A Hybrid Search Agent in Pommerman

Hongwei Zhou
New York University
Brooklyn, New York
hz1101@nyu.edu

Yichen Gong
New York University
Brooklyn, New York
yichen.gong@nyu.edu

Luvneesh Mugrai
New York University
Brooklyn, New York
lm3300@nyu.edu

Ahmed Khalifa
New York University
Brooklyn, New York
ahmed.khalifa@nyu.edu

Andy Nealen
New York University
Brooklyn, New York
andy@nealen.net

Julian Togelius
New York University
Brooklyn, New York
julian@togelius.com

ABSTRACT

In this paper, we explore the possibility of search-based agents in games with resource-intensive forward models. We implemented a player agent in the Pommerman framework and put it against the baseline agent to measure its performance. We implemented a heuristic agent and improved it by enabling depth-limited tree search in specific gameplay moments. We also compared different node selection methods during depth-limited tree search. Our result shows that depth-limited tree search is still viable when presented with inefficient forward models and exploitation-driven selection method is the most efficient in this specific domain.

CCS CONCEPTS

• **Applied computing** → **Computer games**; • **Computing methodologies** → *Game tree search*;

KEYWORDS

Pommerman, Tree Search, Monte Carlo Methods, State Machines

ACM Reference Format:
Hongwei Zhou, Yichen Gong, Luvneesh Mugrai, Ahmed Khalifa, Andy Nealen, and Julian Togelius. 2018. A Hybrid Search Agent in Pommerman. In *Foundations of Digital Games 2018 (FDG18), August 7–10, 2018, MalmÃű, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3235765.3235812

1 INTRODUCTION

Various tree search algorithms, such as Monte Carlo Tree Search (MCTS), assume and require the existence of forward models to advance the state of the game. However, not all games support fast computing forward modeling due to the factors such as complex game rules that require heavy computation to advance to the next state.

This paper tries to explore the potential of a high-performing agent in a resource-intensive, high frame rate and adversarial game environment. Specifically, we intend to search for a balanced solution between using heuristics and tree search algorithms in the Pommerman framework. We intend to enhance the agent's performance with tree search algorithms because certain problems are clearer to express the goal rather than the strategies to reach the goal. We want to present our approach with specification on the algorithm and how we modify the tree search to address the demand of the forward model as well as the selection strategy for specifically MCTS that yields the best performance.

FDG'18, August 7-10, 2018, Malmö, Sweden

ACM ISBN 978-1-4503-6571-0/18/08.
https://doi.org/10.1145/3235765.3235812

2 BACKGROUND

2.1 Pommerman Framework

Pommerman [1] is a variation to the game Bomberman (Hudson Soft, 1983). The game is played in a randomly generated 13x13 grid where four agents are trying to eliminate each other. Each agent starts in a separate corner with a single bomb and can choose one of six actions: STOP, UP, LEFT, DOWN, RIGHT, BOMB. A STOP action will be returned if no action is returned within 100 millisecond. When an agent places a bomb and that bomb denotes cardinally, the agent gains anther bomb to use. Once placed, a bomb takes about 25 frames to explode and its explosion can eliminate agents including its owner.

Figure 1: Example Game of Pommerman

Figure 1 shows a Pommerman level. In addition to the four agents (red, blue, pink and green tiles), the map contains wooden (brown tiles) and rigid walls (gray tiles) with a guaranteed accessible path to each agent. Rigid walls are indestructible and impassible, while wooden walls are impassible until destroyed by bombs. There is a 50% chance that destroying a wooden wall reveals a power up item. The power ups are **Extra Bomb** (Increase agent's ammo by one), **Increase Range** (Increase agent's blast length by one), **Can Kick** (Allow agent to kick bombs in its moving direction), and **Skull** (A random harmful power up).

While there exist other modes, for the empirical study we focus on the Free For All mode, in which each agent goes against every

[1] https://github.com/MultiAgentLearning/playground

other agent. In the case of the study our agent plays against 3 SimpleAgents since this framework and competition are relatively new and no other agents are currently available.

2.2 Heuristic Agents

A heuristic agent utilizes the current state and configuration of the fully/partial observable world to decide the next best action to take [13]. It does not model the future state of the enemy agents and rather treats them as other entities in the state. Every step a set of conditions is used to produce specified outcomes.

One popular method to implement established rules is through the model of a finite state machine [10]. A simple finite state machine consists of different states and the transition rules between those states. Different from a game state, a state of the agent describes a set of behaviors that the agent follows. Traditional finite state machine is a popular technique in the video game industry [12] as it is conceptually organized and manageable as the number of states and their transitions grow [5].

2.3 Monte Carlo Methods

Monte Carlo Methods (MCM) [9] are referred as a class of algorithms that aims to solve a problem by sampling random values and approximating the mathematical property behind the said problem. It is widely adopted in a range of domains. Most notably this technique is combined with tree search to form an algorithm called Monte Carlo Tree Search (MCTS) [1]. MCTS finds the optimal decision in a given domain by randomly sampling the decision space and building a search tree accordingly.

MCM have been used in playing games such as Go [2, 3, 6], Chess [4], Super Mario Bros [7], Starcraft [11], Hearthstone [14], etc. Pepel et al. experiments [8] show Flat Monte Carlo Search to be a stronger technique in Phantom Go (a version of Go in which opponent's stones are not revealed) when compared to techniques such as a Hybrid-MCTS and UCB-MCTS. Flat Monte Carlo Search does not build a tree and only searches its immediate child notes.

3 METHODS

We designed a heuristic agent with rulesets to find the best possible action at each step. Later in the section, we experimented with replacing some of the heuristics with depth limited tree search. In the following subsection, we explain the details of the agent and variants used during our experiments.

3.1 Heuristic Search Agent

Our heuristic search agent has two essential components: the search algorithm and a group of heuristics. As part of the heuristics, We use Dijkstra algorithm to guide the agent through the physical space towards the goal and the heuristics determine the goal for the search algorithm. We use a finite state machine to manage the current active heuristic. The Finite State Machine Agent is divided into three separate states: **Explore**, **Attack**, and **Evade**. The pseudocode for the agent is provided in Algorithm 1.

Exploration Heuristics are used when the agent is safe. It determines if it should place a bomb near a wood block to make a traversable path which helps in discovering power-ups, try to pick up a positive power-up, or move to the least visited tile on the map if there are no more wooden blocks or power-ups.

Algorithm 1 Heuristic Search Agent

```
1:  procedure GetAction
2:      if EvadeCondition() then
3:          EvadeHeuristic()
4:      else if AttackEnemyCondition() then
5:          AttackEnemyHeuristic()
6:      else
7:          ExplorationHeuristic()
8:
9:  procedure EvadeCondition
10:     for direction in [Stop, Left, Right, Up, Down] do
11:         if InRangeOfBomb(direction, 5) then return True
        return False
12:
13: procedure AttackEnemyCondition
        return Agent.BombCount > 0 and EnemyWithinRange(6)
14:
15: procedure EvadeHeuristic
16:     direction ⇐DirectionToSafeLocation()
17:     if InRangeOfBomb(direction, 5) then
          return heuristicAgentOrTreeSearchAgent(evade)
        return direction
18:
19: procedure AttackEnemyHeuristic
20:     if EnemyWithinRange(4) then
          return heuristicAgentOrTreeSearchAgent(attack)
        return directionToEnemy()
21:
22: procedure ExplorationHeuristic
23:     if WoodTileOrPowerupExist() then
          return heuristicAgent(explore)
        return exploreLeastVisitedTiles()
```

Evade Heuristic avoids bombs. Bombs are dangerous if its tick count is less than the tick threshold: $5 + 2 \times bomb\_count$, where $bomb\_count$ is the number of bomb surrounding the agent. It filters all passable positions and selects the closest position thats not under threat as a target location to navigate toward.

Attack Enemy Heuristics are used when an enemy is in Manhattan distance of 6. The agent moves towards the enemy and tries to place a bomb next to it.

Accidental Suicide Heuristics are embedded to avoid accidental suicidal actions. At any time, the agent want to place a bomb in any of the previous states. It will only place it if all the following conditions are satisfied: agent has enough ammo and it is able to escape to a safe location that is connected at least two nearby passable positions.

3.2 Heuristic Search Agent with Tree Search

On average, advancing a game by one clock tick takes 1 ms and creating a new game node with a game state copy take 2 ms on an i7-6700HQ CPU. Using a tree search algorithm won't work as the depth will be short and most of the rewards and actions in the game are delayed. For example, a bomb takes about 25 ticks to explode.

Given the restricted time to return a move, the tree search would not produce a depth of 25 moves.

To kill the enemy, ideally we would want to trap them with bombs. However, this can't be coded easily since the enemy models are unknown. Similar to bomb evading, enemy actions such as blocking the selected exit route are also hard to predict. As such we use depth limited tree search to replace the attack/evade heuristics for our agent, depth limited because of time constraints.

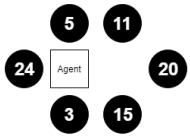


Figure 2: An example of evade score function. Bombs are represented by black circles with their ticks before exploding. The agent is currently in the way of two bombs with lower than 10 clock ticks.

Evader Score Function is a score function used by the tree search during the evade state and is as follows:

$$S_{evade} = 100 - \sum_i p_i \cdot 25 \cdot \frac{11 - tick_i}{10} \tag{1}$$

where i denotes each bomb, p_i evaluates if bomb i can reach the agent **and** has clock tick less than 10 and returns 1 if so and 0 otherwise, $tick_i$ denotes the tick rate of bomb i. The score starts as 100 and only considers bombs with clock ticks under 10 and with explosions that will hurt the agent. The lower the clock tick of a bomb, the greater the deduction to the score. Figure 2 demonstrates the evader score function.

Attacker Score Function is a score function used by the tree search in the attack state. Each tree search only focuses on one single enemy as the target. The target enemy is modeled as a random agent taking random moves during each step. The goal is defined as the target enemy is surrounded by impassible objects such as rigid, woods, bombs or agents so that it's less likely to escape the explosion. The state evaluation starts with 0 score and accesses the target's surroundings by the following:

$$S_{attack} = 100 \cdot (1 - \frac{emptySafeArea}{floodFillArea}) \tag{2}$$

where $floodFillArea$ is calculated using flood fill algorithm from the target position and $emptySafeArea$ is traversable tiles not in range of bombs. Figure 3 demonstrates how the score function works. In this figure, there are total of 4 tiles that are safe(white squares in figure 3d), while the total area is 13 (white squares in figure 3b). The final score for the state is $100 \cdot (1 - \frac{4}{13}) \approx 69$.

We used a Breadth First Search as a base line to show that a guided search has an advantage over uninformed tree search. For all tree searches, we limit the available moves for observation by removing moves that lead to immediate death or no change in the agent's current game state. For example, we remove moving left if the left of the agent is a block. In all tree search methods, the final action is that which leads to the best score.

Breadth First Search (BFS) is an uniformed search algorithm as it does not exploit the states of the board when expanding its tree. The BFS agent follows a standard approach of starting at the root node, denoted by the current state of the board, and explores all the neighbor nodes before moving to the next level of neighbors. If at any time the search reaches the capped maximum time, it will end the search and select the best action corresponds to the best node in the frontier.

Monte Carlo Tree Search (MCTS) is a stochastic tree search algorithm that utilizes Monte Carlo Simulations to approximate the value for the nodes in the tree [1]. MCTS consists of 4 steps: selection, expansion, simulation and backpropagation. During the selection phase, the agent uses the UCB1 function to select the next node in the tree. The following equation explains the UCB1 equation,

$$UCB1_i = \overline{X_i} + C\sqrt{\frac{2 * \ln N_i^p}{N_i}} \tag{3}$$

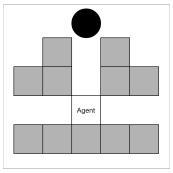
where $\overline{X_i}$ is the exploitation term of the equation and $\sqrt{\frac{2*\ln N_i^p}{N_i}}$ is the exploration term. The exploitation term is the average of the simulated scores. In exploration term, N_i^p is the parent visit count, N_i is the current node visit count. C is a constant. Through testing we found that a C value of 25 results in the best performing agent, as the exploitation term ranges [40, 100] while the exploration term ranges [0.6, 2.4]. During the expansion phase, MCTS creates a new node and add it to the tree. During the simulation phase, MCTS executes random actions for length between [1,3] or till reach a termination condition. During backpropagation, the score of simulation is used to update the exploitation terms of the current node and all of its ancestors.

Flat Monte Carlo Search (FMCS) only expands the root node's immediate children and doesn't allow tree growth compared to MCTS [1]. We perform simulated play on the immediate children and aggregate the current child's score with the simulated play score. We tested FMCS with four selection functions: **Random selection** selects child randomly for the simulations, **UCB1 selection** uses UCB1 equation shown in equation 3 with C equal to 25, **Exploitation selection** uses the exploitation only, and **Exploration selection** uses the exploration term only.

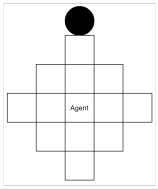
4 RESULTS

In this section, we show the results for the experiments performed on the two agents with all their variants in the Pommerman FFA mode. It should be noted that the framework's base line agent, the SimpleAgent, has roughly a 21.67% win ratio against 3 SimpleAgents. SimpleAgent operates on heuristics set by the author of Pommerman framework. Figure 4 displays the results of each agent playing 300 games against 3 SimpleAgents. From the figure, it is clear that using a tree search is better on average than using heuristic or uninformed search such as BFS, Explore, or Random Selection. The only edge case is the MCTS algorithm, we think that the tree growth with performing simulation wasted some of the time that was used for more evaluations as only tree growth (in BFS) performs better and only simulations (in FMCS) performs better.

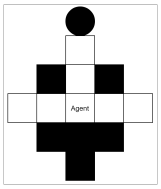
To compare the result between different agents, we conducted Binomial test between each pair of agents. We fixed the seed for all the agent for each run in the 300 runs and compared number of



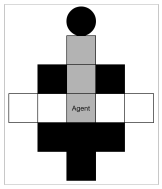
(a) Current game state with walls(gray squares), bomb(black circle) and agent



(b) All tiles within 2 spaces, total area of 13



(c) Black squares indicate unreachable tiles due to walls



(d) Grey squares indicate reachable tiles in range of bomb

Figure 3: A scenario for the attack score function.

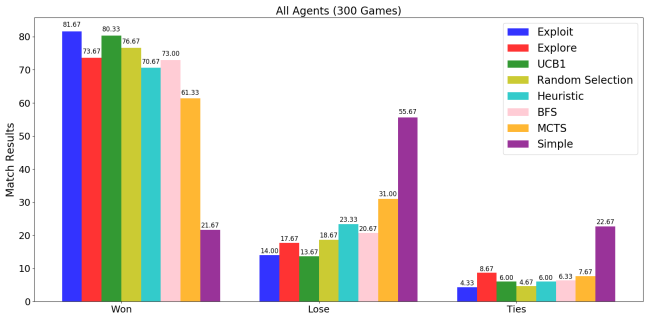


Figure 4: Win, Loss and Tie rates for each agent out of 300 game experiments

Table 1: The win rate of the row agent over the column agent. * means the p-value is less than 0.05, while ** means the p-value is less than 0.01.

	Exploit	Explore	UCB1	Random	Heuristic	BFS	MCTS	Simple
Exploit	-	58.0%*	48.9%	56.5%	64.1%**	62.7%*	76.3%**	95.9%**
Explore	-	-	40.3%*	45.8%	54.1%	50.9%	64.5%**	90.6%**
UCB1	-	-	-	54.9%	63.8%**	60.4%*	73.7%**	94.9%**
Random	-	-	-	-	57.9%*	55.3%	68.3%**	94.1%**
Heuristic	-	-	-	-	-	47.3%	64.3%*	90.2%**
BFS	-	-	-	-	-	-	63.8%*	92.3%**
MCTS	-	-	-	-	-	-	-	87.5%**

times each agent won over the three SimpleAgent while the other agents didn't. Table 1 shows the percentage of wins where the row agent has won over the column agent. From the table, it is clear that all our agents performed well above the SimpleAgent, while the Exploit Agent on average out performed all the other agents.

5 CONCLUSION

This paper has investigated possible tree search techniques integrated as the attack and evade state within a finite state machine, in which the other states are modeled to perform using heuristic functions. We experimented with several search methods such as BFS, MCTS, and FMCS. Our result shows that our agents performs significantly better than the provided SimpleAgent and that using depth-limited tree search for attack/evade states slightly outperform hand-made heuristics.

As for future work, we propose an investigation of comparing agent performance in a partially observable world; the agent's view of the board would be limited to a small surrounding radius. We should also test the level of agents against agents other than the SimpleAgent to analyze and measure the performance of our agents on a global scale. One further possible improvement is to learn the score function using a neural network instead of hand designing.

ACKNOWLEDGEMENTS

Ahmed Khalifa acknowledges the financial support from NSF grant (Award number 1717324 - "RI: Small: General Intelligence through Algorithm Invention and Selection.").

REFERENCES

[1] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.
[2] Christopher Clark and Amos Storkey. 2015. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*. 1766–1774.
[3] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. 2010. FuegoàĂŤan open-source framework for board games and Go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 4 (2010), 259–270.
[4] Hilmar Finnsson and Yngvi Björnsson. 2011. Game-tree properties and MCTS performance. In *IJCAI*, Vol. 11. 23–30.
[5] Edmond S L Ho and Taku Komura. 2010. A finite state machine based on topology coordinates for wrestling games. *Computer Animation and Virtual Worlds* 22, 5 (2010), 435–443.
[6] Shih-Chieh Huang and Martin Müller. 2013. Investigating the limits of Monte-Carlo tree search methods in computer Go. In *International Conference on Computers and Games*. Springer, 39–48.
[7] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. 2014. Monte mario: platforming with mcts. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 293–300.
[8] Tom Pepels, Tristan Cazenave, and Mark HM Winands. 2015. Sequential halving for partially observable games. In *Computer Games*. Springer, 16–29.
[9] Christian P Robert. 2004. *Monte carlo methods*. Wiley Online Library.
[10] John E Savage. 1998. *Models of computation*. Vol. 136. Addison-Wesley Reading, MA.
[11] Dennis Soemers. 2014. *Tactical planning using MCTS in the game of StarCraft*. Ph.D. Dissertation. MasterâĂŹs thesis, Department of Knowledge Engineering, Maastricht University.
[12] Mark Oude Veldhuis. 2010. Artificial Intelligence techniques used in First-Person Shooter and Real-Time Strategy games.. In *Human Media Seminar: Designing Entertainment Interaction*, Vol. 2011. Citeseer.
[13] Vincent Vidal and others. 2004. A Lookahead Strategy for Heuristic Search Planning.. In *ICAPS*. 150–160.
[14] Shuyi Zhang and Michael Buro. 2017. Improving hearthstone AI by learning high-level rollout policies and bucketing chance node events. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*. IEEE, 309–316.