Bandit-Based Policy Optimization for Monte Carlo Tree Search in RTS Games

Zuozhi Yang[1] and **Santiago Ontañón**[1,2]
[1] Drexel University, Philadelphia, USA
[2] Google AI, 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 are still many questions to be answered regarding the general principles to design or learn its playout policy, or the interaction between tree policy and playout policy. Many systems, such as AlphaGo, use a policy optimized to mimic human expert is used as the playout policy of MCTS. In our recent work, we have shown that strong gameplay policies do not necessarily make the best playout policies. In this paper, we take a step further and use bandit algorithms to optimize stochastic policies as gameplay policies, tree policies, and playout policies for MCTS in the context of RTS games. Our results show that strong playout policies do not need to be strong gameplay policies, and that policies that maximize MCTS performance as playout policies are actually weak in terms of gameplay strength. Also, we found optimizing tree policy directly has an edge over optimizing gameplay policy. Finally, we showed that the joint optimization of tree policy and playout policy could be beneficial to the overall performance compared to optimization separately.

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 superhuman performance (Silver et al. 2016). In AlphaGo, a policy optimized for gameplay strength is used as the playout policy of MCTS. However, previous work has shown that having good gameplay strength is not a sufficient condition to be 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 are some times abandoned and replaced by refined evaluation functions (Silver et al. 2017b; 2017a). This paper extends our previous work (Yang and Ontañón 2020), where the authors showed that weak policies can also be strong policies.

Specifically, in this paper we evaluate the difference in behavior of game-playing policies when optimized for gameplay strength, for playout policy performance, and as tree policies. Since our goal is just to understand what makes a good playout or tree 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 different objectives: 1) winrate of the policy directly, and 2) win rate of an MCTS agent when using the policy as the playout or tree policy.

The rest of the paper is structured as follows. First, we provide background on RTS games, MCTS, and policy optimization. Then we describe the baseline, and our approach for optimizing gameplay policy, tree policy, and playout policies and also joint optimization of tree policy and playout policy. We show visualizations of the distributions of the trained policies, then compare them with each other and with baseline policies. Finally, we draw conclusions and discuss lines of future work.

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

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 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. 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 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 (bases, barracks, workers, and light, heavy and ranged military units). 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 and those with red outline belong to player 2. 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.

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.

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).

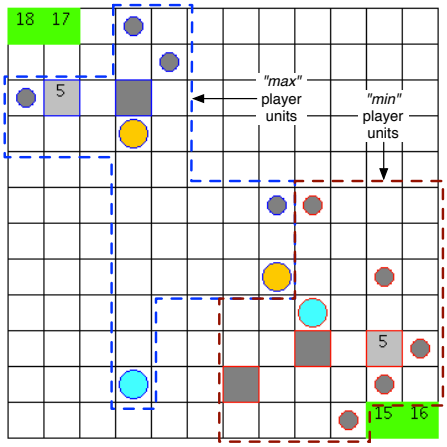


Figure 1: A Screenshot of μRTS.

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 always possible. 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, 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 directly as playout policies and tree 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, parameterized by larger parameter vectors, such as those represented by a neural network, for example (although a different optimization algorithm might be required, such as reinforcement learning).

Policy Optimization

Given the parameterization, we can optimize the policy for many purposes using different optimization algorithms. In this paper, we use repeated game of bandits (RGB) (Cesa-Bianchi and Lugosi 2006; Slivkins 2019). RGB works as in Algorithm 1, where two regret-minimizing agents repeatedly play against each other. And if the repeated game is zero-sum, the empirical distribution of RGB converges to Nash Equilibrium. The motivation is that 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 RGB, 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 RGB.

In particular, in this work we use multiarmed bandits as a way to compute the best response. For bandit optimization, we discretized the search space, allowing each weight to take values in $\{0, 1, 2, 3, 4, 5\}$. Specifically, we model the problem using combinatorial bandits, since the problem has a combinatorial structure where there are 6 types of actions and for each action type there are 6 different weights to choose from. 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).

Zero-Sum Repeated Game of Bandits In order to find the optimal policy within the space of policies defined by our 6-parameter vector, we use Naïve Sampling within the RGB play framework. Specifically, we use Algorithm 1. Given a target set of maps m, we use RGB as follows. We initialize a set of policies N. And then execute T iterations of repeated games between two regret-minimizing bandit agents B_1 and B_2. At each iteration k, two arms, π_k^1 and π_k^2, are pulled from each bandit independently and simultaneously. Then 10 games are played between the policies and the averaged reward $r \in [0, 1]$ is revealed to both bandits (r to B_1, $1 - r$ to B_2). Both of the selected arms are added the N. As we discussed above, it has been shown that N converges to the Nash Equilibrium.

However, in this study, we stick to the single policy for analysis and 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. The final weight vector will be the most visited arm after the bandit optimization process.

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_k^1 and π_k^2, we use MCTS agents where π_k^1 and π_k^2 are used as the playout policies.

Furthermore, we also experiment with optimizing the policy as the tree policy of the MCTS, in order to observe its difference to policies optimized for game-playing strength. The research question to ask is whether it is enough to optimize only for game-playing strength to have a good playout or tree policy.

Finally, we optimize the tree policy and playout policy directly at the same time. The purpose is to see if there are possible interactions between the two types of policies and potentially obtain policy combinations that work better than optimizing them separately.

Experiments and Results

In our previous work (Yang and Ontañón 2020) we presented the result of bandit optimized policies of the same parameterization. However, we did not compare with policies optimized using other techniques, such as *simulation*

Algorithm 1: Repeated Game of Bandits (with Naïve Sampling)

Initialize Nash Equilibrium strategy set $N = \emptyset$.
Initialize two bandit agents B_1 and B_2.
CMAB_1 = new NaïveSampling() bandit
CMAB_2 = new NaïveSampling() bandit
for $k = 1, 2, 3, \ldots, T$ **do**
  Choose arm π_k^1 = CMAB_1.sample()
  Choose arm π_k^2 = CMAB_2.sample()
  r = play a game π_k^1 vs π_k^2 in map m
  CMAB_1.observeReward(π_k^1, r)
  CMAB_2.observeReward($\pi_k^2, 1 - r$)
  $N \leftarrow N \cup \{\pi_k^1, \pi_k^2\}$

balancing (Silver and Tesauro 2009) in order to assess if just using simulation balancing is enough to obtain strong playout policies. Thus, in this paper, we first establish a baseline using simulation balancing and show that it does not scale well in RTS games. Then, we further investigate the bandit based optimization approach and the effect of the different optimization objectives describe above.

Three different maps are used to test the generalizability of our comparison. The maps 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.

Monte Carlo Simulation Balancing

Simulation Balancing (SB) (Silver and Tesauro 2009; Huang, Coulom, and Lin 2010; Graf and Platzner 2016) approach the problem of optimizing for good playout policy by not optimizing policy strength but optimizing policy balance. It is a policy gradient-based method that minimizes "imbalance" in the policies so that so that the small errors cancel each other out during the whole playout. The pseudocode is given in Algorithm 2. The algorithm first constructs a training set of state/state value pairs. The true state value can be estimated by performing a deep MCTS search when expert play is not available Then the algorithm uses Monte Carlo simulation to calculate the actual state value estimation of the given policy. Finally, the algorithm calculates the difference of the true state value and estimated state value to do policy gradient update. The policy gradient $\psi_{\theta_t}(s_n, a_n)$ is the following

$$\psi_{\theta_t}(s_n, a_n) = \nabla_\theta \log \pi_\theta(s, a) = \phi(s, a) - \sum_b \pi_\theta(s, b)\phi(s, b)$$

Algorithm 2: Simulation Balancing

$\theta \leftarrow 0$
for $t = 0$ *to* T **do**
  $(s_1, V^*(s_1)) \leftarrow$ Random choice from training set
  $V \leftarrow 0$
  for $j = 0$ *to* M **do**
    simulate $(s_1, a_1, \cdots, s_N, a_N, z)$ following π_{θ_t}
    $V \leftarrow V + z$
  $V \leftarrow \frac{V}{M}$
  for $i = 0$ *to* M **do**
    simulate $(s_1, a_1, \cdots, s_N, a_N, z)$ following π_{θ_t}
    $g \leftarrow g + z \sum_{n=1}^{N} \psi_{\theta_t}(s_n, a_n)$
  $g \leftarrow \frac{g}{M}$
  $\theta_{t+1} \leftarrow \theta_t + \alpha(V^*(s_1) - V)g$

In the equation, ϕ is the feature vector and θ is the policy.

Now we experiment the performance of SB. We first collect a dataset of estimated true state values from the three maps using NaïveMCTS of 100000 iterations and the value estimation of the root node is recorded as the estimated true state value. 1000 states are sampled from 200 self-played games of two random agents. During training, we first calculate the state value estimated by the playout policy V by averaging 1000 playouts. Then we run another 1000 playouts to calculate policy gradient ψ_{θ_t}. The resulting policy of SB optimization is characterized by the parameter vector $[0.02, 0.32, 0.18, 0.18, 0.17, 0.13]$. Together with a purely random and the built-in RandomBiased bot in μRTS, the result from SB will be used as the baseline in our study.

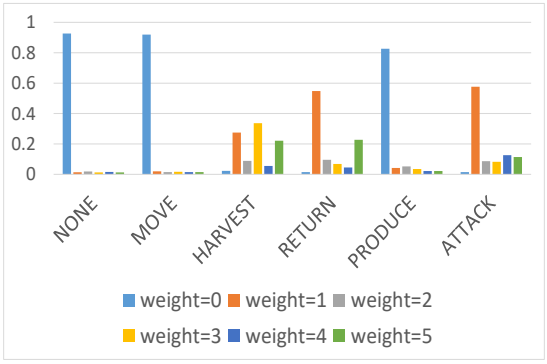
Optimization for Gameplay Strength

In the first experiment, we optimize the policy with multiple maps together and compare with the policies in (Yang and Ontañón 2020). Specifically, we run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents to obtain a history distribution of the process. The arms pulled by the two bandits correspond to the gameplaying policies and play against each other for 10 games to calculate the reward. The result is 20000 policies (the policy of each of the two players over 10000 itertions). We visualized the weight distribution of these 20000 policies.

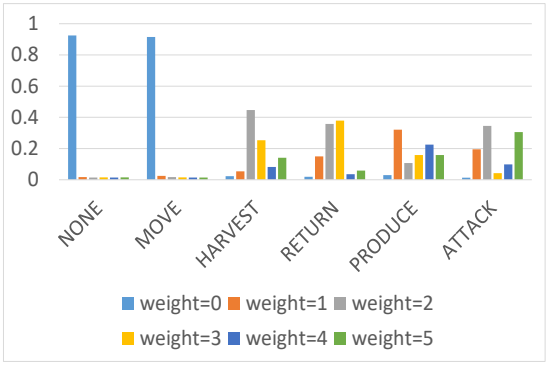
The result for gameplay strength optimization is shown in Figure 2-a. we observe that NONE, MOVE, and PRODUCE are mostly assigned a 0 weight in most of the policies in the distribution. RETURN and ATTACK are mostly given weight of 1. HARVEST and RETURN are given more diverse weights, probably due to the fact that we use different maps, and some values might work better in some maps than in others. Later in the paper, we will evaluate how strong these policies are in actual gameplay.

Optimization for Tree Policy

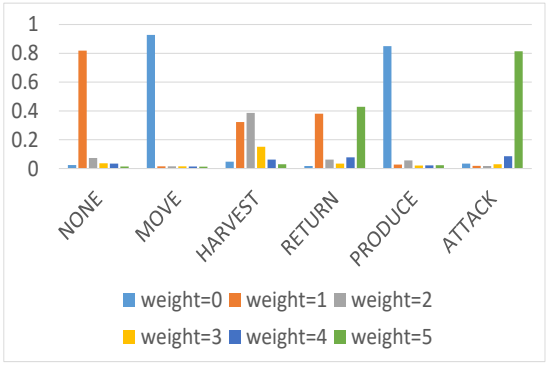
In the second experiment we optimize the tree policy directly as opposed to optimizing for gameplay strength. The



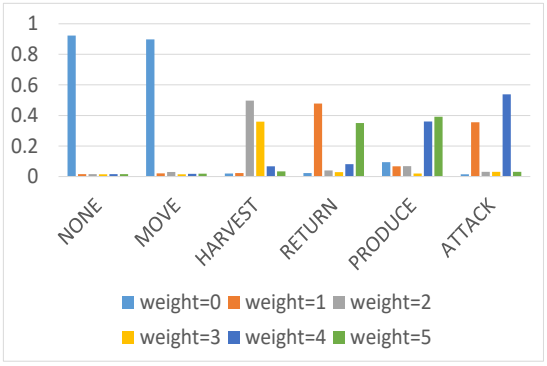
(a) Gameplay Strength Optimization



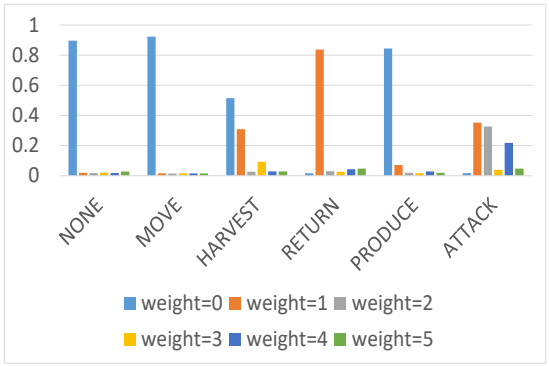
(b) Tree Policy Optimization



(c) Playout Policy Optimization



(d) Tree Policy in the Joint Optimization



(e) Playout Policy in the Joint Optimization

Figure 2: Weight distribution by action types in the history distribution for different optimization objectives.

experimental setup is similar to the gameplay strength optimization but the performance of the policies are measured directly by useing them as tree policies of MCTS. Again, we run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents. The arms pulled by the two bandits are used as tree policies and used by MCTS agents to play against each other for 10 games to calculate the reward.

The result of the optimization is visualized in Figure 2-b. It is easy to see that the results agree with the gameplay strength optimization that NONE and MOVE should assign a 0 weight with high probability, but disagree that PRODUCE should have a low probability for 0. Also, HARVEST, RETURN, and ATTACK have more spread weights than gameplay optimization. This shows that strong gameplay policies tend to be different from strong tree policies.

Optimization for Playout Policy

In the third experiment we optimize for the playout policy with a similar experimental set up as in tree policy optimization. We run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents and the arms pulled are interpret as playout policies and used by MCTS agents to play against each other for 10 games to calculate the reward.

The result of the optimization is visualized in Figure 2-c. We can observe that the weight distribution is very different to the distribution of optimization of tree policy or gameplay

policy. In this weight distribution, NONE is mostly assigned to weight 1. MOVE and PRODUCE are mostly assigned weight 0. And ATTACK is mostly assigned to the highest weight of 5. HARVEST is spread between 1, 2, and 3. RETURN has most of the weights assigned to 1 and 5. Again, we see that strong playout policies are very different from strong gameplay policies.

Joint Optimization for Tree Policy and Playout Policy

To test whether tree policy and playout policy interact with each other, we further investigate by optimizing both at the same time. Similarly, We run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents that choose values for 12 parameters rather than 6, and the arms pulled will be interpret as two policies, one for tree policy and the other for playout policies, and used by MCTS agents to play against each other for 10 games to calculate the reward. Thus, in this experiment, arms pulled by bandits have 12 parameters and the first six parameters are interpret as tree policy and others are interpret as playout policy.

The result of the optimization is visualized in Figure 2-d and Figure 2-e, showing that the jointly optimized policies are different from the policies obtained when optimizing them separately. Let us now compare how strong these policies are in actual gameplay.

Comparing Performance as Gameplay Policies vs. Playout Policies

So far we have policies optimized for different objectives:

- Optimizing "simulation balance".
- Optimizing gameplay strength of the policy.
- Optimized as tree policy of MCTS.
- Optimized as playout policy of MCTS.
- Joint optimization of tree policy and playout policy.

Now, together with the two baselines, Random and RandomBiased, we compare them policies in two tasks: gameplay strength when used directly to play (without MCTS), and gameplay strength when used as playout policies within MCTS. We run 10 rounds of round-robin between all the policies. The winrates are reported in Figure 3 (we tested the tree policies separately as reported below).

First, simulation balancing has a winrate of 0.16 as gameplaying policy and a winrate of 0.29 as the playout policy, which are outperformed by the two baselines as gameplaying policy (winrate of 0.20 and 0.37 respectively), but is better than baselines as playout policy of MCTS (winrate of 0.02 and 0.09 respectively). This is expected, as simulation balancing is supposed to design strong playout policies.

Second, the policy optimized for gameplay strength outperformed the baselines by a large margin and has the best gameplay winrate (0.58) and third best winrate as playout policy (0.52). For performance as playout policy, it is only worse than the two optimized as playout policies directly. The policy optimized as tree policy of MCTS also outperformed baselines, but has worse winrate than gameplay optimized policy in both tracks (winrate of 0.53 and 0.47).

Now we look at the policy optimized for playout policy directly. The result is interesting since it is very weak in terms of gameplay (winrate of merely 0.08), but very strong as playout policy (winrate of 0.62). This suggest that a strong gameplay strength is not a requirement of being a good playout policy, and that simulation balancing does not capture all that is required for a strong playout policy.

Lastly, we have the pair of policies that are optimized together, one as tree policy and the other as playout policy. The tree policy achieved similar winrates (winrate of 0.55 and 0.45 respectively) as singly optimized. The policy optimized as playout policy is interesting that not only it is good as playout policy (winrate of 0.56), but also it has a good gameplay strength (winrate of 0.52).

Strength of Tree Policies

The result of the jointly optimized policies suggest there could be some factor of "match" between the tree policy and the playout policy for them to work well together. Thus, to further verify this hypothesis, we take the best pairs of singly optimized tree policy and playout policies to play against the pair of jointly optimized policies in two MCTS agents.

We run the jointly optimized pair against the pair of best gameplay policy (as tree policy) and best singly optimized playout policy (as playout policy) for 1000 games, and the jointly optimized pair has a winrate of **0.64**. We also run the jointly optimized pair against the pair of best tree policy and best singly optimized playout policy for 1000 games, and the jointly optimized pair has a winrate of **0.67**.

Moreover, we run gameplay optimized policy against an optimized tree policy as the tree policy of an MCTS agent for 1000 games, both with the optimized playout policy. We found the gameplay policy has a winrate of **0.44**, which means that a gameplay optimized policy does not necessarily make for a good tree policy.

Conclusions

In this paper, we have studied policy optimization in several settings. First, we tried simulation balancing for playout policy optimization. We found that although it is better than the baselines as playout policy, its performance is not comparable to optimizing as playout policy directly. We also tried optimizing game policy, tree policy, and playout policy in three maps at the same time. We observed that for some action types like NONE and MOVE, the weight distribution are in consensus for all maps, but for others, weight distributions is spread to multiple categories. This might be because certain weights are good for some maps. Furthermore, we compared the performance as tree policy between optimized gameplay policy and optimized tree policy, and confirmed that optimize tree policy directly does help. Finally, we optimized the tree policy and playout policy jointly. The resulting pair of policies outperforms the combination of the best of tree policy and playout policies, which suggest that the "match" of the tree policy and playout policy can also play an important role in the performance of the MCTS.

For future work, we want to further investigate the simulation balancing algorithm, since there has been good advance

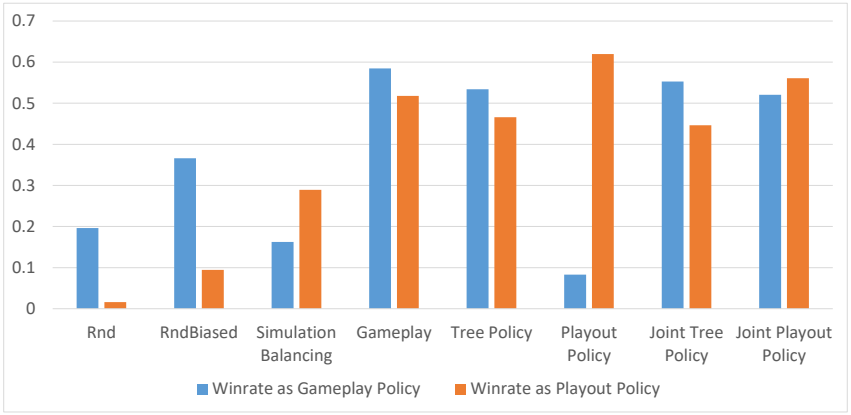


Figure 3: Comparison on winrates of policies serving as game-playing policy and playout policy of an MCTS agent.

in gradient policy algorithms that might help scaling up SB. Also, the joint optimization of different component of the MCTS algorithm seemed to be beneficial. It will be interesting to take more factors, like the evaluation function tuning and exploration parameters, into the optimization process to see if we can push the progress further and gain insight on the interplay between the different pieces of MCTS.

References

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.

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.

Cesa-Bianchi, N., and Lugosi, G. 2006. *Prediction, learning, and games*. Cambridge university press.

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)*, 1–7. 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. 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.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv* abs/1712.01815.

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

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

Slivkins, A. 2019. Introduction to multi-armed bandits. *arXiv preprint arXiv:1904.07272*.

Yang, Z., and Ontañón, S. 2020. Are strong policies also good playout policies? playout policy optimization for rts games. In *Sixteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.