# A Hybrid Search Agent in Pommerman - Technical Statement

Hongwei (Henry) Zhou

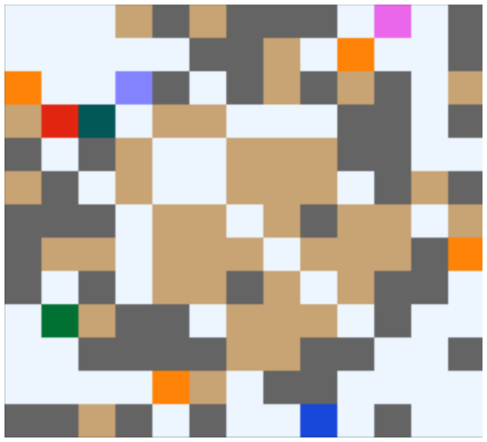
## Overview

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 factors such as complex game rules that require heavy computation to advance to the next state.

This is a class project that I worked on in a Game AI class with two other people. In this work, we try to explore the potential of a high-performing agent in a resource-intensive, high frame rate and adversarial game environment. Specifically, the main work was searching for a balanced solution between using heuristics and tree search algorithms in the Pommerman framework. Our solution enhanced the agent's performance with tree search algorithms because for certain problems it is easier to express the goal rather than the strategies to reach the goal. The notable technical work here is (1) a novel blend between rule-based approach and tree-search-based approach and (2) testing different heuristics for tree search.

## Pommerman Framework

Pommerman is a variation of the game Bomberman. 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 milliseconds. When an agent places a bomb and that bomb explodes, the agent gains another bomb to use. Once placed, a bomb takes about 25 ticks to explode and its explosion can eliminate agents including its owner.



The above figure 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). There were multiple game modes available. For our work, we focused mainly on Free for All, where all four agents are trying to eliminate each other.

**Heuristic Agent**

The Pommerman framework provides a simple gameplay agent that is completely rule-based. This agent is an efficient player already. It is able to explore the map, place bombs to clear out the wooden walls, get upgrades, and evade bombs when appropriate. The most standout downside of this agent is its attack behavior, where it simply places a bomb immediately when it sees an agent, lacking any strategic insight. It uses the Dijkstra algorithm for path finding. We'll call this provided agent Simple Agent.

We first started organizing and adding additional logic to the existing code. We grouped different actions together and eventually came down to three different states ordered in descending priority: *Evade*, *Attack* and *Explore*. Our usage of the word "state" does not imply that this is a finite state machine. The code does not recognize the previous state it was in before. It simply runs conditional testings from the beginning to end every time. We'll call it the Heuristic Agent.

The Evade state is entered when any direction the agent can go (STOP, UP, LEFT, DOWN, RIGHT) is in range of a dangerous bomb. The bomb is defined as dangerous when its tick (countdown to explosion) is less than *5 + 2 * bomb_count*. The *bomb_count* is the number of bombs surrounding the agent. The rational is that the more bombs surround the agent, the earlier the agent needs to get into the Evade state, because it might take more time for the agent to escape.

When in the Evade state, the agent simply travels to the closest safe position using the Dijkstra algorithm. One downside of this method is that the agent will simply stop when any of its adjacent spaces is in range of a dangerous bomb. A better behavior, instead of stopping once out of the bomb range, is continuing to explore the safe areas.

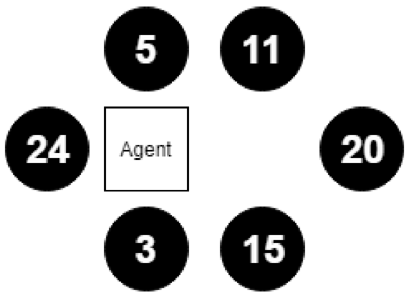
The Attack state is entered when the Evade state condition fails, and the agent possesses bombs and is within the distance of 6 spaces from an enemy. During the initial stage, the agent simply places down bombs when the bomb explosion will cover the enemy's *current* position. This is a rather poor attack behavior because it is too trigger happy. The bomb is not strategically placed to trap the enemy so the enemies tends to be able to escape the bomb.

The Explore state is entered when the two above fail. The agent will simply prioritize getting reachable upgrades, and try to eliminate the wooden walls when there is one in the way. Combining with the Evade state, the agent tends to act like a mine digger, where it places down bombs next to wooden walls and stays in a safe location to wait for the wooden walls to get cleared out.

The Heuristic Agent already managed to achieve a good win rate against the Simple Agent. A large reason comes from the condition for the Evade state. Originally, the Simple Agent considers any bomb in range to be dangerous, while our Heuristic Agent only considers a bomb dangerous when it's below *5 + 2 * bomb_count* ticks from exploding. This allowed the Heuristic Agent to stay in the Attack state longer and to be more aggressive.

**Problems with Heuristic Agent**
One of the reasons the attack behavior is so primitive is because there is no obvious rule-based solution to attack strategies. In this game, the bomb placement is dependent on the position of the enemy and the shape of the terrain. The ideal move is to use bombs to trap the the enemy agent, since bombs block movement once placed. But this is a rare situation. Most of the time, placed bombs do not form perfect entrapments. It is not obvious how to hardcode good bomb placement strategies.

Another problem is with the Evade state. The agent does not consider its maneuver when it's surrounded by bombs. Take the example in the figure below



The black squares represent bombs and the numbers indicate the ticks left before explosion. For the Heuristic Agent, there is no solution to this situation because there is no reachable safe location. But we can see that the agent can utilize the difference in bomb tick to avoid death. If the agent rests in the right position, the top left and bottom left bombs will explode first, letting the agent escape. One possible solution is to have different spaces ranked by the degree of danger based on bomb ticks. But we chose a different approach.

The above two problems do not have obvious rule-based solutions, because rule-based solutions determine actions *based on the current state of the board*. What is needed is for an agent to determine actions based on *hypothetical future states*. This is where we decided to implement Monte Carlo Tree Search to solve the above two problems in one stroke.

**Monte Carlo Tree Search**
To enable tree search, we needed to implement a forward model, which is a function that returns the next frame of the game after we provide a hypothetical action for our agent. One problem was to determine the behavior of the enemy agent. We decided to implement an agent that randomly takes a movement action (STOP, UP, LEFT, DOWN, RIGHT). The main motivation was that, for the Attack state, it's sufficient to assume that the enemy will move randomly, so we can determine the best bomb placement to restrict its movement.

We decided to implement Monte Carlo Tree Search, which is a combination of Monte Carlo Method and Tree Search. Monte Carlo Methods (MCM) [1] are referred to 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) [2]. MCTS finds the optimal decision in a given domain by randomly sampling the decision space and building a search tree accordingly.

Compared to traditional tree search algorithms such as Breadth First Search (BFS) and Depth First Search (DFS), MCTS does not require exhaustive search in either breadth nor depth. This is because of two reasons: (1) MCTS replaces end conditions with scoring functions that determine the quality of each hypothetical state, which we'll discuss more in the **Scoring Function** section (2) based on the scores of explored states, MCTS has a selection function that determines which branch of the search space is more worthy to explore. We'll discuss selection functions further in the **Flat Monte Carlo Search** section.

**Problem with Forward Model**

Having a forward model, we could evaluate hypothetical future game states. But as mentioned, the Pommerman environment enforces a 100 millisecond limit for decision making. On average, advancing a game by one frame takes 1 ms and copying a game state takes 2 ms on an i7-6700HQ CPU. This makes certain end conditions or scoring functions unrealistic for tree search. For example, we cannot define our desirable condition as when the enemy is eliminated because a bomb takes about 25 ticks/frames to explode. We need to define scoring functions with more immediate returns, which means that the quality of the state is close to immediately accessible after an hypothetical action is taken.

**Scoring Functions**

For the Evade state, the scoring is given below:

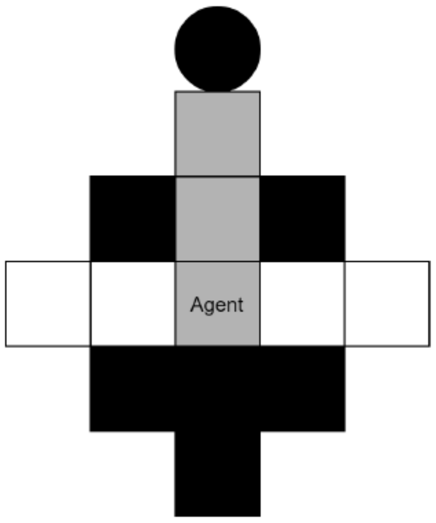
$$Score_{evade} = 100 - \sum_{i} p_{i} \cdot 25 \cdot \frac{11 - tick_{i}}{10}$$

where $i$ denotes each bomb, $p_i$ evaluates two conditions: (1) if bomb $i$'s explosion can reach the agent and (2) if bomb $i$ has tick less than 10. $p_i$ returns 1 if the above two conditions are satisfied and 0 otherwise, $tick_i$ denotes the tick rate of bomb $i$. The score starts as 100 and only considers bombs with ticks under 10. The lower the tick of a bomb, the greater the deduction to the score. This solves the problem stated in the previous section, as the value of a hypothetical state is evaluated based on the ticks of the bombs.

The scoring function for the Attack state is a little more complicated. The basic idea is that we want to place bombs in a way that covers as much of the enemy's navigable areas as possible. The scoring function only focuses on one single enemy target. The state scoring starts with 0 and accesses the target's surroundings by the following:

$$Score_{attack} = 100 \cdot (1 - \frac{emptySafeArea}{totalArea})$$

$totalArea$ indicates the area within a given range (in this case the range is 2, therefore this value is always 13, as shown below) and $emptySafeArea$ is safe and traversable spaces within the total area. Let's take the below example:



The Agent square indicates the enemy target to be eliminated. The given range for $totalArea$ is 2, so we only look at spaces within the distance of 2 around the enemy. Therefore, the $totalArea$ is 13. The black square is the bomb, and the gray squares are within the explosion range of the bomb. The black squares are the unreachable spaces due to walls, bombs or other agents. The $emptySafeArea$ is the area of the white squares, where the enemy can reach and stay safe. In this example, the score for this state is $100 \cdot (1 - \frac{4}{13}) \approx 69$. Higher scores mean fewer reachable safe spaces for the enemy agent.

**Flat Monte Carlo Search**

However, having more immediate scoring functions does not completely solve the problem with expensive forward models. To see how we solve this problem, we need to understand how MCTS works. Standard MCTS has four stages: Selection, Expansion, Simulation and Backpropagation. The key to understanding these stages is to distinguish between hypothetical future states that are *remembered* and hypothetical future states that are *played*. The algorithm stores a tree structure with all remembered future hypothetical states. The selection stage chooses one leaf node within this tree structure. The expansion stage chooses a random child of that leaf node and appends it to the tree structure (new hypothetical states are remembered). The simulation then randomly plays from that appended node until a certain end condition is met. During this stage, the future hypothetical states are played but not remembered, thus not appended to the tree structure.

Once the simulation ends, the resulting hypothetical state is evaluated with a scoring function. Each remembered game state in the tree structure stores two additional values: (1) the average of all its simulated/played scores and (2) how many times it's been visited during the selection stage. In the backpropagation stage, the resulting score of the simulation/play updates the two values of all the nodes in that specific branch, all the way from the leaf back up to the root node (hence backpropagation).

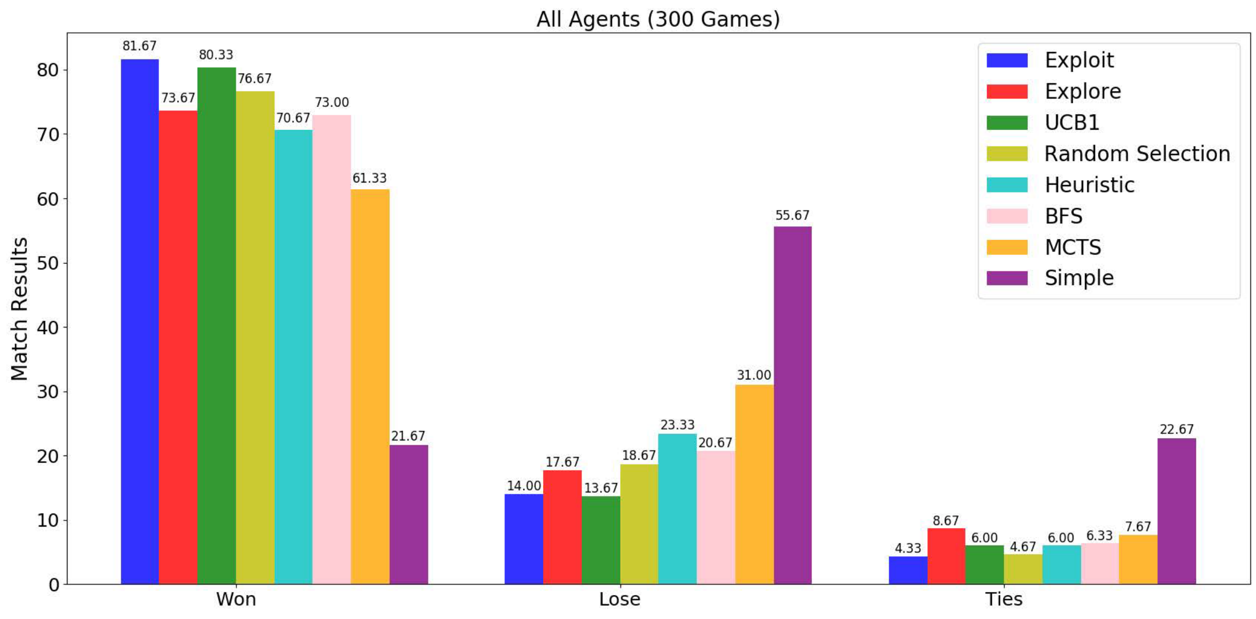
As mentioned above, creating a new game state copy takes 2 ms. This operation is needed for the expansion stage because a new state copy is created to append to the tree structure. To shave this time off, we decide to implement Flat Monte Carlo Search (FMCS), which only expands on the root node. The tree structure only has a depth of 1.

The last element of our solution is about the selection stage. How does the MCTS choose which remembered state to start the simulation/play? The standard solution is the UCB1 equation:

$$UCB1_i = \overline{X}_i + C\sqrt{\frac{2 \cdot ln N_i^p}{N_i}}$$

where $\overline{X}_i$ is the *exploitation* term of the equation and the $\sqrt{\frac{2 \cdot ln N_i^p}{N_i}}$ 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, and $N_i$ is the current node visit count. $C$ is a constant that balances the exploitation and exploration. The easy way to understand this is that if $C$ is low, the equation values the average score. As a result, the nodes with higher simulated average scores are chosen more - hence exploitation. If $C$ is high, the equation values nodes less visited in the tree structure - hence exploration.

**Result / Conclusion**



The graph shows the Win/Loss/Tie rate of the tested agent against 3 Simple Agents for 300 games.

- The Simple agent is the rule-based agent provided in the Pommerman framework.
- Both BFS and MCTS agents are tree search agents properly implemented.
- The Heuristic agent is the refined rule-based agent described in the **Heuristic Agent** section.
- The rest are Flat Monte Carlo Search agents described in the **Flat Monte Carlo Search** section with different selection functions.
    - The Random Selection agent randomly selects the child for simulation.
    - The UCB1 agent uses the full UCB1 equation with a $C$ value of 25.
    - The Explore agent only selects nodes based on the exploration term in the UCB1 equation.
    - The Exploit agent only selects nodes based on the exploitation term.

As indicated in the result, Exploit agent and UCB1 agent perform considerably better than other agents. I hypothesize that Flat Monte Carlo Search works because the nature of the game does not require too much long-term planning. In addition, the flat search approach allows more simulation to be done. As a result, the sampling manages to get closer to the ground truth (the actual best action to take in that moment).

## Bibliography

[1] Christian P Robert. 2004. Monte carlo methods. Wiley Online Library.

[2] 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.