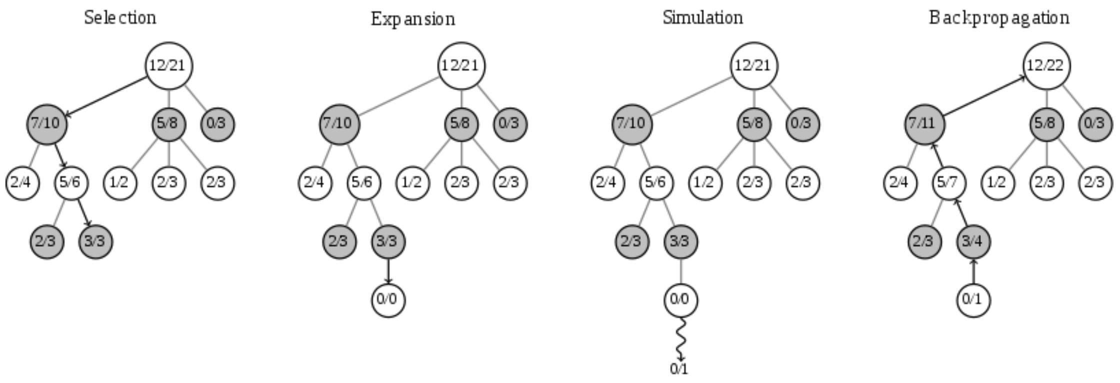
Report

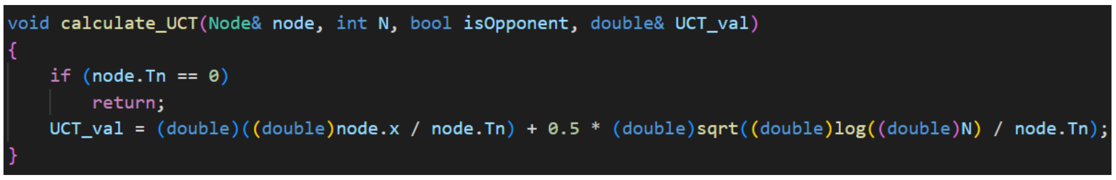
How my game AI works

For my AI model, I use Monte Carlo Tree Search as the algorithm. There are 4 phases in MCTS.



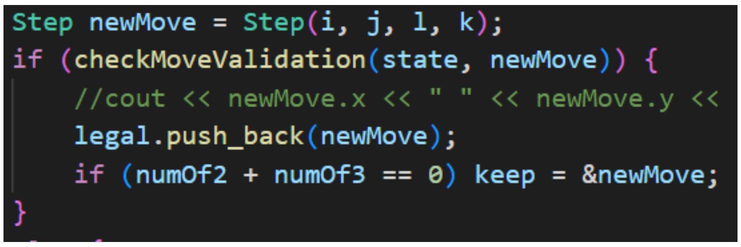
To begin with, MCTS will do selection. It will go through from the root node to the leaf node. To elaborate, MCTS will choose which node to go next depending on the value calculated by UCT formula. The following picture is the formula of UCT. The left term is about exploitation, which is the win rate, while the right term is exploration, which is calculated by the square root of the log of the visited time of the parent node being divided by the visited time of the current node. Finally, c is the exploration parameter, which can determine the proportion of the exploitation term and exploration term. In my case, I set it to 0.5. Moreover, when the node hasn't been visited, I will select that node directly.

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



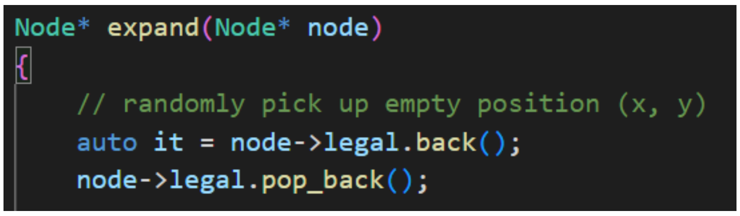
```
void calculate_UCT(Node& node, int N, bool isOpponent, double& UCT_val)
{
    if (node.Tn == 0)
        return;
    UCT_val = (double)((double)node.x / node.Tn) + 0.5 * (double)sqrt((double)log((double)N) / node.Tn);
}
```

Second of all, the next phase is expansion. When it arrives at the leaf node, I will randomly choose an available node for the current board state. For me, to save time, instead of going through the current board to find the available next step in every expansion phase, I will collect and save all available next steps in a vector when first visiting that node so that I can spend less time looping through the current board.



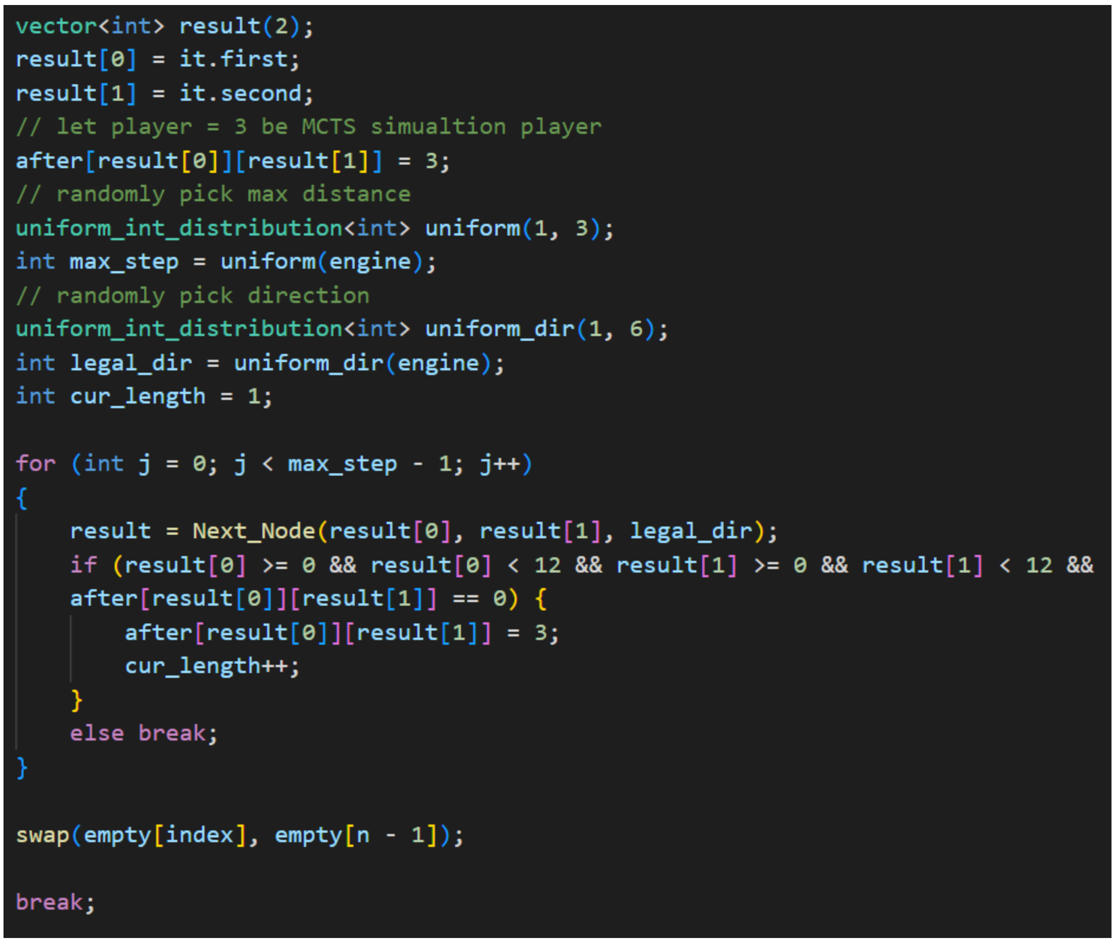
```cpp
Step newMove = Step(i, j, l, k);
if (checkMoveValidation(state, newMove)) {
    //cout << newMove.x << " " << newMove.y <<
    legal.push_back(newMove);
    if (numOf2 + numOf3 == 0) keep = &newMove;
}
```

Then, when expansion, just pop out from the vector keeping available next step.



```cpp
Node* expand(Node* node)
{
    // randomly pick up empty position (x, y)
    auto it = node->legal.back();
    node->legal.pop_back();
```

As for simulation, to save time, I will also pick up available positions in the current board initially and then randomly pick up direction and length. Moreover, I will swap the content with the last element so that there is still an empty position at the current index.



```cpp
vector<int> result(2);
result[0] = it.first;
result[1] = it.second;
// let player = 3 be MCTS simualtion player
after[result[0]][result[1]] = 3;
// randomly pick max distance
uniform_int_distribution<int> uniform(1, 3);
int max_step = uniform(engine);
// randomly pick direction
uniform_int_distribution<int> uniform_dir(1, 6);
int legal_dir = uniform_dir(engine);
int cur_length = 1;

for (int j = 0; j < max_step - 1; j++)
{
    result = Next_Node(result[0], result[1], legal_dir);
    if (result[0] >= 0 && result[0] < 12 && result[1] >= 0 && result[1] < 12 &&
    after[result[0]][result[1]] == 0) {
        after[result[0]][result[1]] = 3;
        cur_length++;
    }
    else break;
}

swap(empty[index], empty[n - 1]);

break;
```

When the randomly pick up position is occupied since the previous move, I will swap it to where the first available position is. Therefore, when i >= n, I will break up and observe whether it is my turn. If it is my turn, then return 0, which means I lose in this simulation. Otherwise, return 1 since I win this turn.

```
// randomly pick up an empty space
std::uniform_int_distribution<int> uniform(i, n - 1);
int index = uniform(engine);
auto it = empty[index];

if (after[it.first][it.second] != 0) {
    // maybe be occupied by 2 or 3 step
    swap(empty[index], empty[i]);
    i++;
}
```

Finally, do backpropagation by adding the simulation result and the visited time.

```
void backpropagate(Node* node, int result)
{
    node->Tn++;
    node->x += result;
}
```

In addition, I will run all 4 steps of MCTS as many times as possible (depending on the time limit).

```
while (1)
{
    traverse(root);
    end = clock();

    if (((double)(end - start)) / CLOCKS_PER_SEC >= time_limit)
        break;
}
```

Then, choose the step with the maximal visited time as the next step to place.

```
int max_count = -1;
Node* bestChild = nullptr;
for (Node* child : root->children) {
    int Tn = child->Tn;
    if (max_count < Tn) {
        max_count = Tn;
        bestChild = child;
    }
}

step[0] = bestChild->parent_move.x;
step[1] = bestChild->parent_move.y;
step[2] = bestChild->parent_move.numOfStep;
step[3] = bestChild->parent_move.dir;
```

Experiments

In my experiment, I will test the performance between different time limits and exploration parameters. Additionally, I will play with sample_2.exe that TAs gave to us. In the light of time limit, it makes sense that the larger the time it can think about each step, the higher the win rate is. Therefore, I set the time each step can think at 5.9s.

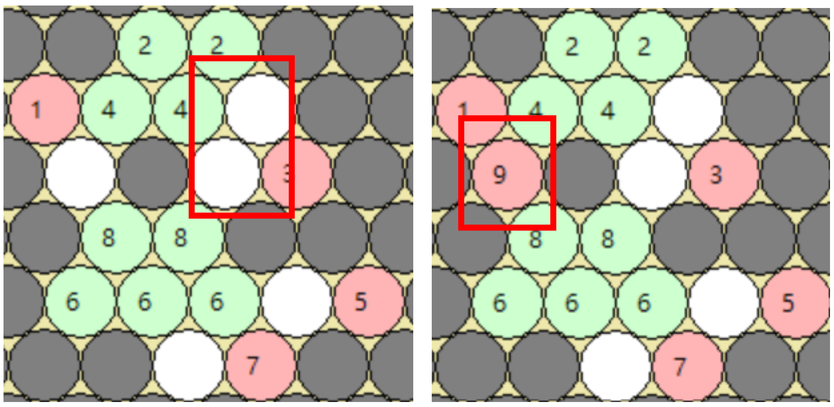
Time limit	Win rate(%)
2	73.3
4	82.0
5.9	86.6

In terms of the exploration parameter, I initially set it to the theoretical value, while it performs worse so I let it be 0.5 and it turns out to be much better. In my opinion, sqrt(2) is so large for this game that it will let the agent explore new nodes too much.

c	Win rate
0.5	86.6
sqrt(2)	66.66

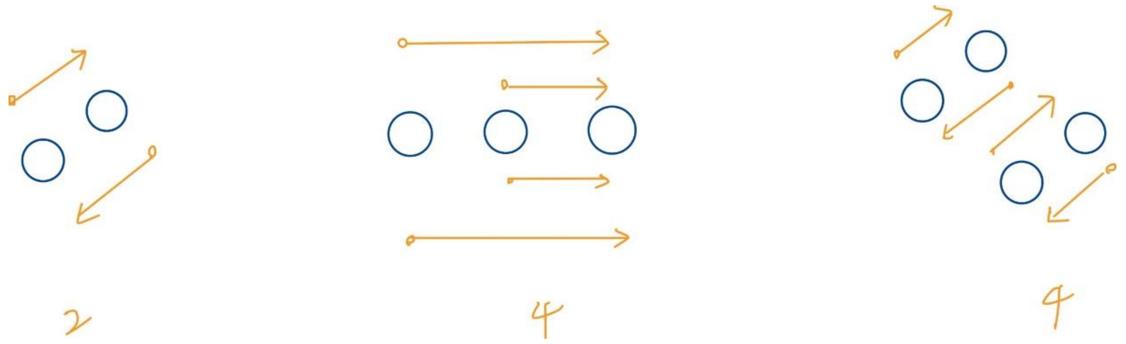
Experiences

I found that my AI model sometimes will place very stupid steps, such as in the following case. If my agent takes all the consecutive 2 steps, it can win the game. Nonetheless, it chooses another step, so it loses the game.



Therefore, I decide to use the knowledge of Endgame. When there is only 1 remaining step that is either 2 or 3 consecutive ones and others are all 1-length steps. I won't run MCTS. Instead, I will use endgame knowledge. If there are an odd number of remaining 1-steps after placing the current step, I win the game. As a result, when meeting the 2-length of steps remaining and the number of all the other steps is odd, I will cross out the whole step. On the other hand, if the number of all the other steps is even, I will only cross out one of a circle of the 2-length step. As for the 3-length step, if the number of remaining 1-length steps is odd, I will cross out

the whole 3-length step. Otherwise, I will cross out the 2 of the 3-length step. What's more, if there remaining 2 of the 2-length steps, I will let my agent cross out the other 1-length steps so that I can avoid being passive.
To do the mentioned above, I will calculate the number of 2-length steps and 3-length steps. If the sum of the two is 2 or 4 when the node is the root, I will use endgame knowledge to decide the next step. The reason for 2 and 4 is shown below.



To be more specific, if the sum is 4, which means there may be one 3-length step remaining or two 2-length steps remaining, I need to go further to judge which case it is by observing whether the length of the step I keep is 2.

Challenge

In this homework, I encounter various problems. I spent a lot of time debugging and enhancing the performance. For example, I initially find the return step will always be wrong at the very beginning. It is because when I do expansion I will keep the parent moving by getting the content from the pointer pointing to the vector keeping the children's back. Hence, next time the back of the vector won't point to the information I need, so it will get wrong. To solve this problem, I just directly point to the move I want. Another challenge is that the performance of my AI model is poor. To cope with this problem, I try many methods, such as RAVE. It is a method that can enhance performance by recording the result and visit time in advance. Unfortunately, there must be something wrong that the performance didn't become better. What's better, I also try to apply endgame knowledge as above mentioned. Fortunately, it does improve performance. Last but not least, I bump into illegal moves many times. Although it seems a trivial problem, it took me lots of time to find the bug. In conclusion, I met so many problems in this project. Although it made me go crazy, I now know MCTS much better than before.