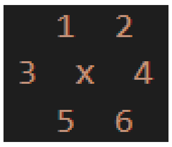
AI Capstone HW2 - MCTS Agent

Game Introduction

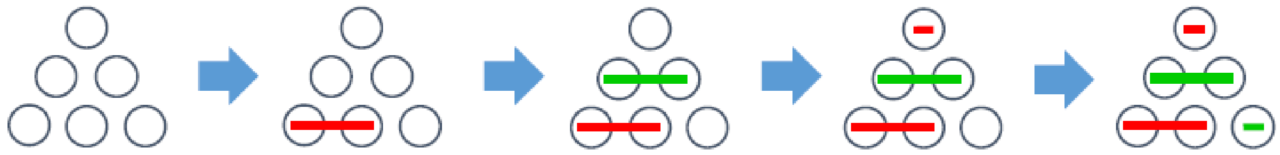
The basic game rules:

- Players take turn crossing out the cells.
- In each move, a player can choose to cross out one, two, or three remaining cells. The selected cells have to be a contiguous straight line for one of the six directions like below.



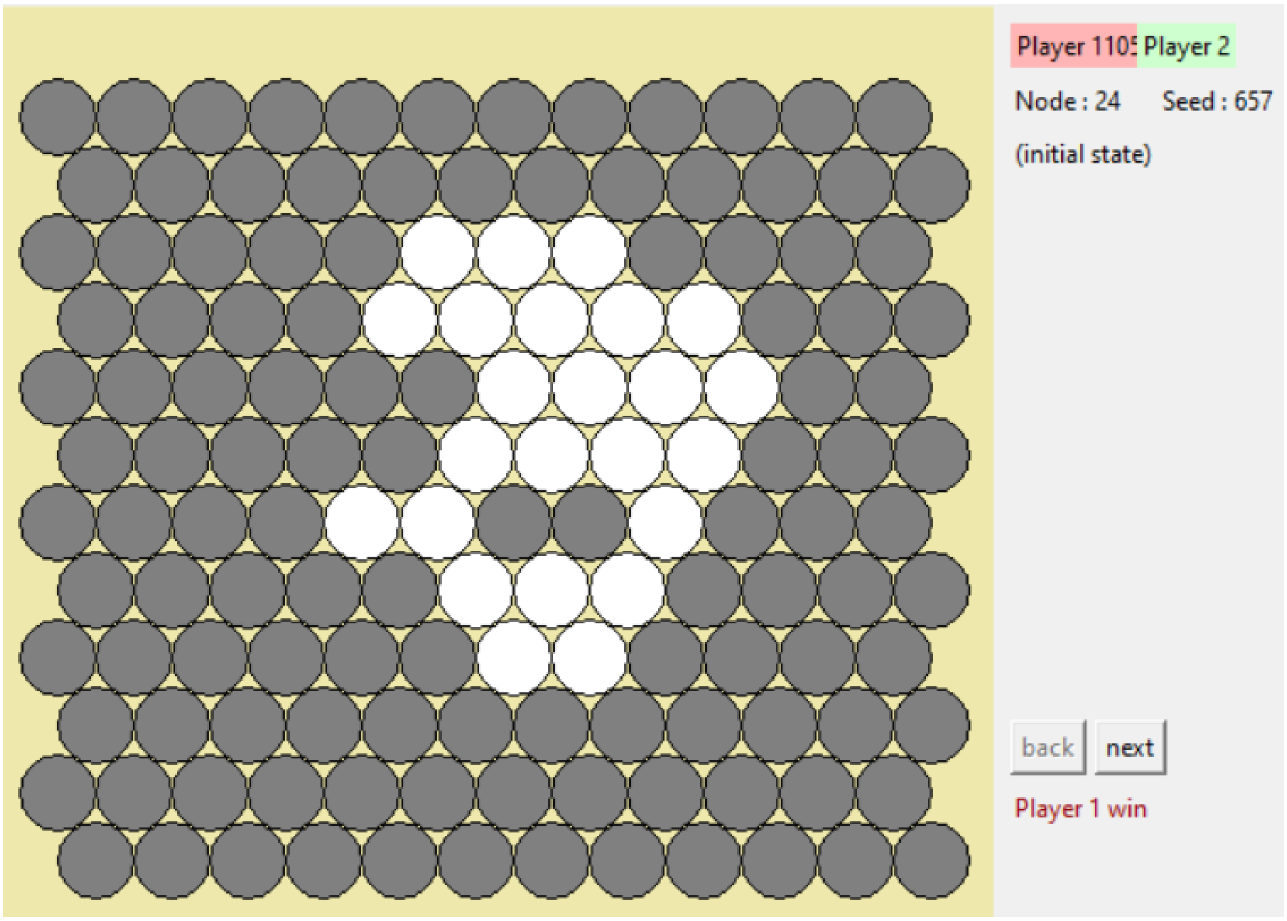
```
  1   2
3   x   4
  5   6
```

- The player that crosses out the last remaining cell loses the game.
- Below is an example sequence where the second player (green) loses:



Game Environment:

- The game board (an example is shown below) is an 12x12 region. However, only a subset of the cells (white cells) are actually used in a game (24 in the example).



- The subset of cells used is generated randomly for each game (random seed is in the range of 1~1000) and always form a connected region. (The size of the subset will fall in the range of 12~32.)

Algorithm - MCTS

MCTS Explanation:

MCTS (Monte Carlo Tree Search) is an algorithm used in game-playing and adversarial search. It works by building a tree of possible states and actions, where each node represents a state and each edge represents a possible action. The tree is built by repeatedly selecting the best node with a selection policy and simulating the possible outcomes of choosing it. After generating a large number of simulations, update all nodes' reward along the path from leaf simulated node to root node. At last, it selects the best possible action.

MCTS Implementation:

The contents below are MCTS implementation details and its explanation. There are 3 big steps for implementation:

1. Initialization: Initialize GameState by mapStat and gameStat and initalize MCTS_Agent by current game state and creating a root node for the game tree.
2. Search: Do Monte Carlo search for this Monte Carlo Tree. It can be splitted into these four following steps:
    1. Selection: Starting from the root node, select a node with a selection policy, which I use UCT algorithm here. Meanwhile, select the node which it hasn't been visited.

        - UCT algorithm is a common used selection policy, which balances exploration (choosing unexplored nodes) and exploitation (choosing nodes with high expected rewards).
    2. Expansion: Expand one leaf node if it was already visited and the game is not terminated with every possible actions it can make.
    3. Simulation: Do simulatation from current state which I get by selection and expansion until the game is terminated. In my implemention, I simulate with choosing action randomly.
    4. Backpropagation: update all nodes' reward along the path from simulated leaf node to root node. My reward policy is listed below:
        - If current player wins the game, then its nodes should have reward value = 1. Meanwhile, enemy's node should have reward value = -1.
        - If current player loses the game, then its nodes should have reward value = -1. Meanwhile, enemy's node should have reward value = 1.
    5. Keep repeating steps 1~4 for up to time_out_sec second.

3. Get best action: Get the best action among all nodes with the highest value of another selection policy.

Experiments

Custom Test Bench Explanation:

For testing hyperparameters, I wrote a test bench to let two players compete with each other for a certain number of matches. Despite that, the two players will take turn to be the first hand or second hand. In this test bench, node_num and the seed of the game board are randomly chosen. On the other hand, `input.txt` for this test bench is a little bit different from the origin one. The first four lines about player IDs and the paths of executable file are the same, but the number at the fifth line means the number of different matches they want to have. The following context is an example of new `input.txt`:

```
110550174
./dist/110550174.exe
2
./Sample_2.exe
20
```

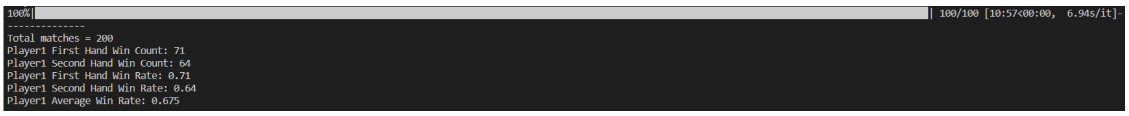
The number `20` at the fifth line means the two players will have total 2 * 20 = 40 matches on 20 different game boards, with player 1 and player 2 playing with first hand 20 times each.

Testing Hyperparameters:

When testing for the best hyperparameters, I set time_out_sec to 0.01s to avoid costing too much time. Besides, I test the hyperparameters in the order of explore_rate -> reward -> best_node -> time_out_sec, which means that I select the best hyperparameter right after testing for that hyperparameter, instead of select the best hyperparameter after testing all possible hyperparameter combinations like grid search because it's too time-consuming. The tested hyperparameters and their testing experiments are listed below (Note: My agent play as player1, and Sample2 play as player2):

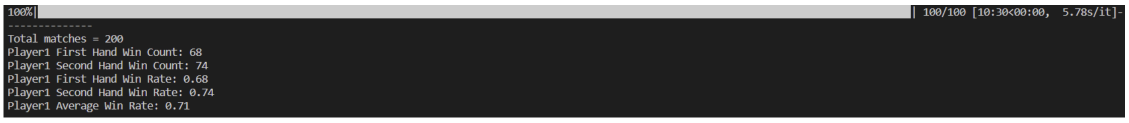
1. Explore_rate for UCT algorithm:
    - Values for testing: 2, sqrt(2), sqrt(3)

- Results:
  - 2:



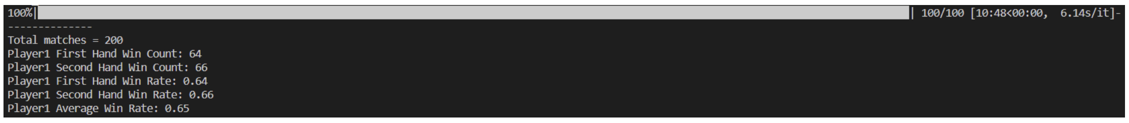
```
100%|██████████████████████████████| 100/100 [10:57<00:00,  6.94s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 71
Player1 Second Hand Win Count: 64
Player1 First Hand Win Rate: 0.71
Player1 Second Hand Win Rate: 0.64
Player1 Average Win Rate: 0.675
```

  - sqrt(2):



```
100%|██████████████████████████████| 100/100 [10:30<00:00,  5.78s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 68
Player1 Second Hand Win Count: 74
Player1 First Hand Win Rate: 0.68
Player1 Second Hand Win Rate: 0.74
Player1 Average Win Rate: 0.71
```

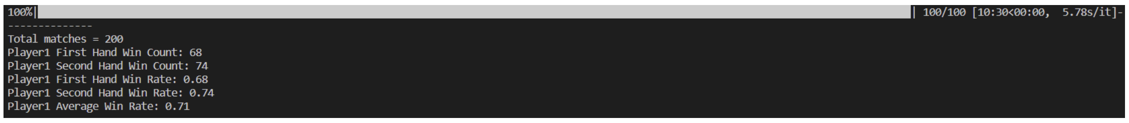
  - sqrt(3):



```
100%|██████████████████████████████| 100/100 [10:48<00:00,  6.14s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 64
Player1 Second Hand Win Count: 66
Player1 First Hand Win Rate: 0.64
Player1 Second Hand Win Rate: 0.66
Player1 Average Win Rate: 0.65
```

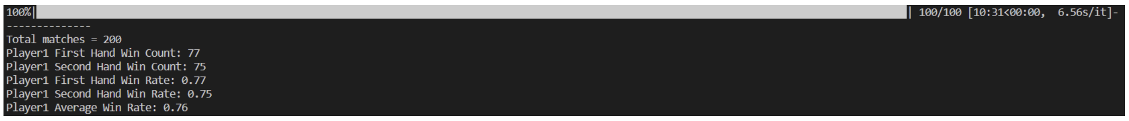
- From the results above, we can see that explore_rate=sqrt(2) has the best average win rate.

2. Reward for Backpropagation (explore_rate=sqrt(2)):
   - Values for testing: 1 for win and 0 for lose or 1 for win and -1 for lose
   - Results:
     - 1 for win and 0 for lose:



```
100%|██████████████████████████████| 100/100 [10:30<00:00,  5.78s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 68
Player1 Second Hand Win Count: 74
Player1 First Hand Win Rate: 0.68
Player1 Second Hand Win Rate: 0.74
Player1 Average Win Rate: 0.71
```

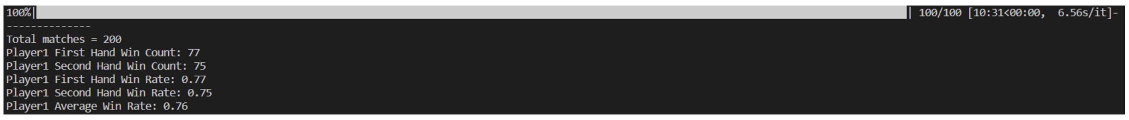
     - 1 for win and -1 for lose:



```
100%|██████████████████████████████| 100/100 [10:31<00:00,  6.56s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 77
Player1 Second Hand Win Count: 75
Player1 First Hand Win Rate: 0.77
Player1 Second Hand Win Rate: 0.75
Player1 Average Win Rate: 0.76
```

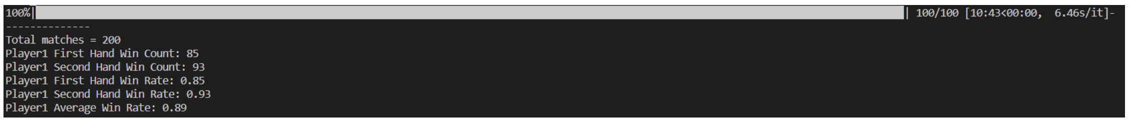
   - From the results above, we can see that reward=1 for win and -1 for lose has the best average win rate.

3. Selection policy to get the best action after searching (explore_rate=sqrt(2), reward=1 for win and -1 for lose):
   - Policy for testing: Select node's action with the most UCT value, Number of visits, or Average value
   - Results:
     - UCT value:



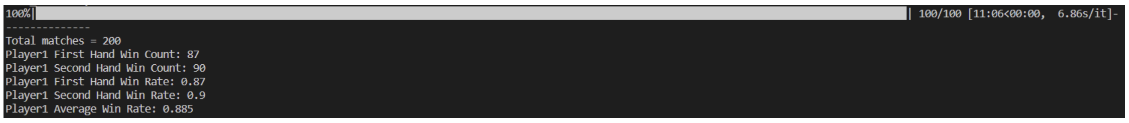
```
100%|██████████████████████████████| 100/100 [10:31<00:00,  6.56s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 77
Player1 Second Hand Win Count: 75
Player1 First Hand Win Rate: 0.77
Player1 Second Hand Win Rate: 0.75
Player1 Average Win Rate: 0.76
```

     - Number of visits:



```
100%|██████████████████████████████| 100/100 [10:43<00:00,  6.46s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 85
Player1 Second Hand Win Count: 93
Player1 First Hand Win Rate: 0.85
Player1 Second Hand Win Rate: 0.93
Player1 Average Win Rate: 0.89
```

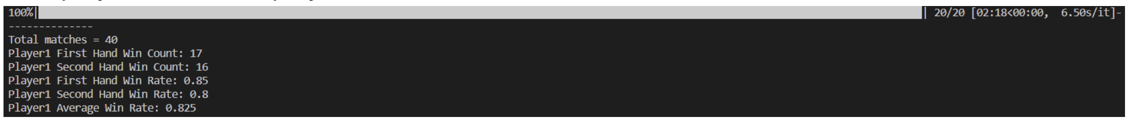
     - Average value:



```
100%|██████████████████████████████| 100/100 [11:06<00:00,  6.86s/it]-
--------------
Total matches = 200
Player1 First Hand Win Count: 87
Player1 Second Hand Win Count: 90
Player1 First Hand Win Rate: 0.87
Player1 Second Hand Win Rate: 0.9
Player1 Average Win Rate: 0.885
```

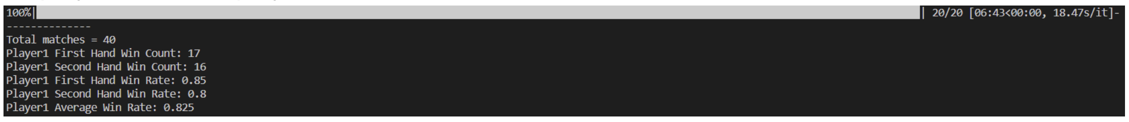
   - From the results above, we can see that using Number of visits and Average value as the selection policy has similar average win rate, so I choose Average value as the selection policy because it's more general.

4. time_out_sec for the search time limit (explore_rate=sqrt(2), reward=1 for win and -1 for lose, select policy=average value):
    - Values for testing: 0.01s, 0.1s, 1s, 5.5s
    - For testing time_out_sec, I make them compete with each other.
    - Results:
        - 0.1s(player1) vs 0.01s(player2):



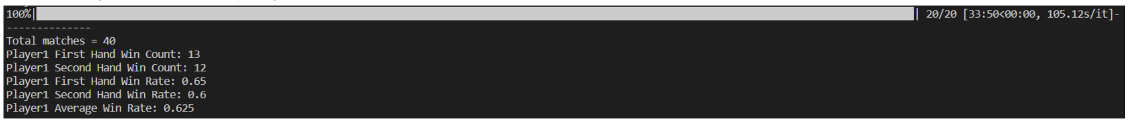
```
100%|████████████████████████████████████████| 20/20 [02:18<00:00,  6.50s/it]-
-------------
Total matches = 40
Player1 First Hand Win Count: 17
Player1 Second Hand Win Count: 16
Player1 First Hand Win Rate: 0.85
Player1 Second Hand Win Rate: 0.8
Player1 Average Win Rate: 0.825
```

        - 1s(player1) vs 0.1s(player2):



```
100%|████████████████████████████████████████| 20/20 [06:43<00:00, 18.47s/it]-
-------------
Total matches = 40
Player1 First Hand Win Count: 17
Player1 Second Hand Win Count: 16
Player1 First Hand Win Rate: 0.85
Player1 Second Hand Win Rate: 0.8
Player1 Average Win Rate: 0.825
```

        - 5.5s(player1) vs 1s(player2):

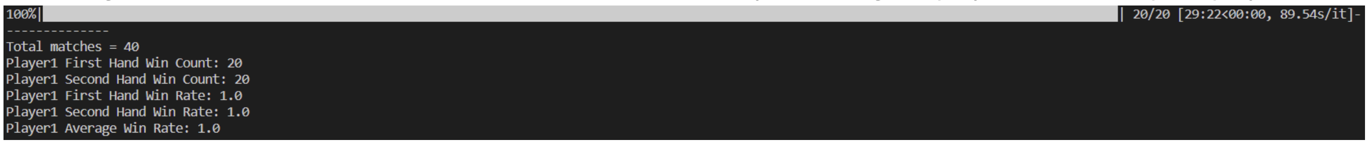


```
100%|████████████████████████████████████████| 20/20 [33:50<00:00, 105.12s/it]-
-------------
Total matches = 40
Player1 First Hand Win Count: 13
Player1 Second Hand Win Count: 12
Player1 First Hand Win Rate: 0.65
Player1 Second Hand Win Rate: 0.6
Player1 Average Win Rate: 0.625
```

    - From the results above, we can see that the performance between them is 5.5s > 1s > 0.1s > 0.01s.

Results

The image below is the result of total 40 matches between my MCTS agent(player1) and Sample2(player2):



```
100%|████████████████████████████████████████| 20/20 [29:22<00:00, 89.54s/it]-
-------------
Total matches = 40
Player1 First Hand Win Count: 20
Player1 Second Hand Win Count: 20
Player1 First Hand Win Rate: 1.0
Player1 Second Hand Win Rate: 1.0
Player1 Average Win Rate: 1.0
```

From the results above, we can see that player1 has won all the matches against player2.

Discussion

- From all the experiment and results above, I discovered something interesting:
    - Because of the small size of the game board, determining whether the game is terminated for the stopping condition of simulation step is already enough, not needing to define another custom stopping condition.
    - For some experiment, I discover that when time_out_sec is set to 5.5s, the epoch for searching starts with a few thousands and end up with more than 100 thousands when the step increases. I think this is due to the simulation at few beginning steps have to go through longer paths until the game is terminated and cost more time than those at few end steps.
- For future work if there were more time available, I would like to dive more deep into the simulation methods because the simulation method I used in this implementation is the simplest randomization, while other more complicated methods might increase MCTS agent's winning rate.