Analysis of Statistical Forward Planning Methods in Pommerman

**Diego Perez-Liebana, Raluca D. Gaina, Olve Drageset,
Ercüment İlhan, Martin Balla, Simon M. Lucas** [1]
[1] School of Electronic Engineering and Computer Science
Queen Mary University of London, UK

Abstract

Pommerman is a complex multi-player and partially observable game where agents try to be the last standing to win. This game poses very interesting challenges to AI, such as collaboration, learning and planning. In this paper, we compare two Statistical Forward Planning algorithms, Monte Carlo Tree Search (MCTS) and Rolling Horizon Evolutionary Algorithm (RHEA) in Pommerman. We provide insights on how the agents actually play the game, inspecting their behaviours to explain their performance. Results show that MCTS outperforms RHEA in several game settings, but leaving room for multiple avenues of future work: tuning these methods, improving opponent modelling, identifying trap moves and introducing of assumptions for partial observability settings.

1 Introduction

Statistical Forward Planning (SFP) methods (i.e Monte Carlo Tree Search - MCTS - and Rolling Horizon Evolutionary Algorithms - RHEA), are approaches that use a Forward Model to simulate possible future states when given state-action pairs. These methods have shown a high proficiency in single player games with full observability, as research on General Video Game AI (Perez et al. 2019) has shown recently. Multi-agent and partially observable games have gained special research interest, as they pose interesting challenges for all AI methods. Examples of this are the Marlö (Perez-Liebana et al. 2018) and Pommerman competitions. Pommerman is a game by Resnick et al. (Resnick et al. 2018a) that enhances the original Bomberman (Hudson Soft, 1983), which presents a battle scenario which demands competitive and cooperative skills in a multi-agent, partially observable environment. For an AI player, Pommerman provides an excellent benchmark for planning, learning, opponent modeling, communication and game theory.

This paper presents the first comparison between SFP methods in Pommerman. The main contribution of the paper is to show how the different methods perform in Pommerman *and* to present an extended analysis on the games played and how the agents behaved in them. We describe game, related work, agents and experiments, to finally propose next steps to be taken to improve the analysed methods.

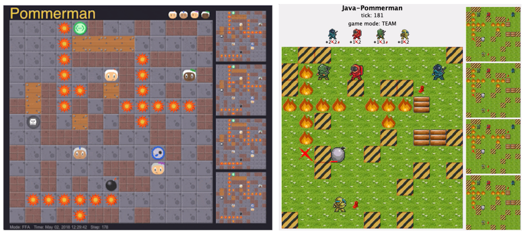

Figure 1: Original Pommerman (left) and Java version (right).

2 Pommerman

Game Rules: The game takes place in a randomly drawn 11×11 grid between 4 players. Players are placed in the 4 corners of the grid and the level is scattered with obstacles of two types: wooden and rigid boxes. The board is symmetric along the main diagonal. The game can be played with full or partial observability. The objective of the game is to be the last player alive at the end, or the last standing team. Players can place bombs that explode after 10 game ticks. These bombs, when exploded, generate flames in a cross-shape with size of (initially) 1 cell around the explosion point. These flames eliminate players, items and wooden boxes on collision, as well as explode other bombs. By default, players can't drop more than one bomb at a time. Bombs, as well as obstacles, can't be traversed. Wooden obstacles, when destroyed, may reveal pick-ups, initially hidden to the players. These items are *extra bombs* (adds 1 to the bombs ammo), *blast strength* (adds one to the explosion size) and *kick* (allows kicking a bomb in a straight line).

Game Modes: The game can be played in three different modes: free for all (FFA), team (TEAM) and team radio. The last mode allows agents in a team to define a communication protocol. In this work we concentrate on non-communication modes only. To make sure games are finite, a limit of 800 game ticks is set. In **FFA**, all players compete against each other, and the last player standing wins.

All players that die immediately lose the game. Players that are alive at the end tie, even if they are eliminated simultaneously at the same (and last) tick. In **TEAM** mode, players compete in pairs, with teammates initially placed in opposite corners of the board. A team loses when both agents on the team die, triggering victory for the other team. It's not necessary that both members of the team are alive at the end of the game for a team to win. If both teams have players alive at the end of the tick limit, both teams tie.

Board Generation: The play area is generated automatically using a random `seed`: first, agents are placed in the corners, allocating some free space around them. Wooden boxes are then placed to allow a passable passage between players, followed by uniform random and symmetric positioning of $X = 20$ rigid and $Y = 20$ wooden tiles. If the number of inaccessible tiles is higher than $Z = 4$, this process repeats. Finally, a number of $W = 10$ items are placed on randomly selected wooden boxes, types also chosen at random. The values of X, Y, Z and W are parameters of the level generator, with given values used in our experiments.

Framework: The experiments shown in this paper do not use the original Pommerman implementation[1], but an alternative one written in Java[2]. Figure 1 shows screenshots of both games. The present framework mimics exactly the original in terms of rules, observations, modes and level generation. Furthermore, it is possible to run agents written for each framework directly in the other one. The Java version, however, is optimized to run faster: while the original Python version runs at 5.3K ticks per second[3], this version runs at 241.4K ticks/s (above 45 times faster). This naturally allows quicker execution for the experiments presented in this paper. Another reason to implement our own framework is to easily incorporate event logging, used for a deeper analysis of the results, as shown in the Results section.

At each game tick, the framework provides each agent with information about the state of the game and the objects in the board. It also provides a Forward Model (FM), which can be used to roll the state forward when supplying a set of actions (one for each player). The agents must return, at each game tick, one of the available actions: `STOP` (does nothing), four directions (`UP`, `DOWN`, `LEFT`, `RIGHT`) and `BOMB` (that places a bomb, if possible, in the current position). For the partial observability settings, the observations received by the agents are limited to their Vision Range (VR). A VR of n tiles determines a square of vision centered on the agent, with n tiles between agent and observation edge. Everything outside the VR is *Fog* and ignored by the FM, initializing it as an empty tile when discovered during the simulations.

3 Related work

3.1 Statistical Forward Planning

Statistical Forward Planning (SFP) is a group of robust, general, stochastic AI techniques that operate without the need of training. SFPs rely on an FM of the game that can be rapidly copied and advanced forward, given actions the agents would play. Examples of SFPs are Monte Carlo Tree Search (MCTS), which uses randomized forward simulations to build a search tree, and Rolling Horizon Evolutionary Algorithms (RHEA), which use the same kind of randomized forward simulations to evolve a plan. Browne's MCTS survey (Browne et al. 2012) outlines game and non game uses of MCTS and its variants, including its ability to perform better than previously used methods in games like Go. Since then, MCTS has also been applied to Go with even greater success in combination with deep learning(Silver et al. 2016). SFPs have shown a high degree of adaptability to previously unseen problems, and have been popular with entries to General Video Game AI (GVGAI; (Perez et al. 2019)) competitions in both single and two-player games. RHEA was first used in real-time games to play the Physical Travelling Salesman Problem (Perez-Liebana et al. 2013), and its tuning and performance was deeply analysed in multiple GVGAI games. RHEA is highly parameterisable and it has been shown to outperform MCTS in many games (Gaina et al. 2017), (Gaina, Lucas, and Perez-Liebana 2017a), (Gaina, Lucas, and Perez-Liebana 2017b), (Gaina, Lucas, and Perez-Liebana 2019).

3.2 Pommerman

The first Pommerman competition was organized in 2018 and focused on the FFA game mode This competition saw a Finite State Machine Tree Search approach come in first, with a rule-based AI in second (Resnick et al. 2018a). The second competition organized received more entries, including both planning and learning approaches. Organized as part of NeurIPS 2018, this competition focused on the TEAM mode (Resnick et al. 2018a). In (Osogami and Takahashi 2019), the authors describe their 2 MCTS-based agents which ranked first and third, and compare them against the 2^{nd} place agent (another MCTS implementation with a depth $D = 2$). Osogami fixes random seeds on the first level of the tree and perform deterministic rollouts with $D = 10$.

Zhou et al. (Zhou et al. 2018) compared different search techniques, such as MCTS, BFS and Flat Monte Carlo search in this game. The authors show, in the full observable mode, that MCTS is able to beat simpler and hand-crafted solutions. The results reported in our paper align with these findings, but we expand the work to partially observable settings, add RHEA to the pool of agents tested, and report a more detailed analysis on how the agents played the games.

An important body of literature on Pommerman is on the challenge of learning to play offline. Peng et al. (Peng et al. 2018) used continual learning to train a population of advantage-actor-critic (A2C) agents in Pommerman, beating all other learning agents in the 2018 Competition. A Deep Neural Network (DNN) is updated using A2C in a process that allows the agent to progressively learn new skills, such as picking items and hiding from bomb explosions. Another Deep Learning approach is proposed by (Malysheva et al. 2018), which uses Relevance Graphs obtained by a self-attention mechanism. This agent, enhanced with a message generation system, analyzes the relevance of other

[1] https://www.pommerman.com/

[2] https://github.com/GAIGResearch/java-pommerman

[3] Windows 10 Intel Core i7 16GB RAM

agents and items observed in the environment. The authors show that their Multi-Agent network outperforms all other tested agents. Resnick et al. (Resnick et al. 2018b) proposed *Backplay*, which speeds up training by starting on the terminal states of an episode. By backtracking towards the initial state, the agent improves on sample-efficiency and can learn faster using curriculum learning. (Gao et al. 2019) implemented Skynet, second-best learning agent in the NeurIPS 2018 Team Competition. Each agent of this team is a neural network trained with Proximal Policy Optimization, reward shaping, curriculum learning and action pruning.

Finally, some relevant work on hybrid methods combines Deep Learning with MCTS. In (Kartal, Hernandez-Leal, and Taylor 2018), also later expanded in (Kartal et al. 2019), the authors train a DNN using Asynchronous Advantage Actor-Critic (A3C) enhanced with temporal distance to goal states. They also integrate MCTS as a demonstrator for A3C, which helps reduce agent suicides during training via Imitation Learning. It is interesting to observe here that these findings align with the lower suicide rate shown by MCTS in the experiments performed for this paper (see Results section).

4 Agents

This section describes the 4 agents used in the experiments and the heuristic employed to evaluate states, in this order.

One Step Look Ahead (OSLA): This is a simple agent that uses the FM to roll the current state with each of the 6 actions available. Each future state is evaluated with a custom heuristic (see the end of this section), which provides a numerical score for it. The action that led to the highest score is applied in the game, ties broken uniformly at random.

RuleBased: This agent is a rule based system that mimics the Simple Agent of the original Pommerman implementation[4]. At each state, the agent uses Dijkstra's algorithm, with a maximum depth of 10, to compute distances to the different game objects in the current game state. Then, it executes the following logic: First, if there are upcoming bomb explosions, it tries to escape. Otherwise, if the agent is adjacent to an enemy, it should lay a bomb. If there is an enemy within 3 steps or a power-up within 2, the agent moves towards it. Alternatively, if the agent is adjacent to a wooden box, it will lay a bomb to try to destroy it and open more open space. Finally, if none of the rules before have triggered, the agent moves randomly to a not recently visited position.

Rolling Horizon Evolutionary Algorithms (RHEA): RHEA is a family of algorithms that use evolution in real-time to recommend an action on each turn for the player to make. In its standard form (Perez-Liebana et al. 2013), used in this paper, RHEA evolves sequences of L actions. Each sequence fitness is calculated by first using the FM to roll the state L steps ahead, and then evaluate the state reached at the end of the sequence of actions. In our experiments, this evaluation is done with the custom state heuristic described at the end of this section. RHEA uses a population of N individuals and regular evolutionary operators (i.e. mutation, crossover and elitism) apply. At the end of the thinking budget, RHEA returns the first action of the best sequence found (the one with the best fitness) to be played in the game. One of the main improvements in RHEA, also used in these experiments, is the *shift buffer* (Gaina, Lucas, and Perez-Liebana 2017b). Once the best individual is selected at one game tick, the first action is executed in the game and removed from the individual. Then, the rest of the action sequence is *shifted*, so the next individual starts with the second action of the best sequence in the previous step. A new action is created, uniformly at random, for the end of the sequence, keeping the individual length at L. This mechanism retains good solutions from previous game ticks. Each solution shifted is reevaluated in the following game ticks to account for inaccuracies in the opponent model or FM.

Monte Carlo Tree Search (MCTS): MCTS (Coulom 2006) is a highly selective best-first tree search method. This iterative algorithm balances between exploitation and exploration of the best moves found so far and those requiring further investigation, respectively. On each iteration of the algorithm, the standard MCTS performs 4 steps: 1) *selection* uses a `tree policy` to navigate the tree until finding a node that does not have all its children expanded; 2) *expansion* adds a new node to the tree; 3) *simulation* performs moves according to a `default policy` until a terminal node or a depth D is reached. The game state reached at the end of this *rollout* is evaluated using a heuristic (see next section); and 4) *backpropagation* updates the visit counts of all traversed nodes in this iteration, as well as the final rewards obtained. At the end of the allowed budget, MCTS returns the action taken most times from the root. Count visits and average rewards are used to inform the tree policy, which normally takes the form of Upper Confidence Bounds for trees (Kocsis and Szepesvári 2006). This policy has a parameter K that balances exploration and exploitation during the *selection* step. The default policy used in this paper picks actions uniformly at random for the *simulation* step.

Custom State Heuristic: OSLA, MCTS and RHEA use the same heuristic to evaluate a state. It is calculated as the difference between the root and the evaluated game states, based on a series of features: number of alive teammates (Δ_t), number of alive enemies (Δ_e), number of existing wooden blocks (Δ_w), blast strength (Δ_b) and the ability of kicking bombs ($\Delta_k \in \{0, 1\}$). The state value is the weighted sum $\sum_i \Delta_i \times w_i$, where $w_t = 0.1$ (0 for FFA), $w_e = 0.13$ (0.17 for FFA), $w_w = 0.1$, $w_b = 0.15$ and $w_k = 0.15$. The weights were manually picked by observing agent performance prior to the experiments.

5 Experimental Setup

We define a level as a game with a fixed board. We generated 20 fixed levels with 20 different random seeds, sampled uniformly at random within the range [0, 100000]. All experiments described in the rest of this paper play each of these 20 levels 10 times, hence 200 plays per configuration. We test 2 different game modes, Free For All (FFA) and TEAM, in 4 different observability settings: vision ranges VR $\in \{1, 2, 4\}$, or fully observable (denoted in this paper as ∞).

[4] https://github.com/MultiAgentLearning/playground/blob/master/pommerman/agents/simple_agent.py

Table 1: Experimental Setup. *All* $\equiv$ VR $\in \{1, 2, 4, \infty\}$. Each set up is repeated 200 times (10×20 fixed levels). There are 32 different configurations, totalling 6400 games played.

Game mode: FFA	
VR	**Agents**
∞	RHEA vs OSLA vs OSLA vs OSLA
	RHEA vs RuleBased vs RuleBased vs RuleBased
	MCTS vs OSLA vs OSLA vs OSLA
	MCTS vs RuleBased vs RuleBased vs RuleBased
All	OSLA vs RuleBased vs RHEA vs MCTS
	RHEA vs MCTS vs RHEA vs MCTS

Game mode: TEAM	
VR	**Agents**
All	RHEA $\times$ 2 vs OSLA $\times$ 2
	RHEA $\times$ 2 vs RuleBased $\times$ 2
	MCTS $\times$ 2 vs OSLA $\times$ 2
	MCTS $\times$ 2 vs RuleBased $\times$ 2
	RHEA $\times$ 2 vs MCTS $\times$ 2

OSLA, RuleBased, RHEA and MCTS were used in the tests. No communication is allowed between agents. For RHEA and MCTS, their look-aheads (L and D) are set to 12 moves. RHEA and MCTS use a budget of 200 iterations per game tick to compute actions, with a uniform random opponent model and the same custom state heuristic (described in the previous section) for evaluating states found at the end of action sequences. MCTS uses $K = \sqrt{2}$ and RHEA evolves a single individual ($N = 1$). New individuals are created every iteration via mutation (rate 0.5), keeping the best individual and a shift buffer is used.

Given that Pommerman is a 4-player game, the number of combinations of agents and modes is prohibitively high, thus we made a selection of the most interesting settings for our tests. Table 1 shows all configurations tested. For each game mode, first we aim to confirm our initial hypothesis that RHEA and MCTS have a higher performance than the other two simpler methods. Afterwards, we try to determine which of these two algorithms achieves better results in direct confrontation. In order to account for possible biases due to symmetry along the main diagonal, we tested RHEA versus MCTS (VR $= \{1, 2, 4, \infty\}$) with swapped positions. These tests showed that there is no relevant difference on the performance of the teams after the position exchange, thus swapped experiments were excluded. All experiments were run on IBM System X iDataPlex dx360 M3 server nodes, each with an Intel Xeon E5645 processor and a maximum of 2GB of RAM of JVM Heap Memory.

6 Results

Overall, the results indicate that RHEA and MCTS are both stronger than the simpler methods tested, OSLA and Rule-Based. This section presents and discusses results in the two game modes tested, FFA and TEAM[5].

6.1 FFA

In FFA games with full observability (first section of Table 2), it is interesting to observe that, while the difference in win rate against the RuleBased AI is quite small (rows 1 and 3) at 46.5% for MCTS and 33.0% for RHEA, MCTS tends to end more of its non-winning games in ties rather than losses, as opposed to RHEA. This suggests RHEA to adopt more aggressive or risky strategies, whereas MCTS plays safer and often avoids dying until the end of the game. This is corroborated by the average number of bombs these agents lay (RHEA uses about 30 more per game, see Fig. 2).

It is also worth taking into account the fact that the number of deaths by suicide in RHEA is higher than MCTS, as shown in Figure 2 (left). We refer to suicides to those cases in which an agent is killed by its own bomb. Note that in some cases this also includes chain reactions from other bombs. Suicides in Pommerman are seen as one of its most challenging aspects (Kartal et al. 2019). In general, MCTS achieves the lowest suicide rate in all VR settings and modes, which helps explain the success of this method.

The win rate percentage for both RHEA and MCTS increase when facing OSLA, with very few ties in both cases. The low number of ties compared to deaths in non-winning games suggests more aggressiveness, although OSLA is hindered by its short look-ahead in trying to avoid bomb explosions and suicides, as its horizon does not reach beyond the 10 game ticks bombs take to explode. MCTS clearly dominates OSLA with a 91.5% win rate. When all 4 methods play against each other (second section of Table 2), win rate heavily shifts in favour of MCTS with very short VR (1). In all VR options, OSLA and RuleBased keep a win rate no higher than 5%. RHEA's performance is similar for all VR values, 10% and 21% of winning. It's interesting to observe that the number of losses is higher with low visibility (74.50%, $VR = 1$) than in the other cases, where ties happen more often.

Finally, MCTS achieves a higher win rate when playing only RHEA in the shorter VR option (1; see third section of Table 2), but very similar to RHEA in the other vision ranges. RHEA, with VR$= 1$, loses 85.5% of games and ties rarely, a trend that changes for the other values of VR where ties become more frequent than losses.

6.2 TEAM

In team games, we can observe a similar performance of the algorithms with interesting differences. When playing against OSLA (top of Table 3), both RHEA and MCTS achieve high victory rates and low ties, as seen in FFA, with MCTS close to 100% win rate in all VR options. RHEA appears to perform better with high VR (and still wins fewer games than MCTS).

The average MCTS win rate when playing against the RuleBased AI is higher than previously observed in FFA games, around 70% vs 46.5%. This is probably due to having two strong agents on the same team. The performance of MCTS is kept fairly consistent regardless of VR (second section in Table 3). When MCTS plays RHEA, the former algorithm dominates when VR$= 1$, but they achieve a simi-

[5] Data and plots are available here: https://github.com/GAIGResearch/java-pommerman/tree/master/data/

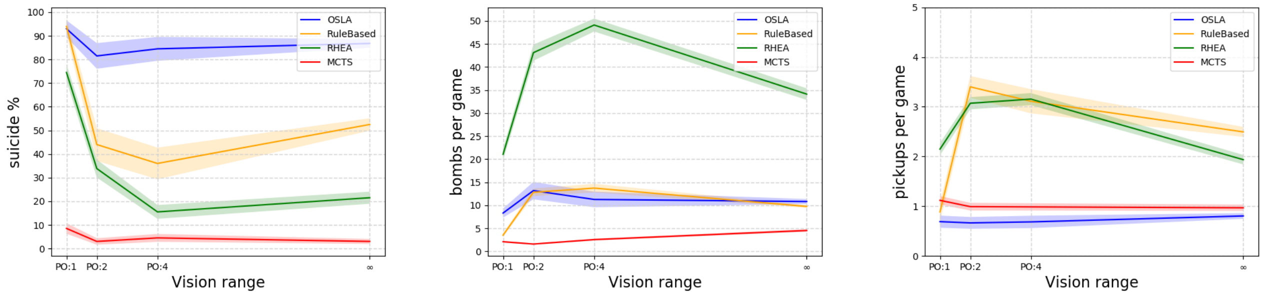

Figure 2: Events recorded during the FFA games played for this paper (results for TEAM are very similar). Left: percentage of deaths caused by the own agent bombs. Center: number of bombs placed per game. Right how many pick-ups are collected by the different agents. All charts show values for VR $= \{1, 2, 4, \infty\}$. Shaded area shows the standard error of the measure.

Table 2: FFA win rate (W), ties (T) and losses (L). 1^{st} column indicates vision range $\in \{1, 2, 4, \infty\}$. Names in italics represent results averaged across players of the same type.

VR	Agents	% Wins	% Ties	% Losses
∞	MCTS *RuleBased*	46.50 (4.0) 3.00 (1.0)	42.00 (3.0) 16.50 (3.0)	11.50 (2.0) 80.50 (3.0)
∞	MCTS *OSLA*	91.50 (2.0) 1.00 (0.3)	5.00 (2.0) 2.00 (1.0)	3.50 (1.0) 97.00 (1.0)
∞	RHEA *RuleBased*	33.00 (3.0) 12.50 (2.3)	22.00 (3.0) 12.67 (2.3)	45.00 (4.0) 74.83 (3.0)
∞	RHEA *OSLA*	65.50 (3.0) 11.17 (2.3)	1.00 (1.0) 0.33 (0.0)	33.50 (3.0) 88.50 (2.3)
1	RHEA OSLA MCTS RuleBased	20.50 (3.0) 2.50 (1.0) 67.50 (3.0) 1.50 (1.0)	5.00 (2.0) 0.50 (0.0) 7.50 (2.0) 3.00 (1.0)	74.50 (3.0) 97.00 (1.0) 25.00 (3.0) 95.50 (1.0)
2	RHEA OSLA MCTS RuleBased	21.00 (3.0) 0.00 (0.0) 18.00 (3.0) 4.50 (1.0)	43.00 (4.0) 3.50 (1.0) 54.00 (4.0) 23.00 (3.0)	36.00 (3.0) 96.50 (1.0) 28.00 (3.0) 72.50 (3.0)
4	RHEA OSLA MCTS RuleBased	19.00 (3.0) 0.00 (0.0) 16.50 (3.0) 3.00 (1.0)	57.00 (4.0) 2.00 (1.0) 59.00 (3.0) 17.00 (3.0)	24.00 (3.0) 98.00 (1.0) 24.50 (3.0) 80.00 (3.0)
∞	RHEA OSLA MCTS RuleBased	13.00 (2.0) 0.00 (0.0) 21.00 (3.0) 1.50 (1.0)	51.50 (4.0) 3.50 (1.0) 61.50 (3.0) 32.00 (3.0)	35.50 (3.0) 96.50 (1.0) 17.50 (3.0) 66.50 (3.0)
1	*RHEA* *MCTS*	8.50 (2.0) 19.50 (3.0)	6.00 (2.0) 40.50 (3.0)	85.50 (2.5) 40.00 (3.5)
2	*RHEA* *MCTS*	8.00 (2.0) 5.50 (1.5)	38.50 (3.5) 56.50 (3.5)	53.50 (3.5) 38.00 (3.0)
4	*RHEA* *MCTS*	1.25 (1.0) 2.00 (0.5)	67.00 (3.0) 74.25 (3.0)	31.75 (3.0) 23.75 (3.0)
∞	*RHEA* *MCTS*	1.75 (0.5) 1.00 (0.5)	73.25 (3.0) 83.75 (2.5)	25.00 (3.0) 15.25 (2.5)

Table 3: TEAM win rate (W), ties (T) and losses (L). 1^{st} column indicates vision range (VR). Results include 2 agents of the same type on a team and average across them. The row agent team plays against the column opponent team.

VR	Agents	% Wins	% Ties	% Losses
Opponent: OSLA				
1	*RHEA* *MCTS*	76.50 (3.0) 97.00 (1.0)	1.50 (1.0) 1.00 (1.0)	22.00 (3.0) 2.00 (1.0)
2	*RHEA* *MCTS*	88.50 (2.0) 95.00 (2.0)	3.50 (1.0) 2.00 (1.0)	8.00 (2.0) 3.00 (1.0)
4	*RHEA* *MCTS*	90.50 (2.0) 97.00 (1.0)	2.50 (1.0) 2.50 (1.0)	7.00 (2.0) 0.50 (0.0)
∞	*RHEA* *MCTS*	81.00 (3.0) 98.50 (1.0)	0.50 (0.0) 1.50 (1.0)	18.50 (3.0) 0.00 (0.0)
Opponent: RuleBased				
1	*RHEA* *MCTS*	76.50 (3.0) 68.50 (3.0)	3.00 (1.0) 19.50 (3.0)	20.50 (3.0) 12.00 (2.0)
2	*RHEA* *MCTS*	45.00 (4.0) 74.00 (3.0)	27.00 (3.0) 22.50 (3.0)	28.00 (3.0) 3.50 (1.0)
4	*RHEA* *MCTS*	55.00 (4.0) 70.00 (3.0)	22.50 (3.0) 28.00 (3.0)	22.50 (3.0) 2.00 (1.0)
∞	*RHEA* *MCTS*	40.50 (3.0) 73.00 (3.0)	23.50 (3.0) 23.00 (3.0)	36.00 (3.0) 4.00 (1.0)
Opponent: RHEA				
1	*MCTS*	68.50 (3.0)	14.00 (2.0)	17.50 (3.0)
2	*MCTS*	22.50 (3.0)	59.00 (3.0)	18.50 (3.0)
4	*MCTS*	7.50 (2.0)	85.50 (2.0)	7.00 (2.0)
∞	*MCTS*	9.00 (2.0)	84.00 (3.0)	7.00 (2.0)

lar performance with higher visibility, increasing the number of matches finished in ties.

This doesn't mean, however, that playing style does not vary when VR is changed. Figure 3 shows heatmaps of bomb locations by MCTS for all VR options. As can be observed, low observability leads to significantly less and very specific bomb placements. When VR $= 1$ there are less bombs placed than with higher VR values, but they are more localized around the starting position. In higher visibility, the

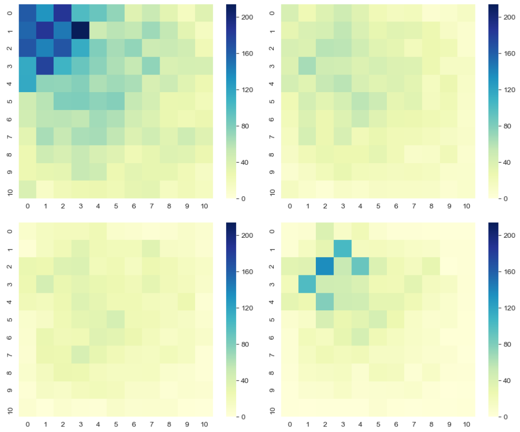

Figure 3: Bomb placement by MCTS in TEAM mode. Top left: $VR = \infty$; top right: $VR = 4$; bottom left: $VR = 2$; bottom right: $VR = 1$. Agent starts on top left corner.

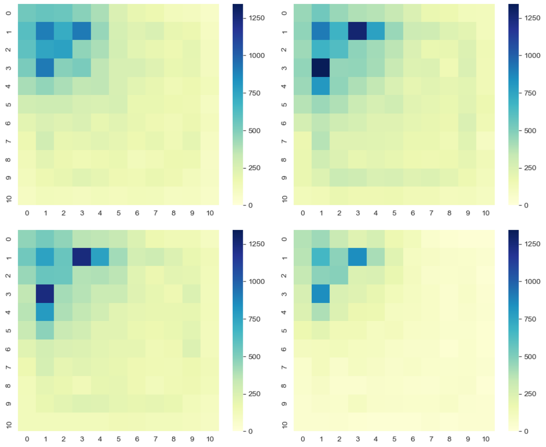

Figure 4: Bomb placement by RHEA in TEAM mode. Top left: $VR = \infty$; top right: $VR = 4$; bottom left: $VR = 2$; bottom right: $VR = 1$. Agent starts on top left corner.

bombs are more spread out around the level. However, making the game fully observable encourages the agent to explore more and scatter bombs across the entire map. It seems clear that the presence of PO hinders the capability of the agents to use many bombs.

RHEA's performance is again low with reduced vision range, but similar to MCTS with vision range 4 (the default option in the Pommerman competition). In terms of bomb placement, however, one can see a clear difference with MCTS. Figure 4 shows a difference between RHEA and MCTS, especially in intermediate VR values, showing a higher number of bombs being drop by the former (which agrees with the observations seen in Figure 2. In the case of RHEA, bombs are more concentrated around the starting position and around the edges of the board than MCTS, where we see a more even spread in the starting corner and towards the center of the map. This clearly shows that both SFP algorithms behave (i.e. explore the search space) differently. The large amount of bombs placed by RHEA may also explain why it tends to suicide more often.

7 Conclusions and Future Work

This paper presents a comparative study on the performance of SFP (MCTS and RHEA) agents on the game of Pommerman. The first conclusions that can be drawn is that SFP methods are stronger than the rule-based and one step look ahead agents they're compared against. Additionally, the configuration tested for MCTS seem the provide a better performance than that of RHEA. The analysis carried out on the different algorithms shows that more offensive strategies (like RHEA having a higher rate of bomb placing) are normally also riskier, due to the known challenge of suicides in this game. In fact, MCTS tends to lay less bombs than the other agents, but achieves a higher winning percentage in most modes and visibility settings. Partial observability increases the number of suicides for all agents, and full observability brings the performance of MCTS and RHEA quite closer. The PO setting also influences where and how often bombs are placed (as seen in the heatmaps presented), and shows differences of behaviour between the different SFP methods. Adding heuristics to remember where and when bombs were placed may lead to a performance boost. Another interesting observation is that many games end up in ties, especially when the visibility range of the agents is greatly limited. One possibility to alleviate this is to adopt the collapsing boards methodology followed in the Pommerman, by which winners are enforced.

RHEA is an algorithm with a large parameterization space. Further tests done in Pommerman with other parameters (N, L, mutation rate, among others) have shown different results for different game settings. A possibility of future work is to automatically tune RHEA parameters to boost the strength of this method, which in many other games has shown competitive performance with MCTS. A similar approach can be taken for MCTS (Sironi et al. 2018), although this algorithm has a smaller parameter space (comparing vanilla versions). Our initial tests on this matter suggest that this is indeed possible.

Other lines of future work, which tackle directly performance in Pommerman, are of a wider interest for the Game AI community. One of them is to learn an effective opponent model of the other agents, which improves upon the random modelling assumed in this paper. Simple statistical modelling based on the frequency of actions have provided promising results in the past for 2-player GVGAI (Gonzalez-Castro and Perez-Liebana 2017). Other interesting avenues for future work are to tackle partial observability by introducing assumptions of the unknown tiles of the FM (Osogami and Takahashi 2019), or learning value functions that identify trap states or moves (that can cause suicides in the game). Last but not least, it would also be interesting to compare these planning SFP approaches to learning agents submitted to the Pommerman competition, or even investigate more hybrid methods for this game.

References

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez-Liebana, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. on Computational Intelligence and AI in Games* 4(1):1–43.

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

Gaina, R. D.; Liu, J.; Lucas, S. M.; and Perez-Liebana, D. 2017. Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing. In *Springer Lecture Notes in Computer Science, Applications of Evolutionary Computation, EvoApplications*, 418–434.

Gaina, R. D.; Lucas, S. M.; and Perez-Liebana, D. 2017a. Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing. In *Proceedings of the Congress on Evolutionary Computation*, 1956–1963.

Gaina, R. D.; Lucas, S. M.; and Perez-Liebana, D. 2017b. Rolling Horizon Evolution Enhancements in General Video Game Playing. In *Proceedings of IEEE Conference on Computational Intelligence and Games*, 88–95.

Gaina, R. D.; Lucas, S. M.; and Perez-Liebana, D. 2019. Tackling Sparse Rewards in Real-Time Games with Statistical Forward Planning Methods. In *AAAI Conference on Artificial Intelligence (AAAI-19)*.

Gao, C.; Hernandez-Leal, P.; Kartal, B.; and Taylor, M. E. 2019. Skynet: A Top Deep RL Agent in the Inaugural Pommerman Team Competition. *arXiv preprint arXiv:1905.01360*.

Gonzalez-Castro, J. M., and Perez-Liebana, D. 2017. Opponent Models Comparison for 2 Players in GVGAI Competitions. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, 151–156. IEEE.

Kartal, B.; Hernandez-Leal, P.; Gao, C.; and Taylor, M. E. 2019. Safer Deep RL with Shallow MCTS: A Case Study in Pommerman. *arXiv preprint arXiv:1904.05759*.

Kartal, B.; Hernandez-Leal, P.; and Taylor, M. E. 2018. Using Monte Carlo Tree Search as a Demonstrator within Asynchronous Deep RL. *arXiv preprint arXiv:1812.00045*.

Kocsis, L., and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *European conference on machine learning*, 282–293. Springer.

Malysheva, A.; Sung, T. T.; Sohn, C.-B.; Kudenko, D.; and Shpilman, A. 2018. Deep Multi-Agent Reinforcement Learning with Relevance Graphs. *arXiv preprint arXiv:1811.12557*.

Osogami, T., and Takahashi, T. 2019. Real-time Tree Search with Pessimistic Scenarios. *arXiv preprint arXiv:1902.10870*.

Peng, P.; Pang, L.; Yuan, Y.; and Gao, C. 2018. Continual Match Based Training in Pommerman: Technical Report. *arXiv preprint arXiv:1812.07297*.

Perez, D.; Liu, J.; Khalifa, A. A. S.; Gaina, R. D.; Togelius, J.; and Lucas, S. M. 2019. General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms. *IEEE Transactions on Games*.

Perez-Liebana, D.; Samothrakis, S.; Lucas, S.; and Rohlfshagen, P. 2013. Rolling Horizon Evolution versus Tree Search for Navigation in Single-player Real-time Games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, 351–358. ACM.

Perez-Liebana, D.; Hofmann, K.; Mohanty, S. P.; Kuno, N.; Kramer, A.; Devlin, S.; Gaina, R. D.; and Ionita, D. 2018. The Multi-Agent Reinforcement Learning in MalmÖ Competition. In *Challenges in Machine Learning (CiML; NIPS Workshop)*, 1–4.

Resnick, C.; Eldridge, W.; Ha, D.; Britz, D.; Foerster, J.; Togelius, J.; Cho, K.; and Bruna, J. 2018a. Pommerman: A Multi-agent Playground. In *MARLO Workshop, AIIDE-WS Proceedings*, 1–6.

Resnick, C.; Raileanu, R.; Kapoor, S.; Peysakhovich, A.; Cho, K.; and Bruna, J. 2018b. Backplay:" Man muss immer umkehren". *arXiv preprint arXiv:1807.06919*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–489.

Sironi, C. F.; Liu, J.; Perez-Liebana, D.; Gaina, R. D.; Bravi, I.; Lucas, S. M.; and Winands, M. H. M. 2018. Self-adaptive MCTS for General Video Game Playing. In Sim, K., and Kaufmann, P., eds., *Applications of Evolutionary Computation*, 358–375. Springer International Publishing.

Zhou, H.; Gong, Y.; Mugrai, L.; Khalifa, A.; Nealen, A.; and Togelius, J. 2018. A Hybrid Search Agent in Pommerman. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 46. ACM.