AI capstone project 2

teamID: 32　team name: 廢羊羊

group members: 109350008 張詠哲　110705013 沈昱宏

Method

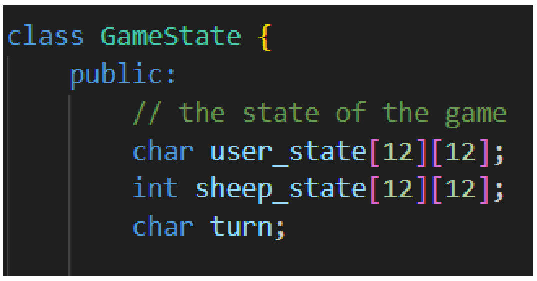
In this project. We implement 2 methods to decide the action: MCTS and MCTS + rule-based. The following are descriptions of the methods mentioned above.

1. MCTS

Before we introduce our implementation, we first simply describe the definition of our game state and MCTS node

- **game state**

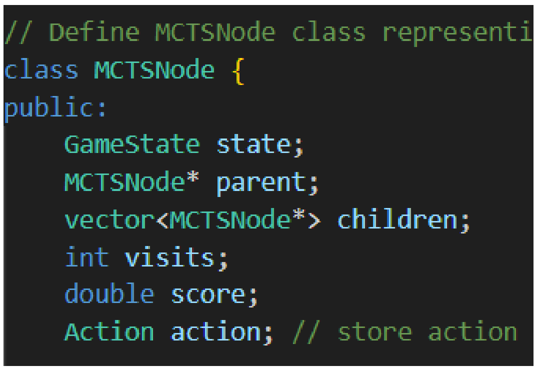
  In the game state class, we store the gamer map, sheeps map, and which player's turn.



```
class GameState {
    public:
        // the state of the game
        char user_state[12][12];
        int sheep_state[12][12];
        char turn;
```

- **MCTS node**

  In MCTS nodes, we store a game state, parent MCTS node, children MCTS nodes, number of visits to this node, and score (number of wins) for calculating UCB values, and the action that causes the stored game state.



```
// Define MCTSNode class representi
class MCTSNode {
public:
    GameState state;
    MCTSNode* parent;
    vector<MCTSNode*> children;
    int visits;
    double score;
    Action action; // store action
```

Here is a short introduction of MCTS (Monte Carlo Tree Search) and how we implement it. There are 4 phases in MCTS algorithm: selection, expansion, rollout (simulation), backpropagation.

- **Selection**

  Starting from the root node, MCTS selects child nodes using a selection strategy, which is usually UCB (Upper Confidence Bounds). The selection aims to balance exploration (trying new paths) and exploitation (choosing known good paths). The UCB formula is:

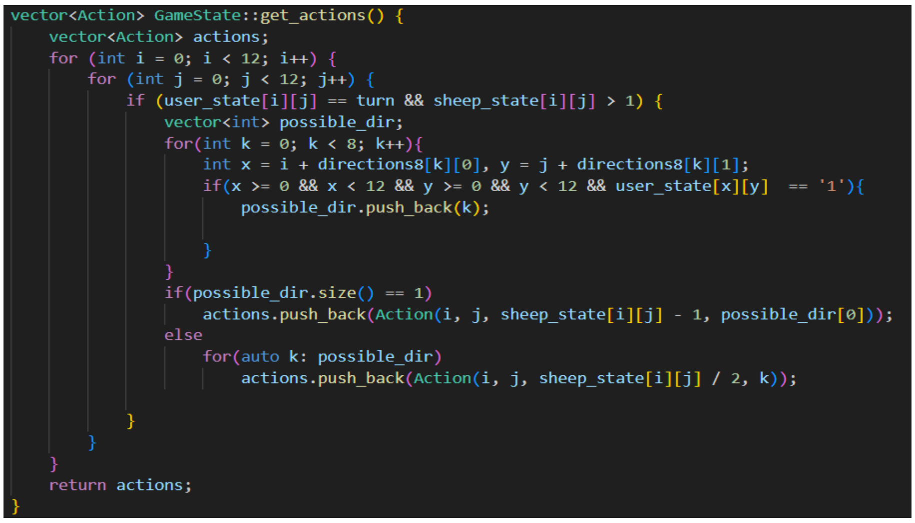
$$UCB = v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

  (v_i is estimated winning rate, n_i is the number of the times the node has been visited and N is the total number of times that its parent has been visited. C is a tunable bias parameter.)

  In this part, we take v_i as average value calculate with division of score(number of the times that wins in rollout phase) and n_i (which is $\frac{score}{n_i}$). Here we choose C = square root of 2, which is a common choice. With this formula, we choose the child node with highest UCB values recursively until there is no child node left.

- **Expansion**

  Once a node is selected, MCTS may expand it by adding child nodes representing possible moves from that state. This step increases the tree progressively. Here is how we select possible actions given a game state. We first go through the whole sheep map. When we find a point where the person is making action, we check how many directions the sheep can be split. If there is only 1 possible split direction, we append possible action (x, y, M-1, direction_index), where M stands for the number of sheeps in (x, y). This avoids the sheeps being blocked in the future. If there are more than 1 possible direction, we split half of M to all the possible split directions. Splitting the half of the sheeps instead of having the action of all possible sheep numbers [1, M-1] helps reduce the action space, making the tree smaller (but not good in some cases).

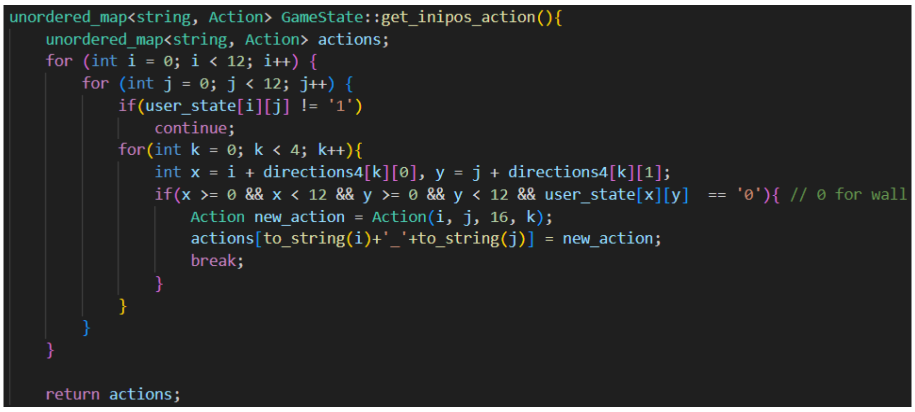


```
vector<Action> GameState::get_actions() {
    vector<Action> actions;
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 12; j++) {
            if (user_state[i][j] == turn && sheep_state[i][j] > 1) {
                vector<int> possible_dir;
                for(int k = 0; k < 8; k++){
                    int x = i + directions8[k][0], y = j + directions8[k][1];
                    if(x >= 0 && x < 12 && y >= 0 && y < 12 && user_state[x][y]  == '1'){
                        possible_dir.push_back(k);

                    }
                }
                if(possible_dir.size() == 1)
                    actions.push_back(Action(i, j, sheep_state[i][j] - 1, possible_dir[0]));
                else
                    for(auto k: possible_dir)
                        actions.push_back(Action(i, j, sheep_state[i][j] / 2, k));

            }
        }
    }
    return actions;
}
```

As for deciding initial placement. It’s like the above one. But it must fit the game constraint (must have a wall in one near place).



```
unordered_map<string, Action> GameState::get_inipos_action(){
    unordered_map<string, Action> actions;
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 12; j++) {
            if(user_state[i][j] != '1')
                continue;
            for(int k = 0; k < 4; k++){
                int x = i + directions4[k][0], y = j + directions4[k][1];
                if(x >= 0 && x < 12 && y >= 0 && y < 12 && user_state[x][y]  == '0'){ // 0 for wall
                    Action new_action = Action(i, j, 16, k);
                    actions[to_string(i)+'_'+to_string(j)] = new_action;
                    break;
                }
            }
        }
    }

    return actions;
```

- **Rollout**

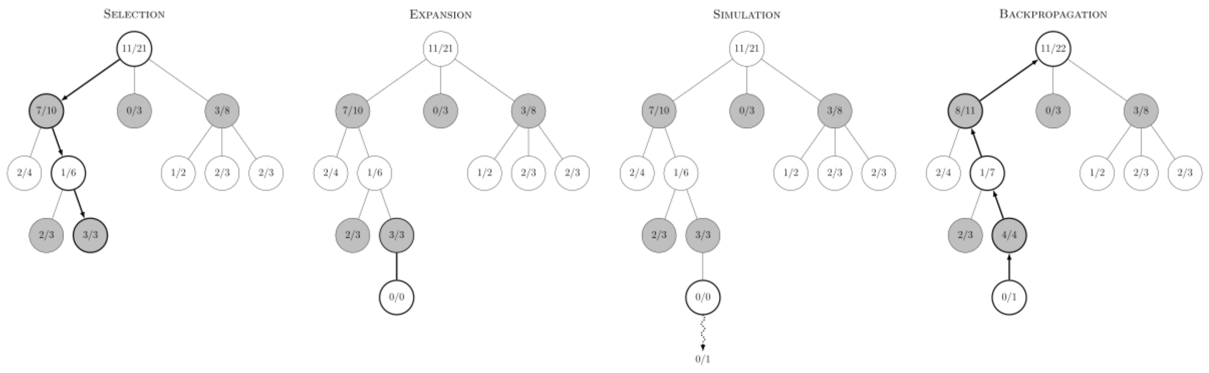
  MCTS performs simulations from the newly added nodes by playing the game until a terminal state is reached. The default policy here is to choose the action randomly for all agents. When the game ends, we return the game result (1 if our agent wins, 0 when loses) using the winning condition according to the game rule.

- **Backpropagation**

  After the simulation, updates the statistics of nodes along the path from the simulated node to the root. We update the number of wins in the node and the number of visits in the node during backpropagation.

One iteration consists of all the 4 steps mentioned above, and the number of iterations we are performing for each decision is a parameter that we can tune. Having a larger number of iterations will make a better decision, but the computational cost will be high.

A MCTS iteration follows the following schematic diagram.



(source: By Rmoss92 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=88889583)

- **Decide action**

  For the all iterations done or timeout we need to decide a action, and here we choose the action from current state with highest win rate ($\frac{score}{n_i}$).

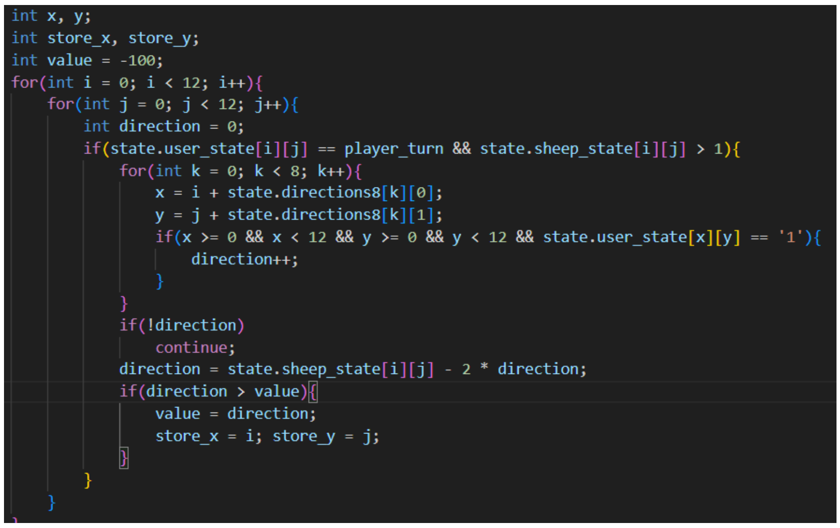
2. MCTS with rule based

In this part, the initial placement is decided using MCTS. The biggest difference is how we decide . Here is how we select an action given the game state.

We split the action decision into 3 parts: decide (x, y), split direction, and split the number of sheeps.

- **Decide (x,y)**

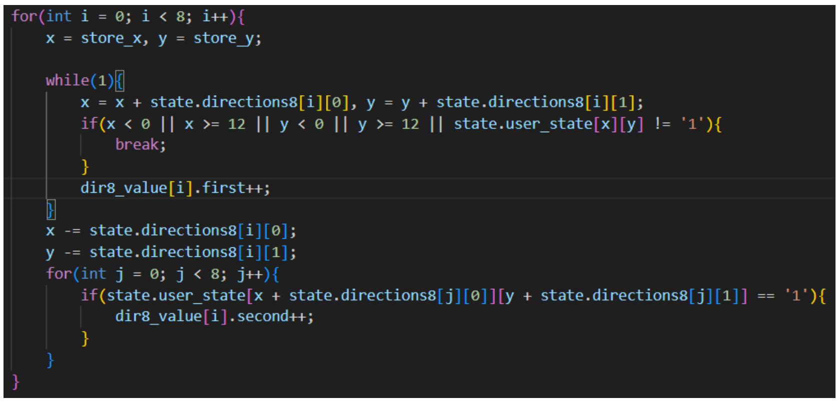
  We choose the place with less valid split direction and more sheeps that it can split. Less valid split direction indicates that it is more likely to be blocked in the future, so we should move it away as early as possible. A lot of your sheeps being blocked is a disaster to you, so if you have more sheeps in one position, you should split it to reduce the risk. Here I select the (x, y) with the largest (sheep_num - 2 * possible direction). This equation trades off between the two metrics mentioned above.



```
int x, y;
int store_x, store_y;
int value = -100;
for(int i = 0; i < 12; i++){
    for(int j = 0; j < 12; j++){
        int direction = 0;
        if(state.user_state[i][j] == player_turn && state.sheep_state[i][j] > 1){
            for(int k = 0; k < 8; k++){
                x = i + state.directions8[k][0];
                y = j + state.directions8[k][1];
                if(x >= 0 && x < 12 && y >= 0 && y < 12 && state.user_state[x][y] == '1'){
                    direction++;
                }
            }
            if(!direction)
                continue;
            direction = state.sheep_state[i][j] - 2 * direction;
            if(direction > value){
                value = direction;
                store_x = i; store_y = j;
            }
        }
    }
```

- **Split direction**

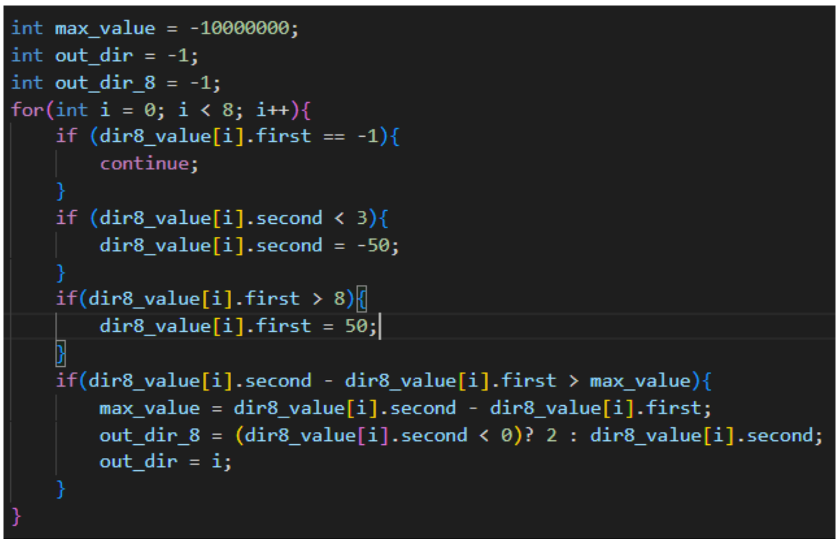
  Next, we choose the split direction. There are also two things we might want to consider, the distance between now & target position and the number of directions we can go at the target position. According to the winner calculation method, we would like our sheeps to be closer to each other to have the power 1.25 value greater. We also need to choose a more promising target point i.e. more possible directions at target directions. We first calculate the 2 things: the distance between now & target position and the number of directions we can go at the target position.



```
for(int i = 0; i < 8; i++){
    x = store_x, y = store_y;

    while(1){
        x = x + state.directions8[i][0], y = y + state.directions8[i][1];
        if(x < 0 || x >= 12 || y < 0 || y >= 12 || state.user_state[x][y] != '1'){
            break;
        }
        dir8_value[i].first++;
    }
    x -= state.directions8[i][0];
    y -= state.directions8[i][1];
    for(int j = 0; j < 8; j++){
        if(state.user_state[x + state.directions8[j][0]][y + state.directions8[j][1]] == '1'){
            dir8_value[i].second++;
        }
    }
}
```

After we get the value, we will filter those extreme values. If the distance is greater than 8, I will assign a large value, and if the target position only has one or two possible directions, I will assign a small value. With this thresholding calculation, we can trade off between the distance and future of the target point while filtering undesired results.



```
int max_value = -10000000;
int out_dir = -1;
int out_dir_8 = -1;
for(int i = 0; i < 8; i++){
    if (dir8_value[i].first == -1){
        continue;
    }
    if (dir8_value[i].second < 3){
        dir8_value[i].second = -50;
    }
    if(dir8_value[i].first > 8){
        dir8_value[i].first = 50;
    }
    if(dir8_value[i].second - dir8_value[i].first > max_value){
        max_value = dir8_value[i].second - dir8_value[i].first;
        out_dir_8 = (dir8_value[i].second < 0)? 2 : dir8_value[i].second;
        out_dir = i;
    }
}
```

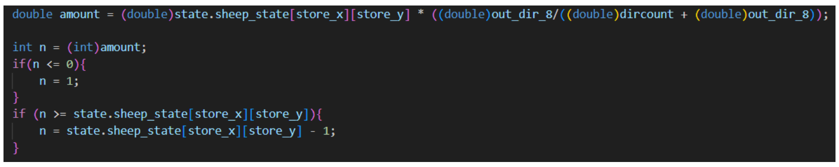
- **Split number**

For split number of sheeps, the calculate formula is

$$split\ number\ =\ sheeps\ number\ of\ decide\ (x,y)\ \times\ \frac{out\ dir}{current\ dir\ +\ out\ dir}$$

(out_dir stands for number of valid split direction after splitting, current_dir stands for number of valid split direction of decided (x,y))

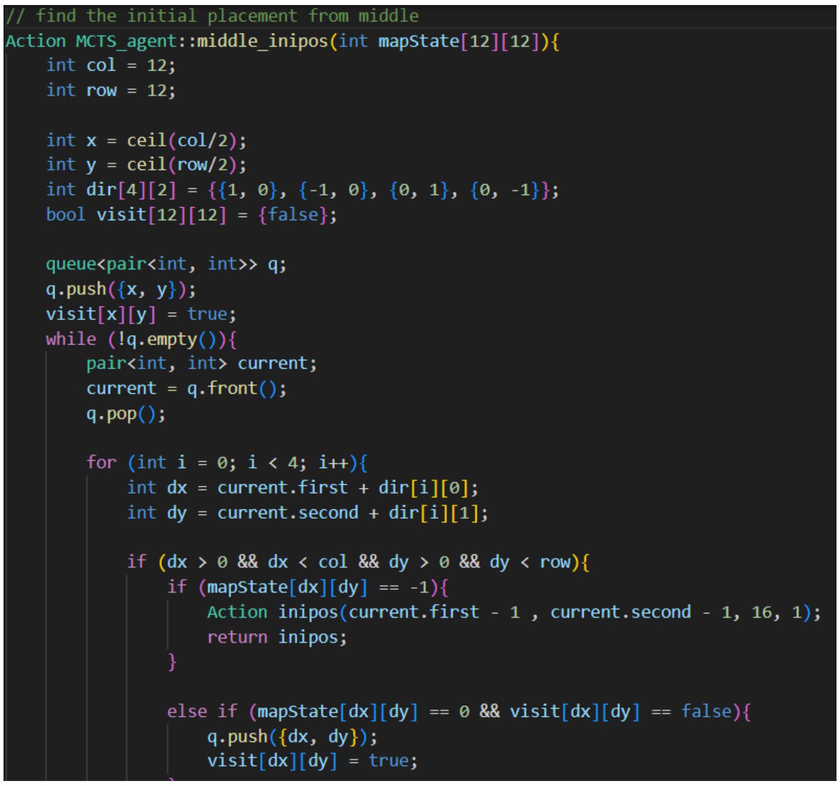
The next 2 if is to avoid illegal move.



```
double amount = (double)state.sheep_state[store_x][store_y] * ((double)out_dir_8/((double)dircount + (double)out_dir_8));

int n = (int)amount;
if(n <= 0){
    n = 1;
}
if (n >= state.sheep_state[store_x][store_y]){
    n = state.sheep_state[store_x][store_y] - 1;
}
```

We also try a rule base method for deciding initial placement, and its concept is quite simple. We simply pick the valid place nearest the middle of the map because it may have more split direction in the middle of the map.



```
// find the initial placement from middle
Action MCTS_agent::middle_inipos(int mapState[12][12]){
    int col = 12;
    int row = 12;

    int x = ceil(col/2);
    int y = ceil(row/2);
    int dir[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    bool visit[12][12] = {false};

    queue<pair<int, int>> q;
    q.push({x, y});
    visit[x][y] = true;
    while (!q.empty()){
        pair<int, int> current;
        current = q.front();
        q.pop();

        for (int i = 0; i < 4; i++){
            int dx = current.first + dir[i][0];
            int dy = current.second + dir[i][1];

            if (dx > 0 && dx < col && dy > 0 && dy < row){
                if (mapState[dx][dy] == -1){
                    Action inipos(current.first - 1 , current.second - 1, 16, 1);
                    return inipos;
                }

                else if (mapState[dx][dy] == 0 && visit[dx][dy] == false){
                    q.push({dx, dy});
                    visit[dx][dy] = true;
```

Similarity and difference in different game

Method 1 - MCTS

Game 1 - normal setting

Game 2 - modify map size and sheep number

Game 3 - fill in the sheep value uniformly

Game 4 - normal setting

Method 2 - MCTS + rule based

Game 1 - normal setting

Game 2 - modify map size and sheep number

Game 3 - normal setting (we don’t need the info. of the opponents)

Game 4 - normal setting

Experiment

Experiment 1 - MCTS vs. rule based

In this part we experiment with 4 conditions. We test different combinations of methods on Initpos and Getstep. Here is the experiment result. We run this experiment on game 1 10 times for each combination to get the win rate. Note that the MCTS iteration setting is 50.

Initpos / Getstep method	rule base / rule base	rule base / MCTS	MCTS / rule base	MCTS / MCTS
win rate (win/total)	3 / 10	2 / 10	4 / 10	3 / 10

As in the above table, we can see that if we take the rule based method for the initpos, the result is not quite good (win rate is 20% & 10%). The cause of the result might be having a valid position nearest to the middle is not suitable for every game. Maybe some games have more barriers in the middle of the map, causing the split direction of the sheep to get blocked. As for the Getstep, the rule based method achieves the best result (win rate is 40% & 30%). Since we have a limitation on computation time, we can not have a MCTS iteration number higher than 40 (this might be caused by my laptop and windows, I checked the task manager in my laptop, and the program only use 3% of CPU, and 70% was used by the windows anti-malware, the program can not have more CPU even when i turn off the defender, so we can have high value on MCTS iterations). Therefore, the win rate of MCTS is not good as a rule base. The whole rollout result is large (a lot of steps), having only 40 iterations is not enough, hence a bad result is expected, but we believe it will get better results if we can have more iterations.

Experiment 2 - Different action space in MCTS

Here we do experiment on the number of sheeps moved

1. M - 1 actions per point (sheep number from 1 to M-1)
This is the full action space, which is quite large and hard to explore.

2. 1 action per point (sheep number = M/2)
This significantly reduced the action space but the flexibility is low.

3. 3 actions per point (sheep number = 1, M/2, M-1)
This trades off between flexibility and action space. The agent can now jump away if there is only one path left, and fill in the empty value if there is only one direction at the target position and the target position is next to the source point.

action space	full action	1 action	3 actions
win rate (win/total)	0 / 10	1 / 10	3 / 10

As in the above table, we can see that the win rate from high to low are 3 actions, full actions, 1 action respectively. The cause of the result might be that the “3 actions” policy has less action space and more flexibility, hence it gets the best result. As for the “full action”, it needs more iterations to explore all possible actions, but since we only set 50 iterations for MCTS due to computation time, therefore it can’t get the best performance. And the “1 action” due to the fixed split number, it can’t adapt to the current game state to make different actions. So, it’s quite intuitive that “1 action” gets the worse performance compared to other policies.

Contribution

109350008 張詠哲: 50%

MCTS, initial placement rule base, report

110705013 沈昱宏: 50%

MCTS, decide step rule base, report