Artificial Intelligence in Games group Coursework

Amit Kotkar

Claudiu Andrei

Khalid Salman

Queen Mary University of London, UK

Table of Contents

Abstract: Real-time imperfect information games pose a significant challenge for Artificially intelligent Agents. A real-time game forces an agent to decide an action within a specified budged (100ms typically), while imperfect information games hide information from the agent which makes it more difficult for an agent to decide an action. This report discusses the development of an artificially intelligent agent that can play a real-time imperfect information game named “Bomber-Man”. The report also includes a literature review that explores similar research in this area. Our agent is a hybrid agent that uses rule-based actions and a modified MCTS algorithm to decide what is the best action to do next. The performance of our proposed agent is compared against previously implemented agents (vanilla MCTS, REHA, OSLA, and rule based) in different settings (full observability, partial observability, free-for-all, teams, and 1-on-1). Results show that our agent surpasses other agents in most experiments.

1 Introduction

There are numerous strategies that can be used to develop an agent for real-time games, including different tree search algorithms, evolutionary algorithms, and even simple rule-based algorithms.
The aim of this report is to detail our attempt at developing a “good” A.I. agent for a real time imperfect game using the “Pommerman” framework, a Java implementation of the game “Bomber-Man”, which relies heavily on strategy: mainly trapping opponents while avoiding enemy bombs. This game is particularly interesting as it does not take into account any performance metric, the agent’s only goal is being the last player alive.
This report outlines some of the previous research on the use of A.I. in games, focusing mainly on A.I. techniques for real-time games whilst also describing how the proposed agent performs against other algorithms in different settings. The results are discussed before giving a conclusion and reflection about future work.

2 Literature Review

Pommerman is a multi-agent game built to test autonomous artificial intelligence systems. In the field of A.I. in games this is a famous problem to tackle: numerous people have attempted to create an agent for this game in the past and many ways of playing have been explored. The MCTS and RHEA based agents are the starting point for the development of our custom agent, which ideally, would perform better than the previously mentioned approaches. Monte Carlo Tree Search (MCTS) is a popular Artificial Intelligence algorithm used in games. It is a search algorithm which finds the solution to the goal state: *I.e it searches through various solutions in the state space until a desired state is reached.* MCTS has traditionally been used in zero-sum turn-based games such as chess, checkers or go, where it achieves acceptable accuracy. [1] proposes several enhancements to MCTS, such as advanced heuristics and custom chips.

This strategy has proved very efficient as it can take advantage of a high time budget without the state changing and therefore perform a search which reaches the terminal state of the game at every turn, allowing to obtain the most accurate reward at all times. This, however, is not the case for multiplayer real-time Strategy (RTS) games such as Pac-Man or Pommerman. For this genre of games, vanilla MCTS does not work effectively as the state changes continuously due to the tick-rate: frequency at which the game updates. This results in the search tree becoming obsolete and thus producing invalid results for the current game tick. MCTS needs to be adapted to work in RTS games: the problem can be overcome by limiting the available time for MCTS to perform the search and select the best action before the time limit for the current tick runs out. For example, [2] has used a 40ms time limit for the game of Pacman: the agent has to strictly make a decision withing that time frame. Once 40ms have elapsed, the game state changes and thus the search tree is obsolete. Having a strict time constraint for decision making requires several enhancements in the MCTS algorithm for it to perform optimally. [2] uses enhancements of variable depth search tree, simulations strategies, delayed rewards and reusing the search tree to get good results for the game of Pac-Man. When simulating the game, MCTS assumes random actions performed by the opponents and builds the search tree based upon that, however this may not be ideal as the opponent plays to maximize its reward. Developing an opponent model to predict the actions of adversaries can greatly enhance the results of the A.I. agent as MCTS is able to perform the search in the right branch of the tree. [3] proposes a hierarchical opponent model for the game of SPRING. Other opponent modeling strategies exist: MiniMax is one of the most frequently used, however it cannot be applied to real-time games efficiently therefore it won't be discussed.

3 Pommerman Framework

The java Pommerman framework [4] is an implementation for the game Bommerman. The rules are simple: agents can move in four directions and lay bombs, which explode after a pre-determined number of in-game ticks, eliminating players caught in the blast. The goal is to be the last player standing. Despite this simplicity, the game offers a plethora of challenges that must be taken in consideration when developing a new Agent. Firstly, as it is a real time game, the agent must take an action within a predefined budget (in this case 100 milliseconds). The A.I. cannot evaluate too many objects on the board, as it may take too long to compute a new action and therefore timeout without executing a move. Secondly, the game can be played in an imperfect information mode, which limit the range of the player's vision. This increases the complexity of the problem greatly as moves have to be calculated with less inspectable data and higher uncertainty.

4 Background

This section will explain the background behind the main techniques used in this report starting with vanilla MCTS in the first subsection (4.1). The two subsequent subsections (4.2 and 4.3) will discuss the background behind some modifications to the vanilla MCTS.

4.1 Monte Carlo Tree Search (MCTS)

MCTS is a tree search algorithm based on random sampling of the state space in any specific domain. The algorithm can be employed as a planning technique for AI agents. In fact, it has been successfully used in this manner in turn-based games such as go [5]. The algorithm was also reliably used in real time games such as MS Pac Man [6] and traveling salesman [7].

In MCTS the root of the tree represents the agent's current state, while the children represent the state-space of actions that the agent can take from that state. In the vanilla version of MCTS, four steps are performed iteratively (see figure 4.1) until a budget is reached:[8]:

- **Selection.** A tree policy is used to select a node for expansion. From the root, search for a leaf allowing for maximization of the tree policy.
- **Expansion.** The selected node is expanded to reach all available moves.
- **Roll-Out.** From the expanded node, actions are performed at random until the terminal state is reached.
- **Backpropagating.** The result of the rollout is returned back from the expanded node to the root of the tree.

During these four steps, statistics are stores in the nodes and visit counts are updated continuously. These information is used by the tree policy and recommendation policy *(section 4.1.1 and 4.1.2 respectively).*

Because statistics of the tree are always *available (results are immediately backpropagated),* MCTS can be terminated at any state while expecting to return a reasonable answer (generally the more the time allowed for planning, the better the action performed).

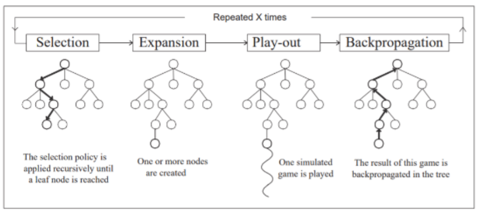

Repeated X times
Selection
Expansion
Play-out
Backpropagation
The selection policy is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 4.1 steps for MCTS [8]

4.1.1 Tree Policy

The tree policy is the policy by which the algorithm selects the next node for expansion given the current statistics of the node.

The most commonly used tree policy is the UCB1 policy (eq 4.1)

$$a = argmax_{a \in A(s)} \left(Q(s,a) + C \sqrt{\frac{lnN(s)}{N(s,a)}} \right)$$

Equation 4.1 The UCB1 tree policy equation.

Where:

- Q(s,a) accumulated reward on the node that represents taking action a from state s.
- N(s) number of visits to state s.
- N(s,a) number of times action a was chosen from state s.

4.1.2 Recommendation policy

The recommendation policy is invoked after planning is complete, allowing to determine what action to actually take in the game.

There are two ways to perform the recommendation policy.

1. Choose the most frequently selected action (with highest N(s,a) when s is the root node).
2. Choose the action with the highest reward (with highest Q(s,a) when s is the root node).

If enough iterations are made through the tree, the tree will converge to being optimal and the action that was chosen most often will eventually be the action with the highest reward.

4.2 Progressive Bias

Progressive bias is a technique that modifies the selection policy using domain specific knowledge. A game dependent heuristic is incorporated into the selection policy. The selection policy then becomes:

$$a = argmax_{a \in A(s)} \left(Q(s,a) + C \sqrt{\frac{lnN(s)}{N(s,a)}} + \frac{h(s,a)}{1 + N(s,a)} \right)$$

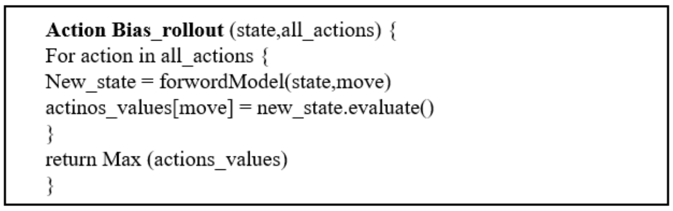
Equation 4.2 UCB1 tree policy with progressive Bias equation.

As the number of visits a node receives increases, the progressive bias term has less contribution to the equation, leading to convergence to the normal UCB1, hence the name “progressive”.

This technique is particularly useful when a node hasn’t been visited enough.

4.3 Biasing rollouts

Instead of picking actions uniformly at random in the rollout step, the rollouts can be biased according to some heuristic. Pseudo-code 4.1.



```
Action Bias_rollout (state,all_actions) {
    For action in all_actions {
        New_state = forwordModel(state,move)
        actinos_values[move] = new_state.evaluate()
    }
    return Max (actions_values)
}
```

Psudocode 4.1 Rollout Bias psudocode

5 Techniques Implemented

This section will describe our agent and the techniques that it uses. The first subsection gives an overall view to the agent discussing each component generally. Subsequent subsections will expand on the small components.

5.1 Overall description

Our agent has four different game modes:

1. Power up collection mode (rule based).
2. Pure safety mode (modified MCTS).
3. Dealing with traps mode (rule based).
4. General mode (modified MCTS).

at each new state, the agent determine which mode is most suitable to follow using a “safetyThreshold” parameter, “CollectMorePowerUps” and “isTrapped” Boolean values.

```
Mode identifyobjective (Game_state , safetyThreshold ) {
closestThreat= Game_state.getclosestThreat () // a threat is either a bomb or a flame
is_safe = evaluateSafety(safetyThreshold,closestThreat)
if(is_trapped) return DealingWithTrapMode
if(is_safe && collectMorePowerUps) return PowerUpCollectionMode
if(! is_safe && collectMorePowerUps) return PureSafetyMode
else return GeneralMode
}
```

Psudocode 5.1 How our agent Identifies its objective for the current game state

5.2 Power up collection mode

In this mode the agent has only one objective, which is power up collection. The agent scans the whole board for power ups and uses BFS to find a path to the closest power up. If none is found in the board the agent scans for the closest wood and destroys it. This mode is rule based.

5.3 Pure safety mode

In this mode the agent uses a modified version of the MCTS algorithm. It uses three different heuristics:

- **State evaluation heuristic:** this is the heuristic used at the end of each roll out to determine how good the state is. we used a very simple heuristic that returns 1 if the agent wins, -1 if it loses and zero otherwise. This simplicity has a key advantage, it makes the agent only care about its survival (it doesn't care about collecting powerups nor destroying woods ...etc.). Only caring about safety will prove very useful in the experiment section.
- **Rollout bias heuristic:** this is the heuristic used to bias the action selection during the rollout step. We used the custom heuristic from the original framework [4] for this purpose. Even though this is a simple heuristic, using it for rollout bias proved to be very effective compared to random rollout as we will discuss in the next section.
- **Progressive bias heuristic:** this is the heuristic used to bias the selection step with game dependent knowledge. This heuristic gives higher value for actions that take the agent away from bombs and flames.

our agent only biases the first half of the rollouts, the second half is randomized (next sections will prove the effectiveness of this approach).

5.4 Dealing with traps mode

When an agent is trapped (can’t move to any of the four directions), the rational thing to do, is to stay still and not place a bomb because placing a bomb will immediately result in the agents suicidal. Even though MCTS achieves the lowest suicide rate compared to other algorithms as mentioned in [9], the algorithm still commits suicide considerable number of times when it’s trapped, and this is mainly due to the delay of the bomb’s explosion (there is no immediate negative reward when placing a bomb while being trapped because bombs take time to explode).
Here comes the importance of this mode. we check the four adjacent nodes to the agent to determine if it’s trapped or not. If it’s trapped, we return a stay-still action to avoid committing suicide.

```
Bool isTrapped (game_state) {

myPosition = game_state.getPosition()
right,left,up,down = myposition.getAllFourDirections()
trap = [rigidTile, Flame]
if(right in trap && left in trap && up in trap && down in trap) return true
else return false
}
```

Psudocode 5.2 Trap awareness psudocode

5.5 General mode

This mode uses MCTS with rollout biasing. The rollout bias heuristic is the same as the one described in 5.3. However, the state evaluation heuristic is the advanced heuristic implemented in [4].

6 Experimental Study

This section will discuss the different experiments performed. The first subsection will briefly describe the different experimental setups used. Subsequent subsections will show the results of these setups.

6.1 Experimental setup

The experiments were divided to three major settings:

- **Free for all (FFA).**
- **Teams.**

- **1-on-1:** because the framework doesn’t allow less than four players, in this setting we used two suicidal players. These players kill them self immediately as the game begins. We also made sure that the agents playing in this mode are spawned in opposite directions (if agent1 is spawned in the top left corner, agent2 will be int the bottom right corner)

For those settings, number of experiments where conducted:

- **Full observability:** the agent can see the whole board (vision range = ∞)
- **Partial observability:** the agent can’t see the whole word (vision range = 3,4,5)

Table 6.1 and 6.2 summarizes the experiment setup.

Table 6.1 Experimental setup for 1-on-1 and team modes. ALL means we tried vision range (VR) configuration of 3,4,5.

Teams and 1-on-1 settings		
Game Mode	VR	Agents
1-on-1	∞	1.Our agent VS MCTS 2.Our agent VS REHA 3.our agent VS Simple Player 4.MCTS VS Simple Player 5.MCTS VS REHA
1-on-1	ALL	1.Our agent VS MCTS
Teams	∞	1.Our agent *2 VS MCTS*2 2.Our agent*2 VS REHA*2
Teams	ALL	1.Our agent *2 VS MCTS*2 2.Our agent*2 VS REHA*2

Table 6.2 Experimental setup for FFA mode. ALL means we tried Vision Range setup of 3,4,5.

FFA setting	
VR	Agents
∞	1.our Agent VS MCTS*3 2.our Agent VS MCTS VS REHA VS Simple Player 3.Our Agent vs MCTS VS REHA VS OSLA
ALL	1.our Agent VS MCTS VS REHA VS Simple Player

6.2 FFA Mode

To analyze this mode, we look at table 6.4. Starting at the third row, we see that the version of our agent that biased all the rollouts, lost to MCTS. Even though biasing the rollouts could prove very effective, taking too much time in this process will have a negative result.
However, the final version of our agent (biases only the first half of the rollouts) was able to win in all other settings.
We should also emphasis that our agent was able to achieve a more comfortable win in partial observability settings. This is because in limited vision settings, it is more likely to get trapped, and our agent has a mode that is specifically designed to protect it in these situations.
It is also important to note that our agent had the most ties percentage in most of the experiments. This means that the agent is consistently able to reach final stages of the game. The pure safety heuristic takes most of the credit for this result. Because whenever the agent is in danger, it's only goal becomes to stay safe ignoring other factors such as collect powerups and kill other enemies.
This opens the possibility to enhance the agent with an end game mode to convert the tie percentage into win percentages.

6.3 Teams Mode

The result of this mode is depicted in the second half of table 6.3. Our agent was able to win comfortably in all experiments except in VR=5. The reason for this remarkable result is mainly because there are two agents that start the game with the sole objective of collecting power ups. This means that when we reach the end of the game, our 2 agents on the board will have most of the power ups making them arguably stronger and more likely to win. Therefore, out of all of the experiments done, the highest winning rate was that of our agent in teams mode.

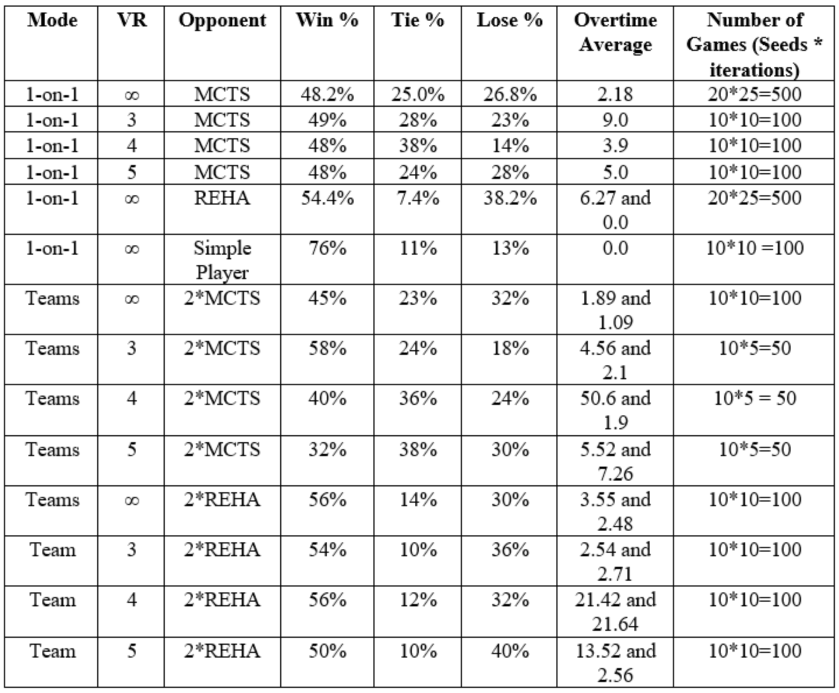
It is also interesting to note that for an MCTS opponent the highest winning rate was in the lowest tested vision range (VR =3). However, against REHA the highest winning rate was in full observability mode.

6.4 One-on-One mode

This mode's results are in the first half of table 6.3.
Here we notice a very interesting result. Table 6.5 shows that MCTS won 67% against REHA and 82% against the simple player. Even though our agent wins against MCTS in all settings with a notable difference (at least 20%), our agent wins less times against REHA and the simple player comparing to when MCTS faced those two opponents (54.4% and 76% for our agent VS 67% and 82%). This is particularly interesting because it proves that performing good against a strong opponent doesn't necessarily guarantee a better performance against weaker opponents.

It is also important to note that the wining rate against MCTS is approximately the same for all VR values (around 48%), However, our agent loses less in partial observability settings.

Table 6.3 Final version (The final version is the version described in section 5) of our agent against different opponents in different VR settings in teams and 1-on-1 modes. VR is the vision range.

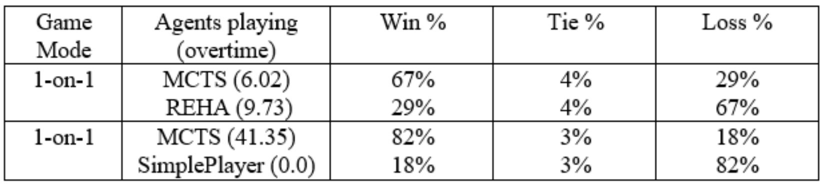


Mode	VR	Opponent	Win %	Tie %	Lose %	Overtime Average	Number of Games (Seeds * iterations)
1-on-1	∞	MCTS	48.2%	25.0%	26.8%	2.18	20*25=500
1-on-1	3	MCTS	49%	28%	23%	9.0	10*10=100
1-on-1	4	MCTS	48%	38%	14%	3.9	10*10=100
1-on-1	5	MCTS	48%	24%	28%	5.0	10*10=100
1-on-1	∞	REHA	54.4%	7.4%	38.2%	6.27 and 0.0	20*25=500
1-on-1	∞	Simple Player	76%	11%	13%	0.0	10*10 =100
Teams	∞	2*MCTS	45%	23%	32%	1.89 and 1.09	10*10=100
Teams	3	2*MCTS	58%	24%	18%	4.56 and 2.1	10*5=50
Teams	4	2*MCTS	40%	36%	24%	50.6 and 1.9	10*5 = 50
Teams	5	2*MCTS	32%	38%	30%	5.52 and 7.26	10*5=50
Teams	∞	2*REHA	56%	14%	30%	3.55 and 2.48	10*10=100
Team	3	2*REHA	54%	10%	36%	2.54 and 2.71	10*10=100
Team	4	2*REHA	56%	12%	32%	21.42 and 21.64	10*10=100
Team	5	2*REHA	50%	10%	40%	13.52 and 2.56	10*10=100

Table 6.4 Different versions of our agent against different opponents in FFA mode. (The final version is the version described in section 5). Number of games is 10 seeds * 10 iterations = 100.

Our Agent Version	**VR**	**Agents (overtime)**	**Win %**	**Tie %**	**Lose %**
Final version	∞	Our Agent (22.3)	30%	21%	49%
		MCTS (69.18)	23%	22%	55%
		REHA (85.56)	22%	9%	69%
		OSLA (0.0)	0%	1%	99%
Final version	∞	Our Agent (1.0)	44%	14%	42%
		MCTS (24.48)	32%	12%	56%
		REHA (36.24)	8%	4%	88%
		SimplePlayer (0.0)	0%	2%	98%
Using full roll out bias	∞	Our Agent (0.46)	25%	15%	60%
		MCTS (17.61)	32%	5%	63%
		REHA (25.0)	23%	13%	64%
		SimplePlayer (0.0)	3%	1%	96%
Final version	∞	Our Agent (0.81)	31%	32%	37%
		MCTS (29.13)	12%	21%	67%
		MCTS (44.68)	13%	25%	62%
		MCTS (26.41)	4%	21%	75%
Final version	3	Our Agent (36.74)	35%	22%	43%
		MCTS (20.61)	15%	18%	67%
		REHA (39.44)	22%	10%	68%
		SimplePlayer (0.0)	4%	0%	96%
Final version	4	Our Agent (20.06)	35%	29%	36%
		MCTS (5.48)	17%	21%	62%
		REHA (5.37)	14%	20%	66%
		SimplePlayer (0.0)	1%	3%	96%
Final version	5	Our Agent (2.71)	40%	20%	40%
		MCTS (23.31)	18%	15%	67%
		REHA (22.53)	17%	9%	74%
		SimplePlayer (0.0)	2%	2%	96%

Table 6.5 Experiments that don't involve our agent to demonstrate how previous algorithms performed.



Game Mode	Agents playing (overtime)	Win %	Tie %	Loss %
1-on-1	MCTS (6.02)	67%	4%	29%
	REHA (9.73)	29%	4%	67%
1-on-1	MCTS (41.35)	82%	3%	18%
	SimplePlayer (0.0)	18%	3%	82%

7 Discussion

Our agent's development rational was that if the agent is able to reach the end of the game, having the most power ups, it is more likely to win. However, collecting the most power ups is a risky process that may increase the chances of dying. Balancing between these tradeoffs was the main challenge.
The high percentage of wins and ties overall suggest that we were able to reach the final stages of the game without dying.
Also results show a very surprising fact, which is just because an agent wins against the strongest agent, this don't necessarily mean it will perform better than it against weaker agents.
Another important insight seen form experiments, spending long time in rollout biasing will have negative results on the agent's performance as it will reduce the number of times it explores other parts of the tree.
Finally, results shows that our agent generally achieves the highest winning rates in:

- Team mode: because the two agents consume almost all power ups in the board.
- partial observability settings: because our agent kills itself less when it's trapped.

8 Conclusions and Future Work

This report presents a comparative study between our developed agent and other previously implemented agents on the game of Pommerman. From this report, the first conclusion that can be made is that using different objectives for the agent depending on the current game state can result in a notable improvement in its performance. Moreover, the configuration used for the MCTS part of the agent had a significant impact in the overall performance as its responsible for harnessing having more power ups to win the game.

In future work, we aim to enhance the performance of our agent even further by incorporating opponent modeling and dealing with imperfect information setup. For opponent modeling, we aim to use offline training and collect data by playing many games between four MCTS agents, the data will comprise of the action performed (the target value) in a particular game state (the features). We then use this data to train a logistic regression classifier. The classifier will behave as the opponent model.

Regarding dealing with imperfect information, we aim to make the agent remember the tiles seen temporary (the remembering period will vary depending on the tile itself). If the tile is Rigid tile, memory = ∞. If the tile is bomb or wooden tile, memory = some parameter to tune.

In-order to perform the second purposed modification, we need to modify the Pommerman framework[4] to enable editing the game state as we will be constantly changing the board (as seen from our agent's perspective).

9 References

[1] M. Campbell, A. J. Hoane, and F. H. Hsu, "Deep Blue," *Artif. Intell.*, vol. 134, no. 1–2, pp. 57–83, 2002, doi: 10.1016/S0004-3702(01)00129-1.

[2] T. Pepels, M. H. M. Winands, and M. Lanctot, "Real-time monte carlo tree search in Ms Pac-Man," *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 3, pp. 245–257, 2014, doi: 10.1109/TCIAIG.2013.2291577.

[3] F. Schadd, S. Bakkes, and P. Spronck, "Opponent modeling in real-time strategy games," *8th Int. Conf. Intell. Games Simulation, GAME-ON 2007*, no.

June, pp. 61–68, 2007.

[4] "GAIGResearch/java-pommerman: Java version of Pommerman." [Online]. Available: https://github.com/GAIGResearch/java-pommerman. [Accessed: 05-Nov-2021].

[5] A. Rimmel, O. Teytaud, C. S. Lee, S. J. Yen, M. H. Wang, and S. R. Tsai, "Current frontiers in computer go," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 229–238, 2010, doi: 10.1109/TCIAIG.2010.2098876.

[6] T. Pepels, M. H. M. Winands, and M. Lanctot, "Real-time monte carlo tree search in Ms Pac-Man," *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 3, pp. 245–257, 2014, doi: 10.1109/TCIAIG.2013.2291577.

[7] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Monte Carlo Tree Search with macro-actions and heuristic route planning for the Multiobjective Physical Travelling Salesman Problem," *IEEE Conf. Comput. Intell. Games, CIG*, 2013, doi: 10.1109/CIG.2013.6633658.

[8] G. M. J.-B. CHASLOT, M. H. M. WINANDS, H. J. VAN DEN HERIK, J. W. H. M. UITERWIJK, and B. BOUZY, "Progressive Strategies for Monte-Carlo Tree Search," *New Math. Nat. Comput.*, vol. 04, no. 03, pp. 343–357, 2008, doi: 10.1142/s1793005708001094.

[9] D. Perez-Liebana, R. D. Gaina, O. Drageset, E. İlhan, M. Baila, and S. M. Lucas, "Analysis of statistical forward planning methods in pommerman," *Proc. 15th AAAI Conf. Artif. Intell. Interact. Digit. Entertain. AIIDE 2019*, pp. 66–72, 2019.