Togelius, J. 2016. Online evolution for multi-action adversarial games. In *Applications of Evolutionary Computation*. Springer. 590–603.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

Lucas, S. M., and Kendall, G. 2006. Evolutionary computation and games. *IEEE Computational Intelligence Magazine* 1(1):10–18.

Miikkulainen, R.; Bryant, B. D.; Cornelius, R.; Karpov, I. V.; Stanley, K. O.; and Yong, C. H. 2006. Computational intelligence in games. *Computational Intelligence: Principles and Practice* 155–191.

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. *Computational Intelligence and AI in Games, IEEE Transactions on* 5(4):293–311.

Perez, D.; Samothrakis, S.; Lucas, S.; and Rohlfshagen, P. 2013. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, 351–358. ACM.

Perez, D.; Rohlfshagen, P.; and Lucas, S. M. 2012. The physical travelling salesman problem: Wcci 2012 competition. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, 1–8. IEEE.

Risi, S., and Togelius, J. 2014. Neuroevolution in games: State of the art and open challenges.

Robot Entertainment. 2012. *Hero Academy*. Robot Entertainment.

Tillman, B. 2014. Jarcraft.