All-Moves-As-First Heuristic in Monte Carlo Pommerman

Eid Alshareeda, Abdulnour Ali Hassan Dualeh, Shucheng Huang

School of EECS, Queen Mary University of London, UK

Abstract

In this paper, we enhance the MCTS agent in Pommerman with three types of AMAF heuristics. Comparisons by win rate against other agents are made among the three, and we also tune the parameters including α, C (in the UCB1 function) and rollout depth to further improve win rate. We conclude that, using α-AMAF, the plain MCTS agent can be effectively improved in both FFA and Team mode with both full and partial observability. And tuning parameters upon that allows us to yield a total increase in win rate of 10.4 percentage points in FFA mode with full observability.

1. Introduction

The study of AI is centered around the question of how to build agents that can always make the right decision in a complex environment. To that end, various kinds of intelligent agents have been designed to play computer games, which pose interesting challenges for agents to solve. Most of the time, the right action to take is not immediately obvious, thus requiring an agent to perform certain kinds of search to find it. In the context of game AI, the search typically involves using a forward model (FM) to simulate a number of future game states and evaluating them statistically. This general approach is known as statistical forward planning (SFP).

Monte Carlo tree search (MCTS) currently represents the state-of-the-art search technique in solving combinatorial games, of which the most studied is Go. More than that, many powerful extensions and variants of MCTS are first proposed for and tested in Go. However, the effectiveness of some of them has not yet been made clear in some other games, such as Pommerman.

In this paper, we address the problem of enhancing the existing MCTS agent in the Pommerman Java framework using All-Moves-As-First (AMAF) heuristic. Section 2 introduces the rules and features of Pommerman, the Java framework, and prior research into SFP methods for this game. Section 3 briefly explains the idea of MCTS algorithm and its AMAF variant. Then in Section 4 we describe our actual implementation of AMAF in the framework and how we organize our exploration into two steps; and in Section 5 we present the experiment results and go on to discuss the possible causes of such results in Section 6. Finally we conclude the improvements we have made to the MCTS agent and further problems worth studying in future.

2. Pommerman

Pommerman is a complex computer multi-agent combinatorial game derived from the famous Bomberman made by Hudson Soft in 1983. The game takes place on an 11 x 11 board with randomly-scattered wooden or rigid obstacles. At every tick, each of the four players takes an action out of the six available ones: STOP, UP, DOWN, LEFT, RIGHT, and BOMB. A dropped bomb explodes in 10 ticks and generates a cross-shaped flame lasting for 5 ticks, which is the only means to eliminate an opponent. And the objective of the game is to be the last player standing (in FFA mode) or the last team to have at least one player standing (in Team mode), within a maximum time of 800 ticks. A tie occurs when more than one player is alive (in FFA mode) or both teams are alive (in Team mode) at the end of the game. The wooden obstacles, when destroyed by flames, may reveal pick-ups: additional bomb ammo, additional flame range, or ability to kick bombs away. To add to the complexity, the game can be run in full or partial observability.

The experiments presented in this paper are conducted in the Java framework of this game. According to Perez et al. [1], this optimized implementation runs 45x faster than the original Python one. This allows us to carry out larger tests and more experiments, obtaining results that are more statistically reliable. The framework has an automatic testing program that takes a specific game configuration, runs repeated games and calculates the overall win rates of each agent.

Built-in in the framework are an FM, heuristics (used to evaluate a simulated game state), and agents based on different algorithms—MCTS, One-Step-Look-Ahead (OSLA), Simple Rule-based, and Rolling-Horizon-Evolutionary-Algorithm (RHEA). The Simple agent acts according to a set of pre-defined rules, and can therefore be regarded as a reflex model. The OSLA agent, in planning forward, only evaluates the six immediately next available actions but not any further. The RHEA agent plans forward using evolution in real-time, with 28 exposed parameters (all kept at default during experimentation). The MCTS agent plans by the MCTS technique described in the next section and has 3 parameters exposed: K, depth, and budget, which are to be tuned in section 5. At every tick, the game engine exposes the latest state, the FM and the heuristics to each agent, so that agents can process state information as per their algorithms to decide on an action to execute.

Since 2018, Pommerman has been used to hold AI competitions by NeurIPS, and extensive comparative studies have been made between different agents in the game. Notably, Perez et al. [1] made a statistical analysis of the performances of the MCTS and RHEA agents,

summarizing that both agents are significantly stronger than OSLA and Simple agents and that MCTS seems to deliver a better performance than RHEA does under their test configuration.

3. Background

MCTS is a tree search algorithm based on the Monte-Carlo simulation method, which relies on the fact that sampling future game states for a sufficiently large number of times makes the evaluated game-theoretic values of these states converge to their mathematical expectation, thus enabling a rational decision-making process.

Every iteration of MCTS consists of four phases: Tree Selection, Expansion, Simulation and Back Propagation. Tree Selection uses a bound function (in this game, the UCB1) to choose a child node, keep moving down in the game tree until reaching a not fully explored node. Expansion adds a new child to it, and Simulation plays a uniform Monte-Carlo rollout from that until the end of the game or a pre-defined depth, which yields a reward that will be fed back into the in-node statistics of every node on the path previously determined by Tree Selection.

After a large number of iterations, the statistics in each node reflects a reasonable estimate of the quality of that node. Since Tree Selection biases the sampling gradually towards the higher-quality nodes, at the end of the algorithm, the most visited child node of the root state is the right action to take.

Changing the pre-defined rollout depth allows the agent to plan further ahead, and changing the K in the UCB1 function makes the agent favor known good paths over unknown ones more or less. These parameters can be tuned, but ultimately the accuracy of MCTS hinges on the number of Monte-Carlo playouts (i.e. the number of iterations).

AMAF is a popular variant of MCTS initially proposed by Brügmann [2] to enhance the performance of MCTS in playing Go. The idea of basic AMAF is to extend the eligibility for update in Back Propagation from only the nodes on the previously chosen path to also the siblings of those nodes, on the condition that the action represented by the sibling has been performed during the rollout. This would “warms up” the in-node statistics more rapidly than plain MCTS does, and the increased knowledge extracted from the rollouts is argued to be possibly beneficial to the statistics. However, Helmbold and Parker-Wood [3] noted that this may come at the cost of including irrelevant or biased knowledge.

Then the second type of AMAF is called α-AMAF. It maintains the regular node score separately from the AMAF score, and uses the parameter α to control how much the two scores should be blended. This parameterized approach gives us a chance to explore the effect of AMAF by tuning.

Last but not least, Rapid Action Value Estimation (RAVE) as the third type of AMAF, also uses the parameter α to control the blending, but in RAVE it decreases as the node get visited for increasingly more times. And the threshold when α reaches (and then stays at) zero is controlled by a new parameter N.

With the three types AMAF described, we can summarize that basic AMAF and the plain MCTS can be seen as two special cases of α-AMAF, namely when α=1 and α=0 respectively. So, fundamentally, the difference between the three AMAF methods is how they control the weight of the AMAF score against the regular score.

4. Method

Inside MCTS package of the Pommerman Java framework, SingleTreeNode.java holds the Java methods that collectively represents the MCTS algorithm; MCTSPlayer.java is the class that interfaces the game API with the MCTS algorithm; and MCTSParams.java holds all the parameters we are going to tune. Since we will need to let the AMAF agent play against the plain MCTS agent, we do not overwrite the existing code of MCTS agent. We implement our AMAF-enhanced MCTS agent in the duplicated classes named MCTSPlayer_2.java and SingleTreeNode_2.java, while the additional parameter α is inserted directly into MCTSParams.java since it does not cause interference.

The actual implementation involves modifying the method SingleTreeNode_2.uct() and SingleTreeNode_2.backUp(), as they control the Tree Selection policy and Back Propagation respectively. New instance variables are also added to the class SingleTreeNode_2, as we need to store AMAF scores separately from the regular scores.

The parameter tuning is done manually, by tweaking the value once at a time and putting it to automatic testing. But multiple tests are run in separate Java processes of the testing program in parallel.

We organize our experimentation into two steps. In Step 1, we implement basic AMAF, α-AMAF and RAVE agents and test them one by one against the original MCTS agents in both FFA and Team mode and in both full and partial observability. By the resultant win rates, we compare the three enhancing methods and choose the best performing one to go on to Step 2—parameter tuning. We tune manually according to the agent's win rate in FFA mode and full observability, so as to further improve its win rate.

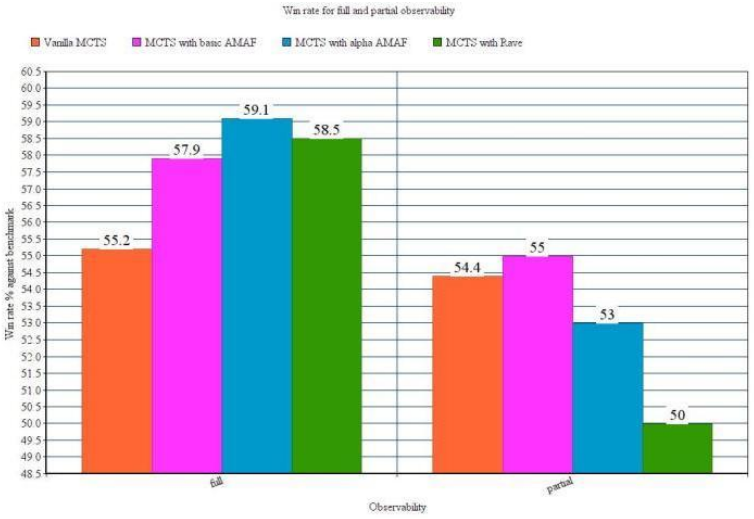
5. Experimental Study

We begin with our experimentation by setting the benchmark win rate. That is obtained by testing the original MCTS agent with the framework-default parameters against other three built-in agents in FFA mode and in both full and partial observability. And the test setting is 10 level seeds, each run 10 times. Benchmarking in Team mode is later done directly by testing the new agents, where the two benchmark agents and two tested agents are put into different teams.

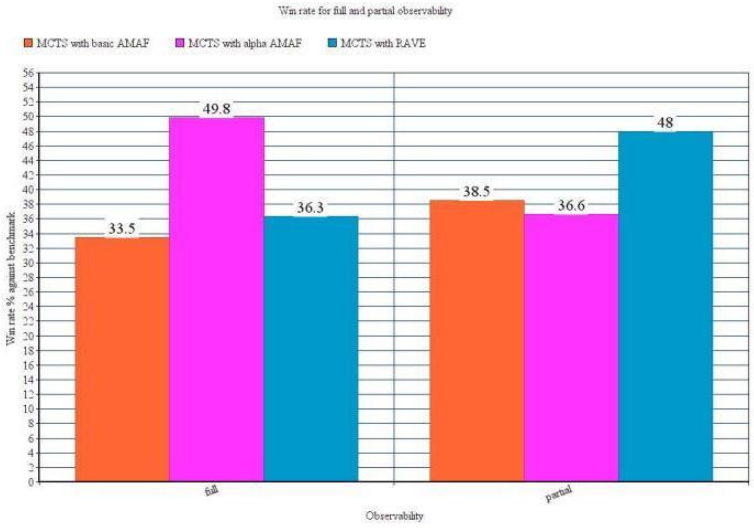
In implementing the α-AMAF, we tentatively set $\alpha = 0.4$; and in RAVE, we tentatively set $N = 20$. The reason we take 20 for N is based on our finding that when we trace the growth of the tree in the original MCTS, we see most depth-1 nodes are eventually visited for merely 30 times or so.

For testing of Step 1, we substitute the benchmark agent (player name "MCTS") with basic AMAF, α-AMAF and RAVE agents respectively (all represented by player name "MCTS_2")

and keep all other configurations the same. The following figure shows the two benchmark scores together with the scores of the three new agents.



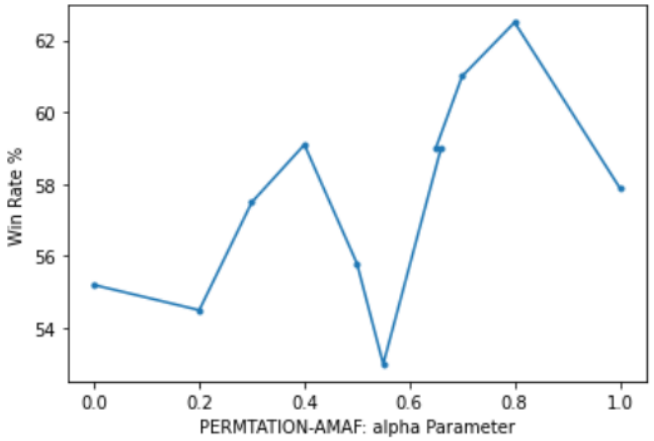
Then, in Team mode, we put two new agents and two benchmark agents into different teams and let them fight each other. Other test settings also remain the same, and both full and partial observability are used.



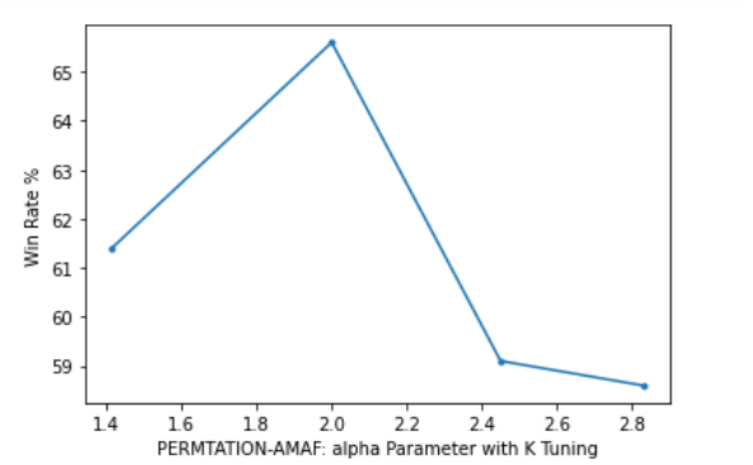
Combining the results on the above two charts, we can summarize that basic AMAF and RAVE does not perform well compared to α-AMAF. They even get beaten by the benchmark agent in some circumstances. By contraste, the α-AMAF generally shows a increase in win rate of about 4 percentage points in full observability, and performs just about as well as benchmark in partial observability. Additionally, α-AMAF is obviously advantageous over the benchmark agent. Thus, we decide to take α-AMAF into Step 2.

In Step 2, due to consideration of time-efficiency and ease of analysis, we only include FFA mode and full observability in testing, which we believe is the most basic circumstance and thus the starting point of tackling any Pommerman agent.

We firstly tune α. This is done through running 6 tests with α set to 0.2, 0.3, 0.5, 0.6, 0.7, 0.8 respectively. Recall that α=0 is namely the original MCTS, α=1 is namely basic AMAF, and α=0.4 is already tested in Step 1. Still, we can plot these three cases together with the 6 tests here to get a better idea of the impact of α.



From the chart we conclude that 0.8 is the optimal value for α. So we take this setting to further tune MCTS parameters including K and budget num of iteration. The reason we did not choose to tune rollout depth is also based on our finding about tree growth mentioned before. We think the number of visits that most nodes have got are way too few than what is ideal for Monte-Carlo simulation to show its power. Therefore the primary task of tuning should be to make the algorithm do more iterations, bumping up the statistics everywhere in the tree, instead of to make it plan too far ahead.



The framework-default value for K is √2 and for the budget number of iterations is 200. With further tuning efforts, we find that raising the number of iterations up to 400 is both safe and effective, whilst further going up to 600 or 800 causes the agent to timeout so often that the win rate is even worse than benchmark. And we find that K is best set at 2. Combining tuned K and tuned budget number of iterations, we eventually hit a win rate of 65.6%—an increase of about 6 percentage point compared to the untuned α-AMAF agent in Step 1.

6. Discussion

Through experimentation, we successfully enhanced the original MCTS agent using α-AMAF, even without any tuning. This is somewhat surprising, since the original design idea behind AMAF is that the moves in the game should be time– and state–independent. This may be true for games like Go, where the concept of "territory" matters. But in Pommerman, all the available actions are so primitive that we think it is hard to argue that their effect is independent of when and in which state they are executed. Yet we found the optimal α at 0.8, which suggests that the game actually "likes the flavor" of the AMAF. Although we do not quite understand this, one possible explanation could be that, for a given level, the specific way of the positioning of some obstacles somehow gives players near them an advantage or disadvantage. For example, if a player is initiated near the corner where there are many wooden obstacles, it may get pick-ups from them and gain advantages; whilst there may be some region on the board that tends to trap a player in and get it killed by a bomb.

As for the tuning of MCTS budget number of iterations, the result we get is largely in accordance with our theoretic analysis that the algorithm simply needs more rollouts to obtain more statistically reliable estimates.

7. Conclusions and Future Work

In this paper, we are successful in enhancing the original MCTS agent in the Pommerman Java framework using α-AMAF. The bare effect of adding α-AMAF is about a win rate increase of 4 percentage points in FFA mode and full observability. Combined with parameter tuning, the α-AMAF agent yields a total increase in win rate of 10 percentage points.

Most of the test statistics shown in this paper is obtained when the numbers do seem to have converged to a reliable degree. However, when we repeat some of the tests (especially Team mode tests), the win rate fluctuates greatly and does not show signs of convergence. This suggests that there is still much more to explore in Team mode, given that Team mode may allow much different survival strategies than FFA mode does. For example, exploring the use of teammate messages may be beneficial to further bumping up the performance.

References

1. D. Perez-Liebana, R. D. Gaina, O. Drageset, E. İlhan, M. Balla, and S. M. Lucas, "Analysis of Statistical Forward Planning Methods in Pommerman", AIIDE, vol. 15, no. 1, pp. 66-72, Oct. 2019.
2. Choe, J. & Kim, J. Enhance Monte Carlo Tree Search for playing Hearthstone. DOI: 10.1109/CIG.2019.8848034.
3. Sironi, C. and Winands, M. Comparing Randomization Strategies for Search-Control Parameters in Monte-Carlo Tree search. DOI: 10.1109/CIG.2019.8848056
4. Helmbold, D. and Parker-Wood, A. All-Moves-As-First Heuristics in Monte-Carlo Go. AMAFpaperWithRef.pdf (ucsc.edu).
5. Gero, K., Asktorab, Z., Dugan, C., Pan, Q., Johnson, J., Geyer, W., Ruiz, M., Miller, S., Millen, D., Campbell, M., Kumaravel, S & Zhang, W. Mental Models of AI Agents in A Cooperative Game Setting. https://doi.org/10.1145/3313831.337631