AI Capstone
Project2 Report

Team ID : 17

Team Name : Nuguseyo

110550090 王昱力, 110550093 蔡師睿, 110550110 林書愷

1 Methods

1.1 Minimax

The main objective of Minimax algorithm is to determine the optimal move for a player, assuming that the opponent will also make the best possible move. The algorithm works by recursively evaluating possible future game states, assigning a score to each state, and selecting the move that leads to the state with the highest score for the player or the lowest score for the opponent. In order to improve efficiency, we also use alpha-beta pruning. This technique can eliminate branches of the game tree that are guaranteed to be worse than previously explored branches.

Algorithm 1 Pseudocode of the $\alpha\beta$ pruning algorithm [1]

```
1: function ALPHABETA(node, depth, currentPlayer, α, β)
2:     if node.isTerminal() or depth ≤ 0 then
3:         return currentPlayer × evaluation(node);
4:     end if
5:     for all c in node.children do
6:         α = max(α, −ALPHABETA(node, depth − 1, nextPlayer, −β, −α));
7:         if α ≥ β then
8:             return β;
9:         end if
10:    end for
11:    return α;
12: end function
```

1.2 Best Reply Search

Schadd and Winands (2011) proposed a new search technique for playing multi-player games, namely Best-Reply Search (BRS), and it showed that BRS performs better in the multi-player games Chinese Checkers and Focus. This technique is

a modification of minimax search to multi-player games. Instead of allowing all opponents to make a move, only one opponent is allowed to do so.

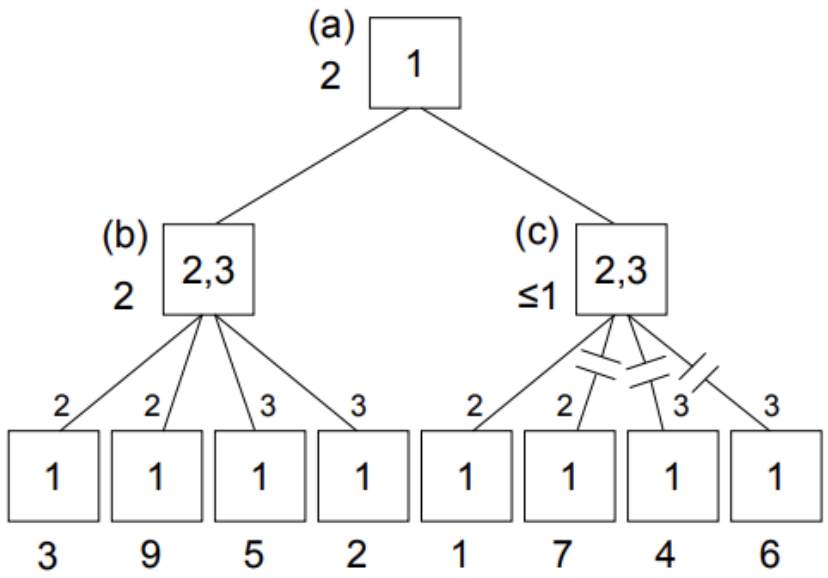

Figure 1: BRS Tree [2]

Algorithm 2 Pseudocode of Best-Reply Search [2]

```
1:  function BRS(alpha, beta, depth, turn)
2:      if depth ≤ 0 then
3:          return eval();
4:      end if
5:      if turn == MAX then
6:          Moves = GenerateMoves(MaxPlayer);
7:          turn = MIN;
8:      else
9:          for all Opponents o do
10:             Moves + = GenerateMoves(o);
11:         end for
12:         turn = MAX
13:     end if
14:     for all Moves m do
15:         doMove(m);
16:         v = -BRS(-beta, -alpha, depth-1, turn);
17:         undoMove(m);
18:         if v ≥ beta then
19:             return v;
20:         end if
21:         alpha = max(alpha, v);
22:     end for
23:     return alpha;
24: end function
```

1.3 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a decision-making algorithm commonly used in games and other domains with large state and action spaces. It consists of four main steps:

1. **Selection** : Starting from the root node of the search tree, MCTS selects child nodes iteratively until it reaches a leaf node. It is typically done using a tree policy which balances between exploration and exploitation. The tree policy we apply here is UCB :

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

, where w_i is the total score, n_i is the total number of simulations, N_i is the total number of simulations for the parent node, and c is the constant representing exploration parameter.

2. **Expansion** : Once a leaf node is reached, MCTS expands the tree by adding child nodes corresponding to possible actions from the current state.

3. **Rollout** : MCTS performs a simulation from the selected child node until a terminal state is reached. During the simulation, actions are chosen according to random policy.

4. **Backpropagation** : After the simulation, the reward obtained from the terminal state is backpropagated up the tree to update the visit counts and cumulative rewards of the nodes along the path from the root to the selected node.

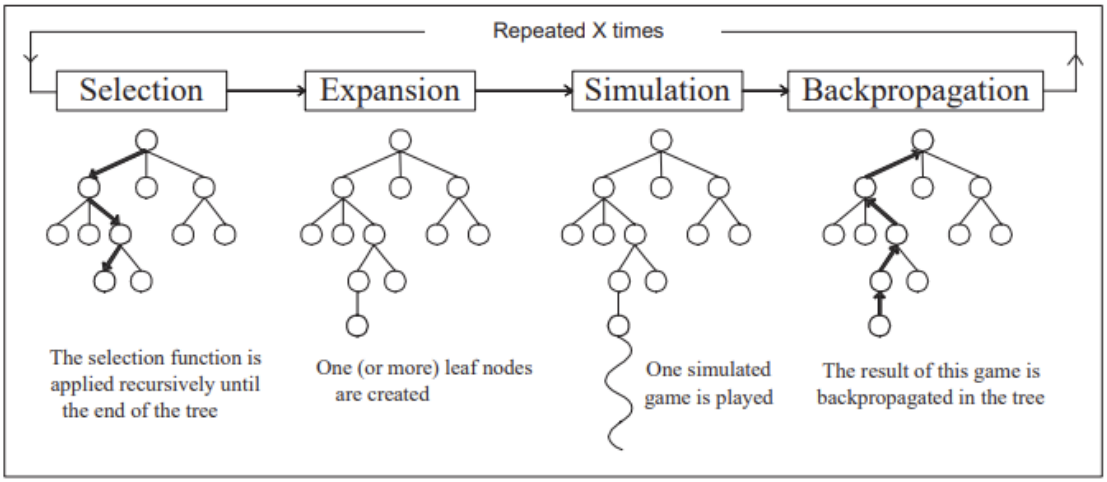

Figure 2: Scheme of MCTS [3]

1.4 Parallel MCTS

Parallel MCTS is a variant of the traditional MCTS algorithm that leverages parallelism to accelerate the search process. It can significantly reduce the search time, especially in applications with large state spaces or complex game trees. In [3], the author introduced the four types of parallel MCTS, which are leaf parallelization, root parallelization, tree parallelization with global mutex, and tree parallelization with local mutex, as shown in Figure 3.

We decided to implement root parallelization MCTS due to its ease of implementation and performance. The method works as follows. It consists of building multiple MCTS trees in parallel, with one thread per tree. When the available time is spent, all the root children of the separate MCTS trees are merged, and then we choosed the node with highest visit count.

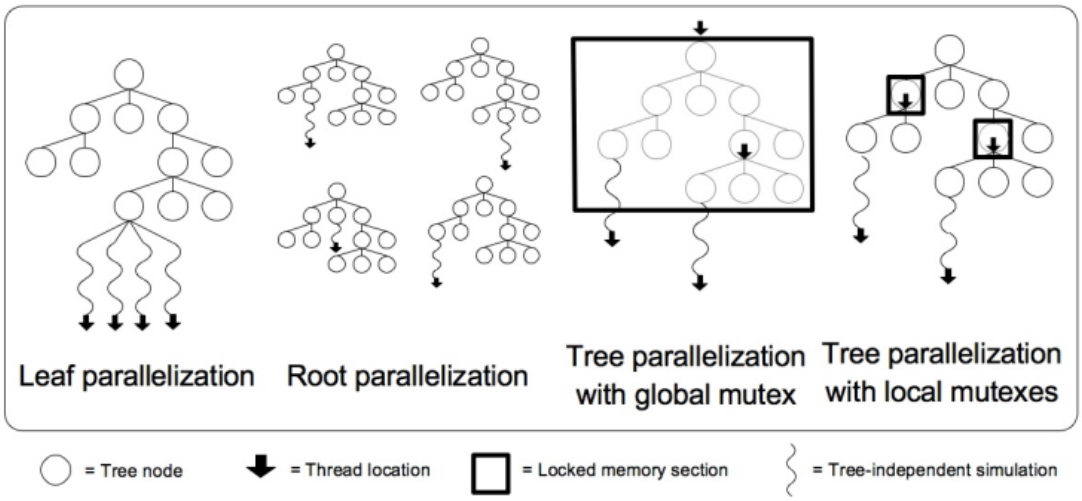

Figure 3: Four kinds of parallel MCTS [3]

2 Game Settings

2.1 Game 1

- Algorithm : Parallel MCTS (Four-thread)
- Exploration Parameter c : 0.75
- Board size : 12*12
- Playing field : 64
- Sheep per agent : 16

2.2 Game 2

- Algorithm : Parallel MCTS (Four-thread)
- Exploration Parameter c : 0.75
- Board size : 15*15
- Playing field : 100
- Sheep per agent : 32

Basically, only an additional parameter, board size, is modified.

2.3 Game 3

- Algorithm : Parallel MCTS (Four-thread)
- Exploration Parameter c : 0.75
- Board size : 12*12
- Playing field : 64
- Sheep per agent : 16
- Restriction : Unknown state of other players' sheep

In game 3, since the server won't give the state of other players' sheep. Therefore, to give algorithm enough information to do prediction we maintain sheep and map state in agent. By tracing the latest move of other player, we select the largest sheep-spot in the eight direction of latest spot and assume that it spit into half as new spot.

2.4 Game 4

- Algorithm : Parallel MCTS (Four-thread)
- Exploration Parameter c : 0.75
- Board size : 12*12
- Playing field : 64
- Sheep per agent : 16
- Additional : Cooperative play (agent1 & agent3 v.s. agent2 & agent4)

In Game 4, the game settings are identical to those in Game 1. The only difference is that we add team IDs, which are used to calculate the team scores and determine the winning agents.

3 Experiments & Results

3.1 Algorithms

Algorithm	Win Rate (%)	Avg. Score
Minimax	0%	18.67
Parallel MCTS	100%	26.36
Random	0%	15.34
BRS	0%	16.91

Table 1: Comparison of different algorithms in Game 1 for 20 times.

From the results of the 20 games in Table 1, it's evident that the MCTS algorithm outperformed Minimax, Random, and BRS algorithms. Achieving a 100% winning rate and nearly reaching the maximum score of 32. In sum, MCTS was selected to be the optimal algorithm in our project.

3.2 Parallelization

To enable MCTS algorithm to explore more possibilities, we run the MCTS on four threads at the same time for every step. As a comparison, we use one-thread MCTS and four-thread MCTS playing in game one while reset agents are samples given.

Threads	Win Rate (%)	Avg. Score
One-thread MCTS	44%	23.85
Four-thread MCTS	50%	25.28

Table 2: Comparison of various computing resources performance on Game 1.

As the outcome shows, with computing resources increasing the performance of MCTS indeeded improved significantly at both winning rate and average scores. All in all, with more computing resources, MCTS can explore more possibilities and therefore affect eventual performance.

3.3 Exploration Parameter c

Parameter c	Win Rate (%)	Avg. Score (20 games)
c = 0.75	66.7%	26.20
c = $\sqrt{2}$	33.3%	25.81

Table 3: Comparison of various Parameters in their performance on Game 1.

From the results shown in Table 3, we found better parameters to surpass the general consensus parameter, $\sqrt{2}$, however, the difference in scores isn't significant. In sum, the chosen parameters are based on different situations, and 0.75 is the one that yields better outcomes in our MCTS algorithms.

4 Future Works

If time and hardware were available, we would like to try Alpha-Zero-General [4], a reinforcement learning framework based on the AlphaGo Zero paper (Silver et al.). Alpha-Zero-General combines neural network and MCTS together, potentially leading to better performance compared to vanilla MCTS. However, according to its README.md, the network needs to train for about three days on a 6x6 Othello board. Obviously, it will take a considerable amount of time to tune the parameters as the framework is applied to our sheep game which is more complicated than Othello.

5 Contribution

- 110550090 王昱力: Design random algorithms, agent2, run experiment, report.
- 110550093 蔡師睿: Design Minimax, BRS, MCTS algorithms, agent1, report.
- 110550110 林書愷: Design agent3, agent4, Optimize algorithm performance, run experiment, report.

References

[1] "Monte-carlo tree search for multi-player games," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:196010916.

[2] M. P. D. Schadd and M. H. M. Winands, "Best reply search for multiplayer games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 1, pp. 57–66, Mar. 2011, ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2011.2107323.

[3] G. M. J. .-. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 60–71, ISBN: 978-3-540-87608-3.

[4] S. Thakoor, S. Nair, and M. Jhunjhunwala, *Learning to play othello without human knowledge*, 2016.