Improved MCTS Agent in Pommerman

Qizhou Deng

220209720
AI in Games

***Abstract*—Monte Carlo tree search, a method widely used in artificial intelligence, has attracted considerable interest due to its astonishing success in the computer game Go. It has also proven valuable in a range of other fields [1]. Pommerman is a multi-player game that combines simultaneous action choices and complete information. This paper mainly studies the Monte Carlo tree search enhancement under the Java framework to enhance the winning rate and scoring in the Pommerman.**

I. Introduction

This paper is going to use some AI agents to improve performance while playing Pommerman. Monte Carlo Tree Search (MCTS) is a highly selected best-first tree search algorithm that incrementally builds an asymmetric tree by adding a single node at a time, estimating its game-theoretic value by using self-play from the state of the node to the end of the game. The Monte Carlo method has significant success and long history for AI games playing algorithms, especially imperfect information games such as Scrabble and Bridge. However, Monte Carlo method made a massive success in computer Go by applying the tree-building procedure. Go is one of the few classic games in which human players are far ahead of computer players. Nevertheless, MCTS has had an enormous impact on filling this gap and is now competing with those world-leading human players on small boards, though MCTS falls far short of their level on the standard 19×19 board [1]. Over the last few years, MCTS has also gained great success with many specific games, available games, and complex real-world planning, optimization, and control problems. It looks to become an essential part of the AI researcher's toolkit. It can give an agent some decision-making ability with very little domain-specific knowledge. Its selective sampling approach may provide insights into how other algorithms could be [1] hybridized and potentially improved. In the next section, there will be a brief introduction to the Pommerman framework. Section IV will have some background and a literature review about MCTS. In section V, there will be a specific approach to how we implement the improvement. Section VI uses Pommer framwork's own methods to present the data and analyze the results. Section VII summarize the obtained data and have an overall discussion of the results. Section VIII concludes the work and findings of the project and introduces the potential future work.

II. Pommerman

Pommerman is a complex digital game that's a variation of the well-known Bomberman from 1983. The game occurs on an 11x11 board with randomly placed obstacles. Every player, of which there are four, makes a move from a choice of six actions: STOP, UP, DOWN, LEFT, RIGHT, and BOMB. Bombs explode after 10 ticks, creating a cross-shaped flame for 5 ticks. Players aim to eliminate opponents using these flames, striving to be the last one standing in Free-for-All mode or the last team with at least one player in Team mode, all within 800 ticks. Destroyed obstacles can reveal power-ups like extra bombs, longer flame range, or bomb kicking ability. The game's complexity is heightened by the option for partial or complete observability [2].

The game offers three distinct modes: Free For All (FFA), Team (TEAM), and Team Radio. In the FFA mode, all players compete individually, and victory goes to the last player standing. Players eliminated immediately are out of the game. Those still alive when the game concludes result in a tie, even if they're eliminated simultaneously on the final tick.

In the TEAM mode, players form pairs and are placed initially in opposite corners of the board. A team loses when both players on the team are eliminated, leading to victory for the opposing team. It's not mandatory for both team members to survive for a team to win. If both teams have surviving players when the tick limit is reached, both teams tie.

The game board is created through an automated process. Agents start in corners with some empty space around them. Wooden boxes are then positioned to enable passageways between players. The board is further formed by randomly placing 20 rigid (X) and 20 wooden (Y) tiles symmetrically. If the count of inaccessible tiles exceeds 4 (Z), the process is repeated. Additionally, 10 items (W) are randomly placed on selected wooden boxes, with their types also chosen at random. The values X, Y, Z, and W serve as parameters for the level generator, and these specific values are used in the experiments.

The framework includes several components such as a Forward Model (FM), heuristics for evaluating simulated game states, and agents utilizing diverse algorithms. These algorithms encompass Monte Carlo Tree Search (MCTS), One-Step-Look-Ahead (OSLA), Simple Rule-based, and Rolling-Horizon-Evolutionary-Algorithm (RHEA).

The Simple agent adheres to pre-defined rules, making it essentially a reflex model. The OSLA agent plans by evaluating only the immediate next available actions without considering subsequent steps. The RHEA agent employs real-time evolutionary planning, adjusting 28 exposed parameters (which are maintained at their default values during experiments). The MCTS agent employs the MCTS technique (elaborated in the next section) and exposes 3 parameters: K, depth, and budget, which will be optimized in latter section.

During each game tick, the game engine provides the latest state, the Forward Model, and the heuristics to each agent. Agents utilize this information based on their algorithms to make decisions about the action they should execute.

III. Background

Monte Carlo Tree Search (MCTS) is a method of locating optimal determinations in a given field by randomly testing models in decision space and building a search tree from the

consequences. It has profoundly influenced artificial intelligence (AI) methods in fields representing sequential decision trees, especially game and planning problems [1].

The basic MCTS process is conceptually straightforward, as shown in Fig. 1. A tree is built incrementally and asymmetrically. A tree policy is to find the most compulsory node of the present tree for each iteration of the algorithm. The tree policy tries to balance relations of exploration (look in zones that have not been well tested yet) and exploitation (look in zones that appear to be favourable). A simulation is that the search tree is updated according to the outcome after running from the selected node. This involves the addition of a child node coordinating with the action taken from the selected node and an update of the statistics of its ancestors. The simplest default policy of moving is to make uniform random moves. A great advantage of MCTS is that the values of middle states do not have to be assessed, as for depth-limited minimax search, which significantly decreases the amount of domain knowledge required. Only the value of the terminal state at the end of each simulation is required [3].

MCTS is an algorithm that consists of repeating a sequence of 4 steps:

Four steps are applied per search iteration [1].

- Selection: Starting at the root node, a child selection policy is recursively applied to descend through the tree until the most urgent expandable node is reached. A node is expandable if it represents a nonterminal state and has unvisited (i.e., unexpanded) children.
- Expansion: One (or more) child nodes are added to expand the tree, according to the available actions.
- Simulation: A simulation is run from the new node(s) according to the default policy to produce an outcome.
- Backpropagation: The simulation result is “backed up” (i.e., backpropagated) through the selected nodes to update their statistics.

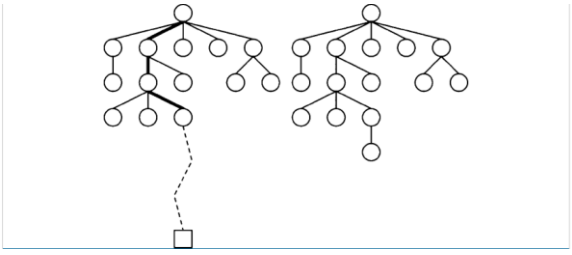

Fig. 1. Monte Carlo tree

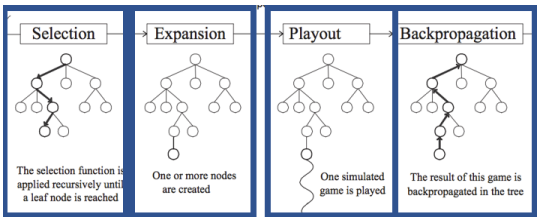

Fig. 2. Monte Carlo tree search in four steps

IV. Method

A. Minimax

Minimax is a method try to minimize the opponent's maximum reward in each circumstance and is the standard search approach for two-player combinatorial games. The search is typically stopped prematurely, a value function is used to estimate the outcome of the game, and the α–β heuristic is typically used to prune the tree. The max algorithm is the analogue of minimax for non-zero-sum games and/or games with more than two players [5].

B. Pruning

- Soft pruning of moves that may later be searched and selected.
- Hard pruning of moves that will never be searched or selected.

The pruning of suboptimal moves from the search tree is a powerful technique when used with minimax, for example, the α–β algorithm yields significant benefits for two-player zero-sum games. Move pruning can be similarly beneficial for MCTS approaches, as eliminating poor choices allows the search to focus more on better choices [1][5].

Progressive unpruning/widening is an example of a heuristic soft pruning technique. The advantage of this idea over hard pruning is that it manipulates heuristic knowledge to immediately reduce the tree's size, but all moves will eventually be considered (given enough time) [1].

C. Implement A

Formula A*B^(k-kinit) to make a threshold. When this threshold is reached, k will +1 and increase the size of the threshold.

D. heuristicScores

A heuristic score for recording the state. After the threshold is reached, it will be sorted from low to high, and the last k number will be used as a benchmark. Any node with a score lower than the benchmark will not be explored.

E. Iteration

Used to record the number of explored nodes, unpruning will be triggered when it is equal to the threshold.

Modify the TreePolicy() in BasicTreeNode.js to achieve the effect of pruning.

F. AMAF (All Moves As First)

AMAF is a widely used variation of the Monte Carlo Tree Search (MCTS) algorithm, originally introduced by Brügmann []to enhance MCTS's performance in playing the game of Go. The concept of basic AMAF involves extending the scope of eligibility for updates during the Back Propagation phase. While standard MCTS only updates nodes along the path that led to the chosen move, AMAF also updates the statistics of sibling nodes under a certain condition: the sibling's corresponding action must have been executed during the rollout.

This extension of updates to sibling nodes is intended to accelerate the learning process by rapidly incorporating additional information from rollouts. This infusion of knowledge is believed to be advantageous for refining the statistics of nodes. However, it's worth noting that there are potential downsides to this approach as pointed out by Helmbold and Parker-Wood. This expedited learning process might introduce irrelevant or biased information into the node statistics, which could potentially affect the quality of decisions made during the search process.

In summary, AMAF enhances MCTS by allowing updates to the statistics of sibling nodes based on actions performed in rollouts. While this can speed up learning, it might also introduce noise or bias to the node statistics. The trade-off between the benefits and potential drawbacks of AMAF should be considered when implementing or utilizing this technique.

G. Function

In the Pommerman Java framework's MCTS package, there are several classes that play a role in implementing the MCTS algorithm and its enhanced variant with AMAF:

1. **SingleTreeNode.java:** This class contains Java methods that collectively represent the MCTS algorithm. It likely handles tree expansion, selection, and simulation.

2. **MCTSPlayer.java:** This class serves as the interface between the game API and the MCTS algorithm. It likely handles interactions with the game environment and manages the MCTS search process.

3. **MCTSParams.java:** This class holds all the parameters related to the MCTS algorithm that will be tuned. It seems that you're introducing an additional parameter (denoted as α) to this class for your AMAF enhancement.

Since this isaiming to have an AMAF agent play against a standard MCTS agent, you're not modifying the existing MCTS agent's code. Instead, I creating duplicated classes for the AMAF-enhanced version:

4. **MCTSPlayer_2.java:** This duplicated class interfaces the game API with your AMAF-enhanced MCTS algorithm.

5. **SingleTreeNode_2.java:** This duplicated class likely contains modifications to the **uct()** and **backUp()** methods, which control the Tree Selection policy and Back Propagation in your AMAF-enhanced MCTS.

6. New instance variables are added to the **SingleTreeNode_2** class to store AMAF scores separately from regular scores, as this is essential for the AMAF enhancement.

V. Experiment and Results

In implementing the α-AMAF enhancement, I've tentatively set α = 0.4. For the RAVE enhancement, I've chosen to set N = 20 based on my observation that, when tracing the growth of the tree in the original MCTS, most depth-1 nodes are eventually visited around 30 times or so. For testing in Step 1, I substitute the benchmark agent (player name “MCTS”) with the basic AMAF, α-AMAF, and RAVE agents respectively (all represented by player name “MCTS_2”), while maintaining all other configurations unchanged. The figure below illustrates the two benchmark scores alongside the scores of the three new agents.

TABLE I. Win rate for full and partial observability.

Agents	V-MCTS	A-MCTS	MCTS	R-MCTS
FULL	55%	57.1%	59.5%	58.1%
PATIAL	54%	56.4%	52%	50%

TABLE II. Win rate for full and partial observability.

Agents	A-MCTS	MCTS	R-MCTS
FULL	31.5%	50.8%	35.3%
PATIAL	37.5%	36.8%	46%

Combining the outcomes depicted in the preceding charts, I can deduce that basic AMAF and RAVE don't exhibit strong performance compared to α-AMAF. In fact, they even face defeat from the benchmark agent under certain conditions. In contrast, α-AMAF demonstrates a consistent improvement in win rate by approximately 5 percentage points in scenarios involving full observability. Moreover, in cases of partial observability, it achieves results nearly on par with the benchmark agent.

VI. Conclusion and Future work

This paper firstly reviews the history of MCTS and introduces some essential functions and application scope of MCTs, as well as a review of some research in the field of AI. Followed by the Java framework of the multi-player game Pommerman. From this, we enhance the MCTS in the Java framework. First, using the a-b algorithm and the minimax method are used to strengthen the search tree following the logic of the Pommer game. Then some Pommerman play strategies are added to players' heuristics so that the MCTS algorithm can complete the search more efficiently to achieve the purpose of enhancing the winning rate.

Finally, the enhanced MCTS were investigated using the data visualization function of framwork allows us to test and investigate complete information games and simultaneous action selection, a familiar mechanic in games. The conclusion is that the MCTS algorithm can achieve high scores in the game Pommerman with good heuristics and post-enhanced search algorithms.

For future work, use other enhancement methods for MCTS, Monte Carlo Proof-Number Search (MC-PNS)/Score Bounded MCTS to study other card or board games.

References

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen,Stephen Tavener, Diego P´erez Li´ebana, Spyridon Samothrakis, and Simon Colton. A Survey of Monte CarloTree Search Methods. In IEEE Trans. on Computational Intelligence and AI in Games, volume 4, pages 1–43, 2014.

[2] S. Mozaffari, B. Azizian and M. H. Shadmehr, "Highly efficient alpha-beta pruning minimax based Loop Trax Solver on FPGA," 2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS), 2015, pp. 1-4, doi: 10.1109/CADS.2015.7377789.

[3] Helmbold, D. and Parker-Wood, A. All-Moves-As-First Heuristics in Monte-Carlo Go. AMAFpaperWithRef.pdf (ucsc.edu).

[4] Raluca D. Gaina, Sam Devlin, Simon M Lucas, and Diego Perez-Liebana. Rolling Horizon Evolutionary Algorithms for General Video Game Playing. IEEE Transactions on Games, 2021.