AI Algorithm for Reversi

Shangxuan Wu 11811535
Department of Computer Science and Engineering
Southern University of Science and Technology
11811535@mail.sustech.edu.cn

1 Preliminaries

In this part, I will introduce my development environment, algorithms used and application of Reversi.

1.1 Problem Description

Reversi is a strategy board game for two players, played on an 8×8 checkered board.[1] In this project, I focus on development of suitable and powerful Reversi algorithms.

All codes of my project are written in *Python* of version 3.7 with *PyCharm*. The main testing platform is *Windows 10 Professional Edition* (version 1909) with Intel$^{@}$Core^{TM}i7-6700 CPU @ 3.40GHz with 4 cores and 8 threads.

Algorithm used in this project is minimax algorithm with alpha-beta pruning. Minimax algorithm is a decision rule used in artificial intelligence, decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. When dealing with gains, it is referred to as "maximin"—to maximize the minimum gain. [2] Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.[3]

1.2 Problem Application

It can be used to develop Reversi games.

2 Methodology

In this part, I will introduce the concepts in this report, structure of my program and detailed algorithms.

2.1 Notation

All my notations in class diagram (Figure 2) are listed as follows:

1. List< Tuple >: list contains points with tuple type
2. Ndarray: chessboard with NumPy Ndarray type

2.2 Data Structure

All my key data structures are listed as follows:

1. point: a tuple contains x coordinate and y coordinate
2. chessboard: a NumPy Ndarray represents for chessboard
3. candidate_list: a list of feasible points which have found. Last tuple in candidate_list is the most valuable position.
4. corner_list: a list of points recording position of corners in chessboard

2.3 Model Design

The flow chart of whole program are shown as below:

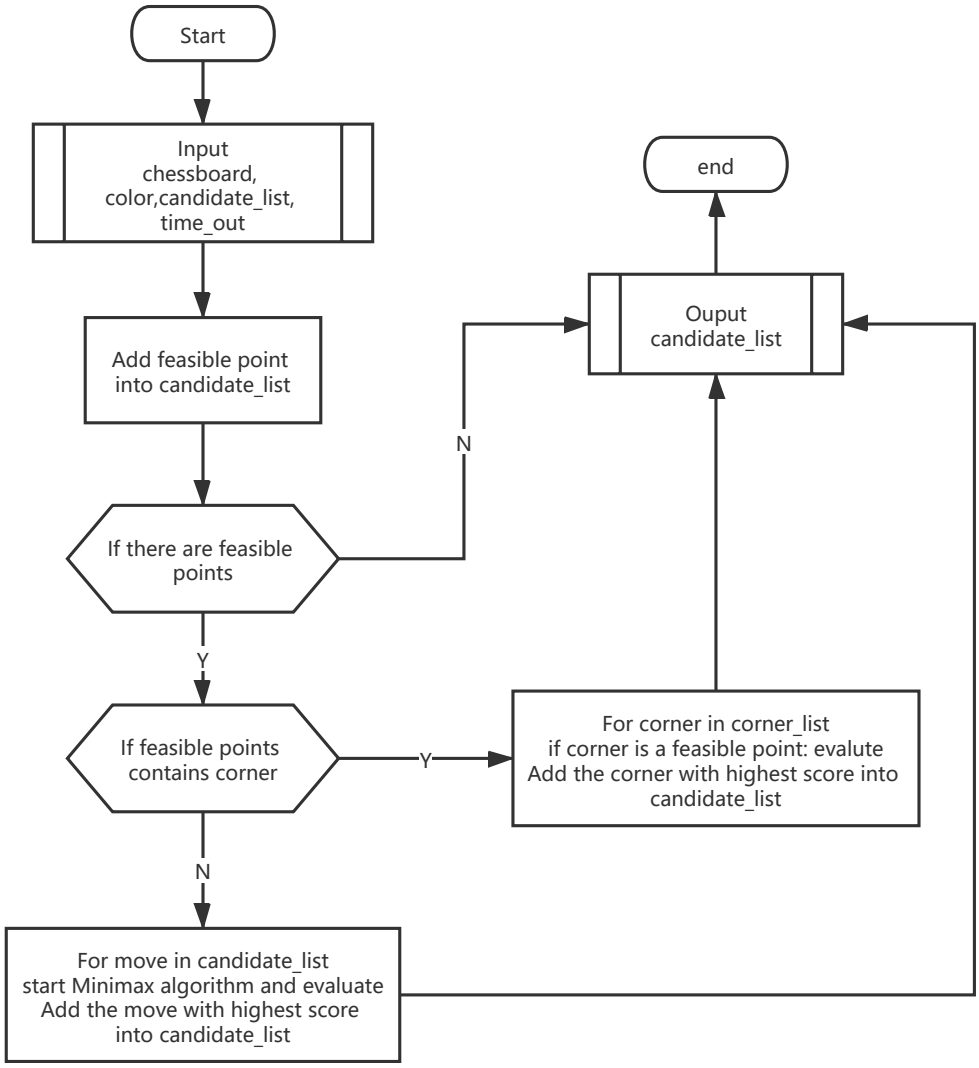

Figure 1 The flow chart of the whole program

The class diagram of whole program are shown as below:

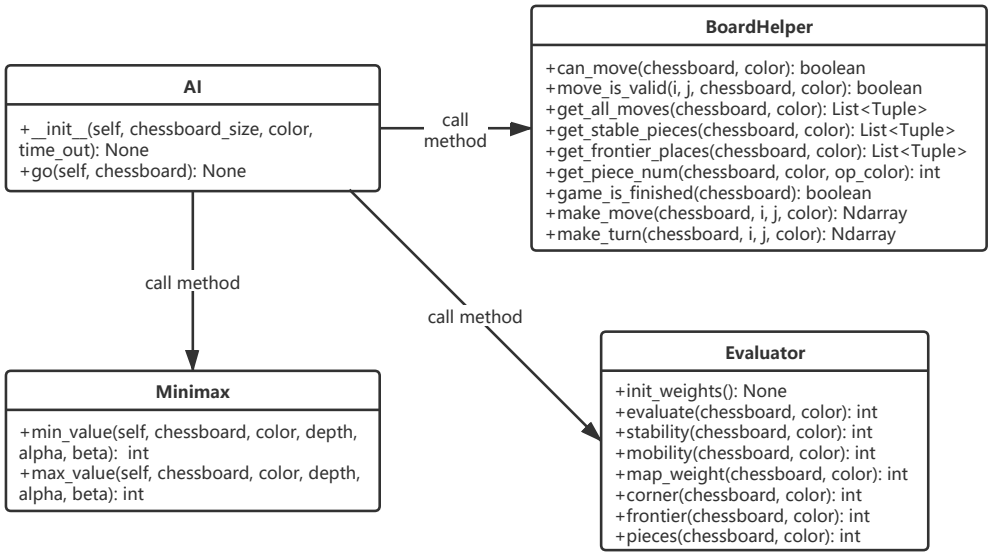

Figure 2 The class diagram of the whole program

2.4 Detail of Algorithm

In this project, I implemented minimax algorithm with alpha-beta pruning. The entire algorithm contains three functions: `minimax(self, chessboard, color)`, `min_value(self, chessboard, color, depth, alpha, beta)`, and `max_value(self, chessboard, color, depth, alpha, beta)`. The first function checks best score and defines best move, the second one implements logic at min nodes, while the third one implements logic at max node. The first function is added into `go(self, chessboard)` in AI class, and the last two functions can be seen in Minimax class in Figure 2.

Considering the unstableness of the evaluation machine, a while loop (i.e. an additional feature) is added into the first function, which is designed by my classmate Zhengxin You. A new minimax query will start with depth plus one if there is still time and last minimax query has ended. Besides, logic to limit timeout is also added into the second and the third functions.

Fake codes are shown below:

```
minimax(self, chessboard, color)
while True:
    if timeout:
        break
    find best move, add it into candidate_list
    depth += 1
```

```
min_value(self, chessboard, color, depth, alpha, beta)
if timeout or depth <=0 or game_finished(chessboard):
    return evaluate(chessboard, color)
call max_value(self, chessboard, color, depth, alpha, beta)
find best_score
if best_score <= alpha:
    return best_score
beta = min(beta, best_score)
```

```
max_value(self, chessboard, color, depth, alpha, beta)
if timeout or depth <=0 or game_finished(chessboard):
    return evaluate(chessboard, color)
call min_value(self, chessboard, color, depth, alpha, beta)
find best_score
if best_score >= alpha:
    return best_score
alpha = max(alpha, best_score)
```

3 Empirical Verification

3.1 Dataset

3.1.1 Test Dataset for finding bugs

Before the points race, I used two methods to test my code: offered usability test and test cases generated by my self. I randomly assigned values to the board, then gave it to my code. If the output not matched my perception, I would find bugs with my program and fix it. During the points race, I obtained more ways to find bugs. I used "autoplay" or "playto" function every once in a while to look for problems and check strength of my algorithm at the same time. If the system told me I returned invalid feasible point, I would download chesslog and find bugs.

3.1.2 Test Opponent help improve code

During the points race, I not only relied on the Reversi platform to allocate an opponent. Fortunately, classmate Zhengxin You wrote a local battle platform using *Python* and shared his code with us. Through that, we can play to each other even without "playto" function in the Reversi platform and the disability of it have little impact on me.

Based on official platform and local platform, my code promotion process consists of the following three steps:

1. Fight against the old code as soon as possible to determine the strength of the algorithm
2. Fight against my classmates, then exchange experience and experience
3. If the strength of the code does improve, submit it to the platform and use "playto" and "aotoplay" functions

3.2 Performance Measure

As mentioned above, I use both official platform and local platform to measure the performance. Besides, discussing with classmates also helped me a lot. Time and strength of evaluation function are measured.

3.2.1 Time Measure

At the beginning of the points race, I did not adjust the search depth based on the remaining time and only search 4 floors at a time. With the increase in the number of simultaneous matches and the complexity of the evaluation function, I found my search timeouts were increasing. I tried to solve this problem and found that after reaching a certain depth, the best feasible point searched out in most cases would not change. That is why I added a while loop outside minimax algorithm. The timeout situation had been greatly alleviated after it, which was showed by my rising ranking and chess logs.

3.2.2 Evaluation Function Measure

Evaluation function is the most important part I focused on. I measure it by fighting with as many classmates as possible and check competition result for winning one person does not mean the code strength has increased. Meanwhile, I discussed with my classmates to find more kinds of evaluation functions and it did worked.

3.3 Hyper Parameters

My evaluation function is shown below:

```
evaluate(chessboard, color)
if piece_num <= 50:
    score = map_weight + 10 * stability + 30 * mobility
else:
    score = map_weight + 10 * stability + 30 * pieces
```

```
weight = np.array([[500, -25, 10, 5, 5, 10, -25, 500],
                   [-25, -45, -1, -1, -1, -1, -45, -25],
                   [10, -1, 3, 2, 2, 3, -1, 10],
                   [5, -1, 2, 1, 1, 2, -1, 5],
                   [5, -1, 2, 1, 1, 2, -1, 5],
                   [10, -1, 3, 2, 2, 3, -1, 10],
                   [-25, -45, -1, -1, -1, -1, -45, -25],
                   [500, -25, 10, 5, 5, 10, -25, 500]])
```

`weight`: weight table of board position

`map_weight = sum(weight * color or op_color) * color`

`stability`: amount of my stable pieces

`mobility`: amount of feasible pieces

`pieces`: amount of my pieces

The fight experience tells me that mobility is the most important in the middle game while amount of my pieces is the most important in the late game. Amount of stable pieces is also important, but giving it too high weight may result in focusing on stable pieces and ignoring total amount of pieces. Moreover, a function that prevent AI chooses bad location is necessary. I give negative scores to positions surrounding the corner, which can be seen in Ndarray `weight`.

It is clear that once the corner is occupied, it will not be flipped. To help my AI get corners, I gave highest scores to them and made a special judgment before going into minimax algorithm in `go(self, chessboard)`. The fake codes are shown as follows:

```
handle_corner(candidate_list, chessboard, color)
for each corner:
    if corner in candidate_list:
        evaluate(chessboard, color)
choose the corner with highest score
```

3.4 Experimental Results

I passed all test cases in the usability test.

My rank in the point race is 49.

My rank in the round robin is 56.

3.5 Conclusion

To be honest, I am satisfied with the experimental result I got because in this project I got familiar with algorithm writing and parameter tuning techniques and tempered my mentality. I used different evaluation functions in different stages of the game and parameters in evaluation function were adjusted many times, they are advantages of my program. Of course, there are still many imperfections in my code. I have implemented different evaluation methods in my code, but only some of them are actually used because the parameter adjustment is too troublesome. In addition, I found the opening library and added it to my code, but I didn't use them in the end. I will use them to improve my evaluation function later.

4 References

[1] En.wikipedia.org. 2020. *Reversi*. [online] Available at: https://en.wikipedia.org/wiki/Reversi.

[2] En.wikipedia.org. 2020. *Minimax*. [online] Available at: https://en.wikipedia.org/wiki/Minimax.

[3] En.wikipedia.org. 2020. *Alpha–Beta Pruning*. [online] Available at: https://en.wikipedia.org/wiki/Alpha-beta_pruning.