CS303 Project-1 Report

Jiajian Yang
Southern University of Science and Technology
12012711@mail.sustech.edu.cn

1. Introduction

1.1. Background

Othello is an interesting game being popular in some western countries. The game is played by flipping each other's pieces and the winner is determined by who has the most pieces on the board. It is easy to begin a game because the rules of the game are simple, yet it is full of strategies and wisdom. There is a saying that "a minute to learn... a lifetime to master"[1].

In recent years, artificial intelligence(Ai) has dramatically developed. Many classic Algorithms were proposed (e.g., minimax search, alphabeta search) and the technology in machine learning is also used to train Ai model. Hence, with the interests in Othello, and the purpose of having a further acknowledge of Ai technology, in this course project, we decided to implement a simple Ai for Othello and note the process in the report as a review of exploration.

1.2. Problem Formalization

Unlike normal Othello, the winning condition in this project is that the player with the fewest pieces wins at the end. Let P, O be the player and opposite player respectively and $\mathcal{A}_P, \mathcal{A}_O$ represents the actions for each player. The utility function r will assess each move for a player and return a score. The goal of the game is to maximize the opponent's score.

$$max\{r(a), a \in \mathcal{A}_O\}$$

2. Methodology

2.1. General workflow

Step1: first, we need to find all the actions that we can move according to the current game board.

Step2: Then for each piece in actions, we perform alphabeta search to get an score of each move, and pick up the maximum value move.

2.2. Details & bright spots

2.2.1. Board Dictionary. It is a Board class extending DefaultDict in Python3 for convenient management of the chessboard, and improve the efficiency of the program. The class contains member variables, including chessboard_size, to_move and utility. 'To_move' indicates who should play on the current board, and 'utility' indicates the current player's assessment of the current situation.

2.2.2. Find Player Actions. Considering how to find all the actions, we designed an effective algorithm. We first find all the opponent pieces, and for each opponent piece, if there is a blank being adjacent to it, then we search for its reverse direction. If we meet a piece with the same color of ours, the blank is a valid action(**See Algorithm 1**).

2.2.3. Alpha-Beta Search. As for Alpha-Beta Search based on Minimax search, which is an algorithm that finds the minimum value of the maximum probability of failure. That is, it is assumed that each move by the opponent will lead us in the direction of the least valuable situation from the current point of view. Thus, our strategy is to choose the best of the worst-case scenarios that the opponent can achieve, which means to minimise the damage that the opponent would inflict on us if he were to make a perfect decision.

However, the efficiency of Minimax is refered as $O(b^d)$, where b is the number of legal actions and d is the max depth of DFS. As for Othello, b's max value can be more than 17, and if we search for 5 layer, there are $17^5 = 1,419,857$ nodes. It will take computer a lot of time to calculate. As a result, we introduce Alpha-Beta pruning, which can reduce many useless calculations in Minimax search by $O(b^{m/2})$(**See Algorithm 2, 3**).

min_value has the similar structure of max_value

2.2.4. Evaluation Function. For the Ai, the evaluation function can be the most important part. The evaluation function of this project mainly considers four factors, including the weight of the chessboard, the opponent's mobility, the number of pieces flipped each time and the stable pieces. We divide the whole chess game into four stages: early stage, middle stage, late stage and final stage. In the early stage, there were 0 to 15 pieces. Because there were few pieces on the chessboard, there was no need to think too much, so more consideration was given to the number of pieces flipped each time and the weight of the chessboard. In the middle stage, there are 16 to 35 pieces. At this time, the middle position is basically occupied, so the position of pieces probably appears on the four edges of the chessboard. According to experience, the position of

Algorithm 1: find_actions

```
Input: opponent list, chessboard, player
Output: action list
action ← [];
player ← board.to_move;
directions ← ([1, 0], [−1, 0], [0, 1], [0, −1],
[−1, −1], [1, 1], [−1, 1], [1, −1]);
for i in opo do
    neighbor ← [];
    for dir in direction do
        temp ← (i[0] + d[0], i[1] + d[1]);
        if temp is valid and unoccupied then
            neighbor.append(temp))
        end
    end
    for disc in neighbor do
        x ← i[0], y ← i[1];
        dir ← (x − disc[0], y − disc[1]);
        while True do
            x ← x + dir[0], y ← y + dir[1];
            if (x,y) is valid then
                if chessboard[x,y] == player then
                    action.append(disc);
                    break;
                end
                else if chessboard[x,y] is blank then
                    break
                end
                else
                    continue
                end
            end
            else
                break
            end
        end
    end
end
;
```

Algorithm 2: alphaBeta_search

```
Input: self,board
Output: optimised move
player ← board.to_move, deep ← 0;
candidate ←
  max_value(board, deep, −infinity, +infinity);
if candidate != None then
    self.candidate_list.append(candidate)
end
```

Algorithm 3: max_value

```
Input: board, deep, alpha, beta
Output: value, move
deep ← deep + 1;
if deep == SETDEEP then
    return board.utility, None;
end
if is_terminal(board) then
    return board.utility, None;
end
v, move ← infinity, None;
for a in self.action(board) do
    v2, _ ←
      min_value(result(board, a), deep, alpha, beta);
    if v2 >v then
        v, move = v2, a
    end
    ;
    if v ≥ beta then
        break
    end
    ;
    if v >alpha then
        alpha = v
    end
end
```

the four sides is very unfavorable, so we lowerd the weight of the chessboard on the four edges, and chessboard weight takes more proportion in the mid-term, so that Ai can occupy the edge as little as possible. In the later stage, there are 36 to 50 pieces. At this time, the space left on the chessboard is insufficient, so we should try to reduce our own space to increase the opponent's mobility, so that the opponent can occupy more pieces. There are 51 to 60 pieces in the final stage. At this time, the weight of the chessboard has become useless, but the opponent's mobility is more important. So we increased the proportion of the opponent's mobility. In addition, in order to occupy the eight powerful points (1, 0), (0, 6), (7, 1), (6, 7), (0, 1), (6, 0), (7, 6), (1, 7) in the search process, as long as one of the eight points is found, you will immediately return and perform the chess operation on these position.

3. Experiments

3.1. Hardware and Software

CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30GHz
GPU: NVIDIA GeForce MX350
Python: 3.9.12
pandas: 1.4.3
matplotlib: 3.5.3
Numpy: 1.23.2

4. Datasets

We use the game data downloaded from the platform http://172.18.34.89:8080/ and use them to do the test.

4.1. Analysis

We implement two methods and test for its efficiency respectively.

First, we compared the running time of Minimax search and AlphaBeta search with the search depth:6.

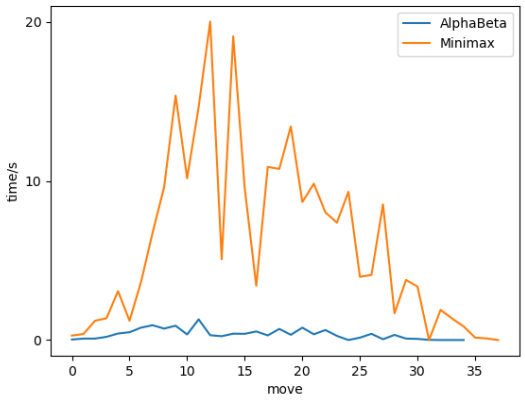

Figure 1. The running time of each move in a game

Figure 1 demonstrates that the running time of Minimax search is dramatically longer than that of AlphaBeta search. The average running time of Minimax is about $6.13s$, while AlphaBeta has only $0.36s$ when there are 6 searching layers. The longest time taken by Minimax happening in the 13^{th} move, which is 20.03s, but the longest time cost by AlphaBeta is $1.3s$ when it comes to the 12^th move. In conclusion, the speed has improved by 15.6 times.

Then, we test the running time of AlphaBeta search in different depth condition.

Figure 2 demonstrates that the average running time increases as the depth raise. Except for certain moves, which runs for a long time, the other moves are keep in a stable running cost. And the project requires no more than 5 seconds, so we finally decide to choose AlphaBeta search with depth=7.

5. Conclusion

5.1. Problem

In the process of completing the project, I did encounter many problems. There were problems with python programming, such as how to write code elegantly. How to cleverly design data structures that make the program easy to manage and read. I would also have to consider how to optimise my Ai, as no machine learning is used, so we have to review the game after each match and manually adjust the

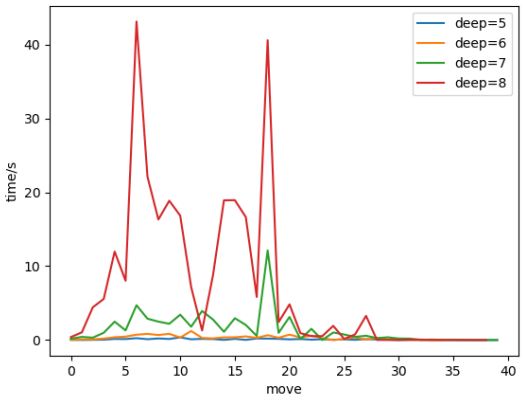

Figure 2. The running time of different depth in AlphaBeta

parameters of the evaluation function. This would be a huge undertaking. However, the result is relatively good, as the highest rank of my Ai on the open platform is 145.

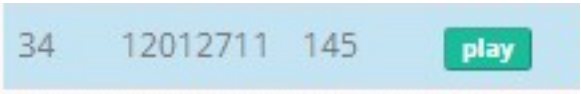

Figure 3. The highest rank on the platform

References

[1] Othello: A Minute to Learn... A Lifetime to Master. Gabriel. January 1, 1978

[2] Minimax principle. Encyclopedia of Mathematics. EMS Press. 2001 [1994]