Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-20)

Are Strong Policies Also Good Playout Policies? Playout Policy Optimization for RTS Games

Zuozhi Yang,[1] **Santiago Ontañón**[1,2]
[1]Drexel University, Philadelphia, USA
[2]Google Research, Mountain View, USA
zy337@drexel.edu, santiontanon@google.com

Abstract

Monte Carlo Tree Search has been successfully applied to complex domains such as computer Go. However, despite its success in building game-playing agents, there is little understanding of general principles to design or learn its playout policy. Many systems, such as AlphaGo, use a policy optimized to mimic human expert as the playout policy. But are strong policies good playout policies? In this paper, we take a case study in real-time strategy games. We use bandit algorithms to optimize stochastic policies as both gameplay policies and playout policies for MCTS in the context of RTS games. Our results show that strong policies do not make the best playout policies, and that policies that maximize MCTS performance as playout policies are actually weak in terms of gameplay strength.

Introduction

Monte Carlo Tree Search (MCTS) tends to outperform systematic search in domains with large branching factors. The most prominent success of MCTS is in the domain of Computer Go, where an agent, AlphaGo, built using a combination of MCTS and neural networks achieved super-human performance (Silver et al. 2016). In AlphaGo, a policy optimized to mimic human expert is used as the playout policy of MCTS. However, previous work has shown that having good gameplay strength but non-optimal might not result in a good playout policy (Silver and Tesauro 2009; Huang, Coulom, and Lin 2010; Graf and Platzner 2016).

Motivated by the question of what makes a good playout policy, in this paper, we empirically study the effect of optimizing playout policies with different objectives for MCTS in the domain of real-time strategy (RTS) games. In almost all variations of MCTS, playout policies, also called simulation policies, are used to select actions for both players during the forward simulation phase of the search process. Since the quality of the playout policy has a great impact on the overall performance of MCTS, previous work has covered various methods to generate these policies such as handcrafted patterns (Munos and Teytaud 2006), supervised learning (Coulom 2007), reinforcement learning (Gelly and Silver 2007), simulation balancing (Silver and Tesauro 2009; Huang, Coulom, and Lin 2010; Graf and Platzner 2016), and online adaptation (Silver, Sutton, and Müller 2012; Baier and Drake 2010). However, there is little generalizable understanding about how to design or learn good playout policies in systematic ways. Optimizing directly on the gameplay strength of the playout policy often yields decreased performance (Gelly and Silver 2007) (an effect we also observed in preliminary experiments, and which partially motivated this work). In recent Go research, playout policies is some times abandoned and replaced by refined evaluation functions (Silver et al. 2017; 2018).

Specifically, in this paper we evaluate the difference in behavior of game-playing policies when optimized for gameplay strength and when optimized for playout policy performance. Since our goal is just to understand what makes a good playout policy, we employ very simple policies, and use bandit algorithms for the optimization process. μRTS [1] is used as the testbed, as it offers a minimalistic yet complete RTS game environment and a collection of MCTS implementations. We optimize for two objectives: 1) winrate of the policy directly, and 2) win rate of an MCTS agent when using the policy as the playout policy.

The rest of the paper is structured as follows. First, we provide background on RTS games, MCTS, and policy optimization. Then we describe our approach for optimizing both gameplay and playout policies where the resulting policies are less exploitable. We show examples of both types of policies, then compare them with each other and with baseline policies. Finally, we draw conclusions and discuss lines of future work.

Background

Real-time strategy (RTS) is a sub-genre of strategy games where players aim to defeat their opponents (destroying their army and base) by strategically building an economy (gathering resources and building a base), military power (training units and researching technologies), and controlling those units. The main differences between RTS games and traditional board games are: they are simultaneous move games (more than one player can issue actions at the same time), they have durative actions (actions are not instantaneous), they are real-time (each player has a very small

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]https://github.com/santiontanon/microrts

amount of time to decide the next move), they are partially observable (players can only see the part of the map that has been explored, although in this paper we assume full observability) and they might be non-deterministic.

RTS games have been receiving an increased amount of attention (Ontañón et al. 2013) as they are more challenging than games like Go or Chess in at least three different ways: (1) the combinatorial growth of the branching factor (Ontañón 2017), (2) limited computation budget between actions due to the real-time nature, and (3) lack of forward model in most of research environments like Starcraft. Many research environments and tools, such as TorchCraft (Synnaeve et al. 2016), SCIILE (Vinyals et al. 2017), μRTS (Ontañón 2013), ELF (Tian et al. 2017), and Deep RTS (Andersen, Goodwin, and Granmo 2018) have been developed to promote research in the area. Specifically, in this paper, we chose μRTS as our experimental domain, as it offers a forward model for application of Monte Carlo Tree Search as well as existing implementations of MCTS and stochastic policies for optimization.

μRTS

μRTS is a simple RTS game designed for testing AI techniques. μRTS provides the essential features that make RTS games challenging from an AI point of view: simultaneous and durative actions, combinatorial branching factors and real-time decision making. The game can be configured to be partially observable and non-deterministic, but those settings are turned off for all the experiments presented in this paper. We chose μRTS, since in addition to featuring the above properties, it does so in a very minimalistic way, by defining only four unit types and two building types, all of them occupying one tile, and using only a single resource type. Additionally, as required by our experiments, μRTS allows maps of arbitrary sizes and initial configurations.

There is one type of environment unit (minerals) and six types of units controlled by players, which are:

- Base: can train Workers and accumulate resources
- Barracks: can train attack units
- Worker: collects resources and construct buildings
- Light: low power but fast melee unit
- Heavy: high power but slow melee unit
- Ranged: long range attack unit

Additionally, the environment can have walls to block the movement of units. A example screenshot of game is shown in Figure 1. The squared units in green are Minerals with numbers on them indicating the remaining resources. The units with blue outline belong to player 1 (which we will call *max*) and those with red outline belong to player 2 (which we will call *min*). The light grey squared units are Bases with numbers indicating the amount of resources owned by the player, while the darker grey squared units are the Barracks. Movable units have round shapes with grey units being Workers, orange units being Lights, yellow being Heavy units and blue units being Ranged.

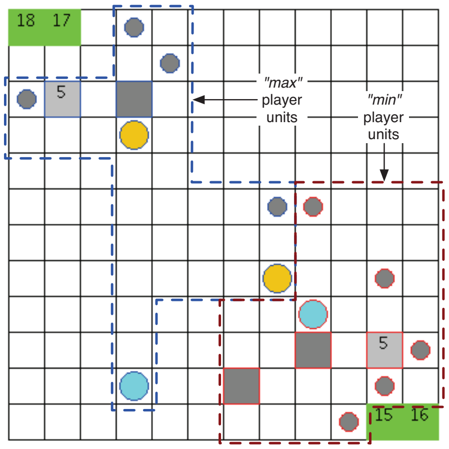

Figure 1: A Screenshot of μRTS.

Monte Carlo Tree Search in RTS Games

Monte Carlo Tree Search (Browne et al. 2012; Coulom 2006) is a method for sequential decision making in domains that can be represented by search trees. It has been a successful approach to tackle complex games like Go as it takes random samples in the search space to estimate state value. Generally, MCTS algorithms can be broken down into the following four stages (Browne et al. 2012):

1. Selection: Starting at root node, recursively select child nodes according to a pre-defined *tree policy* until a leaf node L is reached.
2. Expansion: If the selected node L is a not a terminal node then add a child node C of L to the tree (also using the tree policy).
3. Simulation: Run a simulation from C according to a *playout policy* until a terminal state is reached.
4. Backpropagation: Update the statistics of the current move sequence with the simulation result.

Most of the classic tree policies of MCTS, e.g. UCT (Kocsis and Szepesvári 2006), do not scale up well to RTS games due to the combinatorial growth of branching factor with respect to the number of units. Sampling techniques for combinatorial branching factors such as Naïve Sampling (Ontañón 2017) or LSI (Shleyfman, Komenda, and Domshlak 2014) were proposed to improve the exploration of MCTS exploiting combinatorial multi-armed bandits (CMABs). There have been many other enhancement techniques of the tree policy. But since our focus in on the playout (a.k.a. simulation) policy, we employ MCTS with Naïve Sampling in this paper for simplicity (NaïveMCTS).

Playout Policies in MCTS

If we had the optimal policy available, playout according to this policy would produce accurate evaluations of states. However, having such optimal policy is not possible in many situations. If a policy is not one of the optimal ones, no matter how good the policy is, some error is introduced into the evaluation and accumulated in the playout sequences. If the error is unbalanced (canceled in the long run), even a strong policy can result in a very inaccurate state evaluation. Previous work on simulation balancing (Silver and Tesauro 2009; Huang, Coulom, and Lin 2010; Graf and Platzner 2016) approach this problem by not optimizing policy strength but optimizing policy balance. In that way, the errors are canceled out in the long run.

Although the general principles to generate good playout policies are not yet fully understood, in practice, when learning a playout policy, the policy is trained to mimic a simulation balanced agent. This can be either an expert that can evaluate states accurately or a strong agent that can analyse the positions deeply. In the work of Silver and Tesauro (2009), the expert agent is used, and in other work (Huang, Coulom, and Lin 2010; Graf and Platzner 2016) apprenticeship learning of deep MCTS is shown to be effective. However, it isn't clear that simulation balancing is the only factor to take into account when designing playout policies. Thus, in this paper, we take a different approach, and optimize playout policies to maximize MCTS performance directly.

Policy Optimization in μRTS

In order to study the differences between policies optimized for gameplay and those optimized as playout policies, we define a very simple parametrized policy, and use an optimization process to optimize these parameters.

Policy Parameterization

We employ a simple stochastic parameterization of the policy, where we define a weight vector $\mathbf{w} = (w_1, ..., w_6)$, where each of the six weights $w_i \in [0, 1]$ corresponds to each of the six types of actions in the game:

- NONE: no action.
- MOVE: move to an adjacent position.
- HARVEST: harvest a resource in an adjacent position.
- RETURN: return a resource to a nearby base.
- PRODUCE: produce a new unit (only bases and barracks can produce units, and only workers can produce new buildings).
- ATTACK: attack an enemy unit that is within range.

A policy is totally represented by the vector $\mathbf{w}$. During gameplay, the action for each unit is selected proportionally to this weight vector. To choose the action for a given unit, the following procedure is used: given all the available actions for a unit, a probability distribution is formed by assigning each of these actions the corresponding weight in $\mathbf{w}$, and then normalizing to turn the resulting vector into a probability distribution. If the weights of all the available actions are 0, then an action is chosen uniformly at random. Notice that this defines a very simple space of policies, but as we will see below, it is surprisingly expressive, and includes policies that are stronger than it might initially seem.

The goal of keeping the policy space simple is to be able to find near-optimal policies (within the policy space), in a computationally inexpensive way. The same ideas presented here would apply to more expressive policies, parametrized by larger parameter vectors, such as those represented by a neural network (with more complex parameterization and differentiable object function), for example.

Policy Optimization

Given the parameterization, we can optimize the policy for many purposes using different optimization algorithms. In this paper, we use Fictitious play (Brown 1951). Fictitious play is one of the earliest learning algorithms in games that is able to compute a Nash Equilibrium. This is interesting since, if a policy is optimized to maximize win rates against a single other agent, cycles might be created, where we have three policies A, B, and C, and A beats B, B beats C, and C beats A. To avoid these cycles and compute the least exploitable agent, we need to approximate the Nash Equilibrium. In each iteration of fictitious play, the best-response against our current belief of the optimal strategy needs to be computed. Many algorithms can be used to compute the best response in each iteration of fictitious play.

In this paper, we used multi-armed bandit algorithms for optimization, which, given the large amount of uncertainty in the domain, and the small size of the parameter space, converged faster in our experiments than other optimization techniques such as reinforcement learning or genetic algorithms. We discretized the search space for optimization, allowing each weight to take values in $\{0, 1, 2, 3, 4, 5\}$. Moreover, notice that if we multiply a weight vector by a scalar strictly larger than zero, the resulting policy is identical in behavior. Internally, when interpreting the weight vectors as policy, the vector will be normalized to a probability distribution (that sums up to one).

To optimize a policy, we use a multi-armed bandit as follows: given an opponent policy π_o, and a map, we consider a multi-armed bandit with $6^6 = 46656$ arms (one per possible value combination, as we have 6 parameters in $\mathbf{w}$ with 6 possible values each). We pull one arm from the bandit, that represents a policy, and play one game of μRTS. After the game, we obtain rewards of 0, 0.5 or 1 for a loss, a tie or a win respectively. We keep iterating this process, pulling one arm from the bandit for each game, until the bandit eventually converges to the optimal policy.

Since we have a policy with 6 parameters, the problem has a combinatorial structure in the search space, i.e., each of the 6^6 arms is a *macro arm* that can be seen as pulling one arm from six different *local MABs* (corresponding to each of the 6 parameters to optimize), and each local MABs has six possible values to choose from. Thus, we can use the model of combinatorial multi-armed bandits (CMAB) (Ontañón 2017), or combinatorial bandits for short. Specifically, we use Naïve Sampling, as it is already implemented in μRTS.

Naïve Sampling is a sampling strategy for CMAB problems, where it is assumed we have a collection of n discrete variables, and we need to find a value assignment (a macro-arm) to each of them that maximizes some unknown and stochastic reward function. As usual, Naïve Sampling decomposes this problem into *exploration* and *exploitation*. During exploration, Naïve Sampling assumes that the reward of the macro-arm can be decomposed as the sum of the expected rewards for each of the individual variables. With

Algorithm 1: Naïve Sampling Fictitious Play(m)

```
Initialize the set of the Nash Equilibrium strategy $N$ as
  a set with one policy at random from the space of
  policies.
for $k = 1, 2, 3, \ldots, T$ do
    CMAB = new NaïveSampling() bandit
    for $k = 1, 2, 3, \ldots, K$ do
        Randomly pick one opponent policy $\pi_o \in N$
        Choose arm $\pi_k$ = CMAB.sample()
        r = play a game $\pi_k$ vs $\pi_o$ in map $m$
        CMAB.observeReward($\pi_k, r$)
    $a^*$ = CMAB.bestMacroArm()
    $N \leftarrow N \cup \{a^*\}$
```

this assumption, we can decompose the CMAB problem into n child MAB problem, denoted as $MAB_1, \ldots, MAB_n$. During *exploitation*, this assumption is not needed, and a global MAB, denoted as MAB_g is used to find the best macro-arm amongst all the ones that were proposed in past iterations. This global MAB ensures convergence to the optimal macro-arm regardless of whether the target reward can be decomposed into per-variable rewards or not.

Fictitious Play with Bandit Optimization. In order to find the optimal policy within the space of policies defined by our 6-parameter vector, we use Naïve Sampling within a fictitious play framework. Specifically, we use Algorithm 1. Given a target map m (notice that, in principle, we can use a set of maps, but for simplicity, we just optimized policies for specific maps in this paper), we use fictitious play as follows. We initialize a set of policies N with a single policy chosen at random, and then execute T iterations of fictitious play. At each iteration, we optimize a policy a^* to be the best response against the policies in N. To do this, we use K iterations of NaïveSampling, after which we add this new best response policy to N, and iterate again. As we discussed above, it has been shown that N converges to the Nash Equilibrium. Finally, since in this paper we just want to select a single policy at the end, in our experiments, we then ran a final NaïveSampling optimization process with a very large number of iterations, trying to find the best response policy against the Nash Equilibrium N. We do this, in order to obtain a policy represented just as a vector of 6 numbers, and make results interpretable, so we can compare the result of optimizing for gameplay strength, versus optimizing for playout strength.

In order to optimize a policy for being a strong playout policy, rather than a strong gameplay policy, we use the same exact procedure, except that when playing a game between π_o and π_k, we use MCTS agents where π_o and π_k are used as the playout policies.

Experiments and Results

In this section, we describe the experiment set up and results. Specifically, we report three experiments:

- Optimizing policies for gameplay strength,
- Optimizing policies for being good playouts,
- Comparing the resulting policies of the two previous experiments, to gain insights into what makes a good playout policy.

Three different maps are used to test the generalizability of our comparison. The maps (shown in Figure 2) are:

- Map 1: *8x8/basesWorkers8x8A.xml*: In this map of size 8 by 8, each player starts with one base and one worker. Games are cut-off at 3000 cycles.
- Map 2: *8x8/FourBasesWorkers8x8.xml*: In this map of size 8 by 8, each player starts with four bases and four worker. Games are cut-off at 3000 cycles.
- Map 3: *NoWhereToRun9x8.xml*: In this map of size nine by eight, each player starts with one base and the players are initially separated by a wall of resources, that needs to be mined through in order to reach each other. Games are cut-off at 3000 cycles.

Bandit Optimization for Gameplay Strength

In the first experiment, we optimize directly for gameplay strength of the policy for each map. For gameplay policy optimization, we run $T = 500$ iterations of fictitious play. And during each iteration of fictitious play, we run a bandit optimization of $K = 2000$ iterations. The final policy is obtained by optimizing against the final Nash Equilibrium N of fictitious play for 50000 iterations of NaïveSampling. To reduce variance, instead of running a single game, we run 20 games in parallel and use the average as the actual reward.

Figure 3 shows the gameplay strength of the optimized policies in each of the three maps against two baseline policies: `Rnd` and `RndBiased` which correspond to two base agents included in μRTS (RandomAI and RandomBiasedAI), and against the policies optimized as playouts (which we will describe in the next section). `Rnd` is a random agent with uniform weight vector and `RndBiased` is a biased random agent where HARVEST, RETURN, and ATTACK has five times the weights than other action types (approximate weight vector $[0.06, 0.06, 0.28, 0.28, 0.06, 0.28]$). We report the win-rate in a round-robin tournament repeated 200 times. So in each optimization and each map, each policy plays 600 games.

The gameplay-optimized policies achieved 0.997, 1.00, and 0.862 winrate respectively in the three maps, showing that when compared against the other policies in the policy space, these are, as expected very strong policies compared to the baselines. These results provide a starting point for comparison.

Bandit Optimization for Playouts

For playout policy optimization, due to the fact that it is extremely slow with MCTS in the loop, we only run $T = 20$ iterations of fictious play and $K = 1000$ iterations of bandit optimization. The final policy is obtained by optimizing against the final Nash Equilibrium N of fictitious play for 10000 iterations of NaïveSampling. To reduce variance, instead of running a single game, we run 20 games in parallel and use the average as the actual reward.

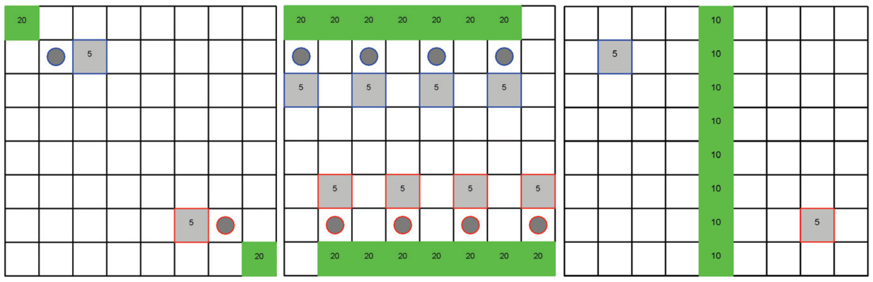
Figure 2: The three maps used in our experiments

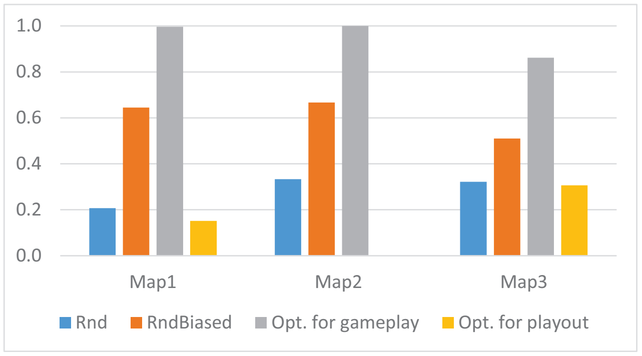

Figure 3: Round-robin Winrate Comparison of Optimized Policies as Gameplay Policies

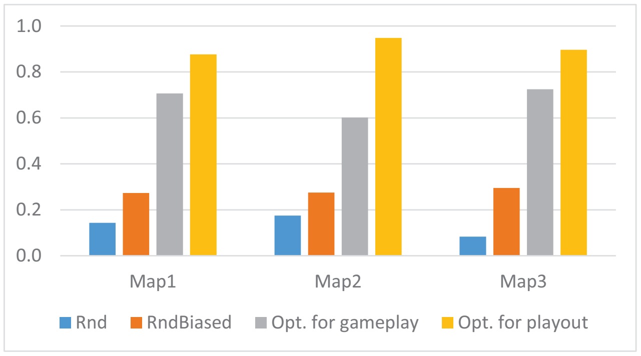

Figure 4: Round-robin Winrate Comparison of MCTS Agents using Optimized Policies as Playout Policies

During gameplay, we severely limit the computation budget of MCTS in order to reduce computational cost, and only allow the agents 100 iterations of MCTS per move (which is very low). The playout policy simulates forward for 100 game cycles and returns the state evaluation from the predefined evaluation function in the standard implementation of NaïveMCTS in μRTS.

Looking again at Figure 3, which shows a comparison of the gameplay strength of all the optimized policies. We see that policies optimized as playout policies are very weak when it comes to playing the game. For example, in map2, the policy that was optimized for playouts, won 0% of the games. It was not even able to win a single game against the `Rnd` agent, which just selects actions at random.

The picture looks very different when we look at Figure 4, which shows the overall winrate of a similar round-robin experiment, but this time when used as playout policies of a NaïveMCTS agent. What we can see is that `RndBiased` (the default playout policy of NaïveMCTS in μRTS) is a better playout policy than `Rnd`. The policy optimized for gameplay strength is a much better policy still, but the policy optimized for playout even stronger. This is very interesting, as it shows that there are policies that are very weak when used as gameplay policies result in very strong performance when used as playout policies. For example, pair-wise win rates in these round-robin tournaments are reported in Tables 1 and 2, where we can see that NaïveMCTS using a policy optimized for playouts defeated standard NaïveMCTS (which uses `RndBiased` as the playout policy), 96%, 97% and 99% of the times in the three maps respectively!

Comparing Gameplay Policies vs. Playout Policies

We have now seen that policies optimized as gameplay policies achieve very strong gameplay, and are better than random policies for playouts, but we can get even stronger playout policies by directly optimizing for this. Let us compare the actual policies that resulted of our optimization processes to gain insights into the types of policies that are good for each task.

The Left-hand side of Figure 5 shows a visualization of the three resulting policies for gameplay strength optimization, which correspond to the following three policies:

- In Map 1: $[0.00, 0.00, 0.20, 0.20, 0.30, 0.30]$
- In Map 2: $[0.00, 0.00, 0.00, 0.25, 0.00, 0.75]$
- In Map 3: $[0.00, 0.00, 0.07, 0.29, 0.29, 0.35]$

The middle three policies of Figure 5 represent the three playout optimized policies for each of the maps, corresponding to the following three policies:

- In Map 1: $[0.11, 0.00, 0.00, 0.11, 0.22, 0.56]$
- In Map 2: $[0.67, 0.00, 0.00, 0.17, 0.17, 0.00]$
- In Map 3: $[0.06, 0.06, 0.28, 0.28, 0.06, 0.28]$

Finally, the right-hand side shows `Rnd` and `RndBiased`.

Table 1: Pairwise gameplay winrate comparisons for maps 1, 2, and 3.

	Rnd	RndBiased	Gameplay	Playout
Rnd	-	0.05	0.00	0.58
RndBiased	0.95	-	0.01	0.97
Gameplay	1.00	0.99	-	1.00
Playout	0.42	0.03	0.00	-
	Rnd	RndBiased	Gameplay	Playout
Rnd	-	0.00	0.00	1.00
RndBiased	1.00	-	0.00	1.00
Gameplay	1.00	1.00	-	1.00
Playout	0.00	0.00	0.00	-
	Rnd	RndBiased	Gameplay	Playout
Rnd	-	0.36	0.10	0.50
RndBiased	0.64	-	0.19	0.70
Gameplay	0.90	0.81	-	0.88
Playout	0.50	0.30	0.12	-

Table 2: Pairwise winrate comparisons for maps 1, 2 and 3, when using the policies as playout policies for NaïveMCTS.

	Rnd	RndBiased	Gameplay	Playout
Rnd	-	0.34	0.06	0.04
RndBiased	0.66	-	0.12	0.04
Gameplay	0.94	0.88	-	0.29
Playout	0.96	0.96	0.71	-
	Rnd	RndBiased	Gameplay	Playout
Rnd	-	0.37	0.15	0.00
RndBiased	0.63	-	0.17	0.03
Gameplay	0.85	0.83	-	0.12
Playout	1.00	0.97	0.88	-
	Rnd	RndBiased	Gameplay	Playout
Rnd	-	0.20	0.02	0.04
RndBiased	0.80	-	0.07	0.01
Gameplay	0.98	0.93	-	0.26
Playout	0.96	0.99	0.74	-

An interesting thing to note is that the weight for the NONE and MOVE actions in policies optimized for gameplay is always 0. Which basically means that what the optimization process has found is that if there is any other action that is available, that should be preferable to either MOVE or NONE. And an extreme policy is the optimal policy for map2, where, since the agents already start with 4 workers, which is enough to attack, full priority is given to attack, and weight 0 is given to even harvesting new resources. However, when optimizing for playout, 0 weight is given to ATTACK in map2, and NONE has the highest weight instead. Moreover, both for gameplay nor playouts, no policy set a weight different than 0 for the MOVE action, which means that moving units is only done if none of the other actions with weight higher than 0 are available.

Map2 is interesting. It is the most "chaotic" map of the three we used, as players start very close to each other. We hypothesize, thus that there is a high degree of uncertainty in playout evaluations in this map, as combat results from stochastic playouts can vary wildly. Thus, in this map, the optimization process has opted for a playout policy that

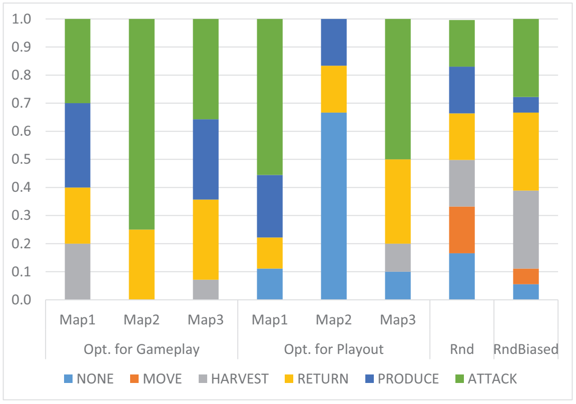

Figure 5: Resulting Weight Vectors for Each Map from Both Optimization Process

mostly keeps units standing still (executing the NONE action, which makes units just wait doing nothing). In this way, attacks are minimized during playouts, making the evaluation function be applied to a state that is more similar to the state right before the playout started, which might result in a more stable evaluation. In maps that are less "chaotic", such as map1 and map3, however, playout policies include a strong weight for attack.

Another interesting result is that policies optimized for different maps are very different, and thus, subsequent studies should look into generalized policies for groups of maps.

Conclusions

The long term goal of this paper is to understand playout policies in MCTS and thus provide guidance to designing more effective playout policies. Specifically, we report an empirical study on optimizing and comparing policies with two different goals, gameplay strength and playout strength in the domain of RTS games.

By optimizing very simple parametrized policies, our results suggest that there is a major difference between the policies that are optimized for different objectives. Furthermore, compared to baseline policies, playout-optimized policies performs well as playout policy, as expected, but very poorly as gameplay policy. Also, for our simple policy parameterization, optimizing policies for gameplay strength is better than using random policies, but is not as good as optimizing them directly for playouts. Note that it still needs to be investigate if the same holds in other domains or more complex policy parameterization.

For future work, we would like to further investigate the optimization process of the playout policies, with the long term goal of obtaining an explicit optimization objective that a policy should be optimized for in order to be a good playout policy (rather than the computationally costly process used in this paper, of optimizing them using MCTS in the optimization loop). A potential direction is to adapt simulation balancing into the domain of RTS games where the action space is combinatorially structured.

References

Andersen, P.-A.; Goodwin, M.; and Granmo, O.-C. 2018. Deep RTS: A game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Baier, H., and Drake, P. D. 2010. The power of forgetting: Improving the last-good-reply policy in monte carlo go. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4):303–309.

Brown, G. W. 1951. Iterative solution of games by fictitious play. *Activity analysis of production and allocation* 13(1):374–376.

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. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

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

Coulom, R. 2007. Computing “ELO ratings” of move patterns in the game of go. *ICGA journal* 30(4):198–208.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, 273–280.

Graf, T., and Platzner, M. 2016. Monte-Carlo simulation balancing revisited. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 186–192. IEEE.

Huang, S.-C.; Coulom, R.; and Lin, S.-S. 2010. Monte-Carlo simulation balancing in practice. In *International Conference on Computers and Games*, 81–92. Springer.

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

Munos, S. G. W., and Teytaud, O. 2006. Modification of UCT with patterns in monte-carlo go. *Technical Report RR-6062* 32:30–56.

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

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ontañón, S. 2017. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research* 58:665–702.

Shleyfman, A.; Komenda, A.; and Domshlak, C. 2014. On combinatorial actions and CMABs with linear side information. In *ECAI*, 825–830.

Silver, D., and Tesauro, G. 2009. Monte-Carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 945–952.

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.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354–359.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Silver, D.; Sutton, R. S.; and Müller, M. 2012. Temporal-difference search in computer go. *Machine learning* 87(2):183–219.

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*.

Tian, Y.; Gong, Q.; Shang, W.; Wu, Y.; and Zitnick, C. L. 2017. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)*.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. StarCraft II: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.