CS303 Artificial Intelligence Project - Reversi Intelligent Program

Shuaihan Cheng ID 11811811
Computer Science and Technology
Southern University of Science and Technology
11811811@mail.sustech.edu.cn

1. Preliminaries

Reversi is a strategy board game for two players, played on an 8×8 uncheckered board, invented in 1883. *Othello*, a variant with a change to the board's initial setup, was pattened in 1971. In a Othello games, sixty-four identical game chess pieces are called disks or discs, which alternately are white and the other side is black. Players take turns placing disks on the board, with their assigned colors facing upwards. During a turn, any disc that is in line with the opponent's color and limited by the disc just placed, and another disc of the current player's color will be converted to the current player's color. The purpose of the game is to make most of the disks rotate to show color when the last playable blank square is filled.

From early 1950s, people were trying to realize auto-playing chess game with computer in a intelligent way. Nowadays, from super computers to smart phones, computer chess applications that reach the level of a chess master or higher can be used, a standalone chess game machine can also be used. This project can provides an AI program for auto-playing Reversi game on PC platform or mobile platform.

In this project, I was provided a Reversi online platform realized an intelligent program to play chess game like human expert. It works better than even skillful Reversi human player.

1.1. Software & Hardware

This project is written in Python with python IDE Pycharm Professional.

The main testing platform is Windows 10 Enterprise Edition (version 10.0.19041) with CPU AMD Ryzen 7 4800HS with Radeon Graphics @2.9GHz 4.2GHz with 8 cores and 16 threads. /notation /notation

1.2. Algorithms

Minimax search algorithm, a decision algorithm often used in artificial intelligence and chess game to minimize the possible loss for a possibly worst case scenario, was picked-up in my project.

For the purpose of optimize the running efficiency of my program, **alpha-beta pruning algorithm**, a search algorithm always used in Minimax algorithm seeking to decrease the number of nodes was also selected in my program.

In order to evaluation the condition of a given chessboard, an **evaluation algorithm** considering disks number, stable disks, chess position scores ,mobility and other evaluation factors was used in my program.

2. Methodology

2.1. Representation

2.1.1. Notation.

- **Chessboard**: equivalent the chessboard in the Data Structure section.
- **CanList**: equivalent the candidate_list in the Data Structure section.
- **Player**: An integer. The value is exactly 1 or -1, corresponding to black player or white player.
- **Cal_List(Chessboard, Player)**: The function calculating the CanList with Chessboard and Player as input, return the CanList.
- **Eva(Chessboard, Player)**: The function evaluating the situation of a given Player, an integer value. The positive value represents a good scenario for given player, vice versa.
- **AVB(chessboard,Player)** : Avoid bad list function. This function calculates some simple rules from the board position. These rules will avoid choosing bad moves.
- **evaluation matrix**: equivalent the evaluation_matrix in the Date Structure section.

2.1.2. Data Structure.

- candidate_list: After program execution, it will give out a list(candidate_list) containing all the coordinates of point that can be taken by current player. It's a python list structure with multiples or 1 tuple in it. Each tuple expresses a coordinate of chessboard. The list can also be empty, which means that current player can't get a position to move. The last element of the list is the selected point of current step.

- chessboard: a 8 × 8 matrix. Each element of matrix shows the situation of corresponding chessboard with 1 for black, -1 for white and 0 for uncaptured position. For example, the matrix below

$$\begin{Vmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{Vmatrix}$$

  indicates a standard start scenario of Reversi. The coordinate of top left point is (0,0). The coordinate of top right point is (0,7). The coordinate of bottom right point is (7,0). The coordinate of bottom right point is (7,7).
- evaluation_matrix: a matrix with different integer value. Each value is the score of corresponding position. Higher score means better position. For example, the matrix below

$$\begin{Vmatrix} 500 & -250 & 0 & 0 & 0 & 0 & -250 & 500 \\ -250 & 0 & 0 & 0 & 0 & 0 & 0 & -250 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -250 & 0 & 0 & 0 & 0 & 0 & 0 & -250 \\ 500 & -250 & 0 & 0 & 0 & 0 & -250 & 500 \end{Vmatrix}$$

  is a sample evaluation matrix, is shows that the position in the corner scores most.
- bad_list: the data structure contains some absolutely bad or valueless coordinate positions. The program will try to avoid these positions.

2.2. Architecture

2.2.1. Model design. The running process of the program is as follows.

First, we determine the candidate list based on the input board status and the current player. Then, we made a special rule. Try to avoid that the last element of the candidate list is the element in our bad list. Then, we make a judgment. Determine how many pieces are in the board. If the total number of existing pieces exceeds 58, a minimax search is performed, otherwise, a brute force search is performed. Finally, use the special rule function to process the candidate list we got, and then end the entire program.

2.2.2. Flow Chart of the Whole Program. Figure 1 shows the whole process of my program.

2.2.3. Calculate the candidate list. The function **Eva(Chessboard, Player)** is called and returns the candidate list. This function traverses each piece on the chessboard, and judges whether there is a corresponding

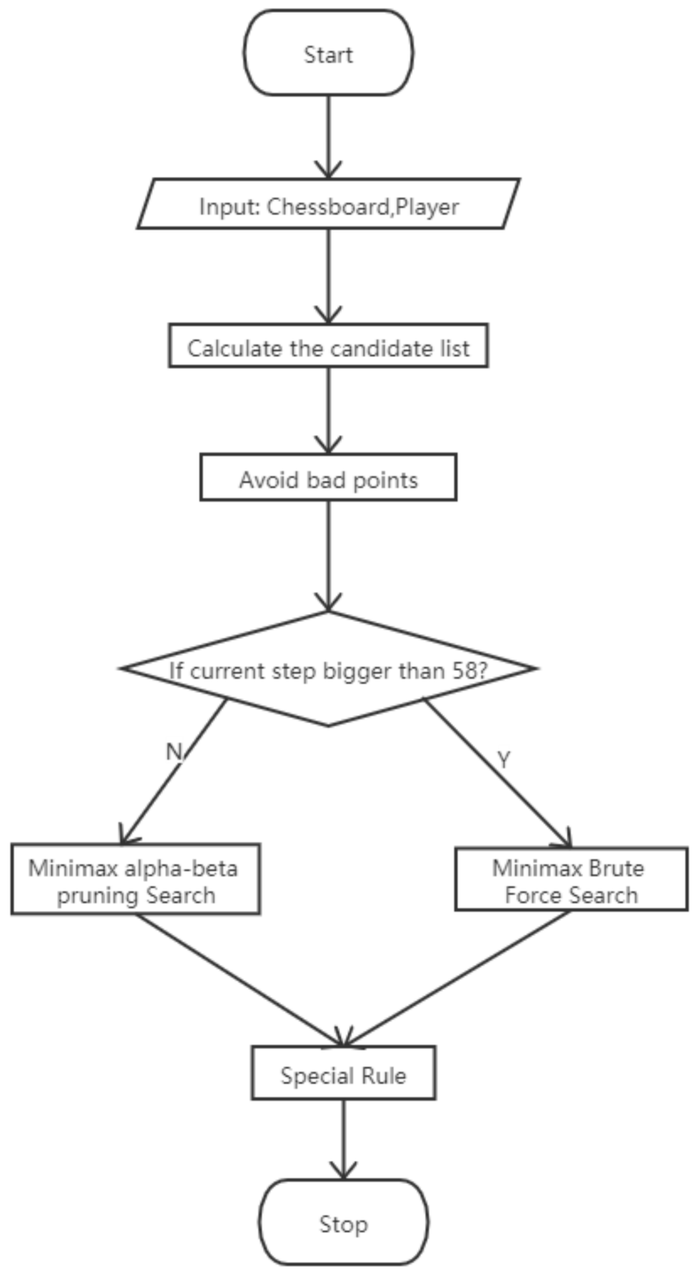


Figure 1. The Flow Chart of the Whole Model

empty and rule-compliant position in the eight directions for each piece.

2.3. Evaluation function

Input the current chessboard and current players into the evaluation function, which will comprehensively consider many factors for scoring.

For example, in the early time of chess games, a stable optical disc(high stability) was considered an important factor.

As the game progresses, a movibility chessboard will be obtained, which means that it is difficult to flip the disc at that point. We calculate a factor called stability on the board to estimate the stability. The higher the stability score, the better the situation is. I got the idea of selection of idea from works of Korf [1].

In addition, the value of the decision matrix (evaluation matrix) is also an important reference point. In the later stages of the game, the real deciding factor is the number of chips on both sides, so in different stages of the game, the evaluation method and the formula of the evaluation function are also different.

The formula of the evaluation function can be writen as:

$$score = x*matrix\_score+y*stable\_score+z*move\_score$$

The x, y, z vary as game processes. I also referred to some article to learn a better way to get my evaluation function. [2]. For example, how to selected a better situation in win or draw chessboard.

2.3.1. Avoid bad point. AVB(chessboard, player)function calculates some simple rules from the chessboard situation. These rules will avoid choosing bad moves.

2.3.2. select search function. Select the search function that should be used in the current situation. If the total number of chess pieces in the current chessboard is greater than 58, it means that the chess game has reached a late stage, and the brute force minimax search function is selected at this time. Otherwise, choose minimax function with alpha-beta prunning.

Because if the total number of chess pieces is greater than or equal to 58, the size of the remaining possible search branch trees will be small, and the computing power of the computer can handle these situations.

2.3.3. minimax search and alpha-beta prunning. These two algorithms are the main algorithms when my program is running. The minimax algorithm has been significantly applied in the field of human and machine chess, and has achieved very good results. Here, when I search using the minimax algorithm, I adjusted the number of search layers according to the current game progress. When the authorities are not complicated, I will search more deeply. I will try to make sure that every time I search, I don't time out. When a leaf node is found, the evaluation function described above will be called to evaluate the current situation. In order to speed up the search, I used the alpha beta pruning algorithm to remove nodes that may not be very important, but this may affect the evaluation nodes.

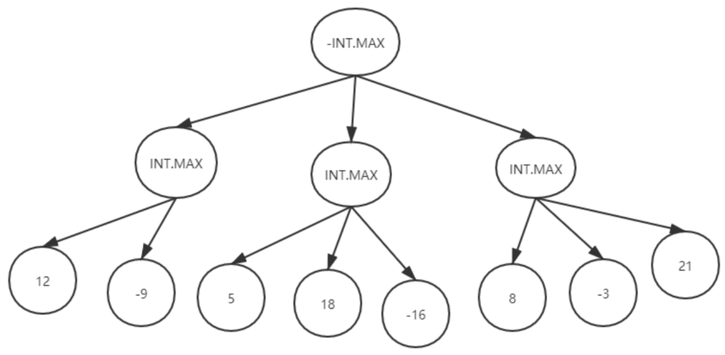
2.3.4. Special Rule. Finally, set some special rules for the number of moves selected to prevent some very bad positions from being selected in this action.

2.4. Detail of Algorithm

2.4.1. AVB function. A typical bad list is mainly composed of X points and corner points. Therefore, I will preset the coordinate list of these points and exclude the candidate coordinates from the list.I will also add to the bad list those positions that will make the opponent capture side position.

The pseudo code(1) of the AVB function is shown below.

2.4.2. calculate the candidate list. Iterate through every position on the board. If the current position is your own color, start to judge. Starting from this point, move forward in eight directions. For each direction, if you encounter your own color, move on. If you encounter the enemy's color or go to the border, you will jump out of the loop and start to judge the next direction. If a blank is encountered, it means that the blank position can be added to the candidate list.



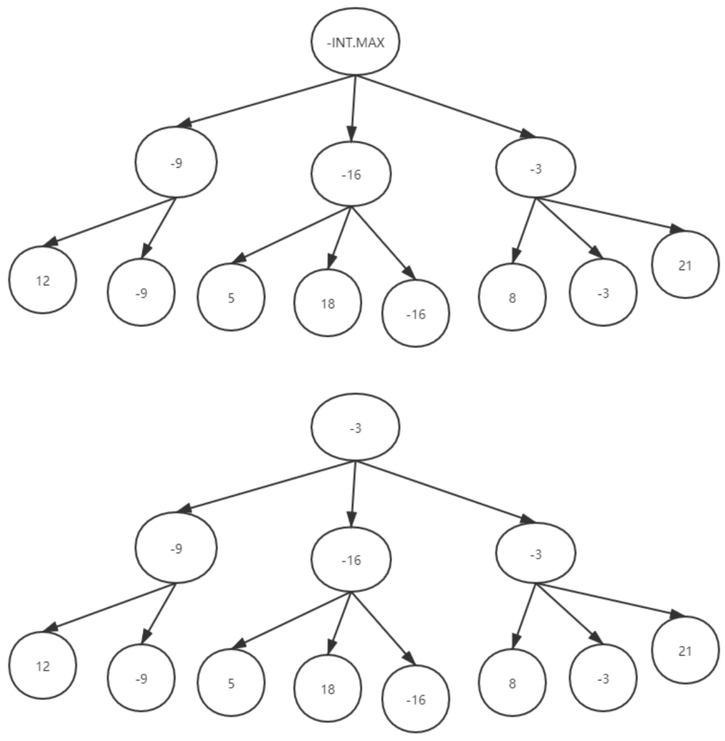


Figure 2. A Sample procedure of minimax search

Algorithm 1 Avoid Bad List Algorithm

Input: $Chessboard$, $Player$
Output: $candidate\_list$ without bad points

```
0: bad_list get all the blank X points
0: bad_list get all the blank corner points
0: bad_list get all the points which make opponent capture side
0: Calculate the bad_list
0: for each X in candidate_list do
0:     if X in bad_list then
0:         swap X and the last element in candidate_list
0:     end if
0: end for=0
```

The next piece of pseudo code(2(shows how I can get the current player's candidate list from the chessboard data.

2.4.3. Evaluation Function. Input a current board and current player to the evaluation function. The evaluation function will comprehensively consider many factors to give an evaluation score.

For example, in the early stage of a chess game, stable discs are considered an important factor. In the course of the game will produce stable points, which means the disc on that point is hard to be reversed. We calculate a factor called stability on the board to evaluate the degree of stability. The bigger stability scores indicates a better situation.

In addition, the value of the decision matrix (evaluation matrix) is also an important reference. In the later stages of the game, the real decisive factor is the number of pieces on

Algorithm 2 Calculate the candidate list

for all the points X in $chessboard$ **do**
  if X is current player **then**
    for each $direction$ in 8 directions of the board(represent in(1,1),(1,-1),(0,1).....) **do**
while true do
      go alone this direction
      renew now_coor
  if meet opponent color **then**
      $ifCan \leftarrow$ **false**
  else if meet blank **then**
      $ifCan \leftarrow$ **true**
  end if
end while

    if $ifCan$ **then**
      add $now_coor$to $candidate\_list$
    end if
    =0

both sides. Therefore, in different stages of the game, the evaluation method and formula of the evaluation function are also different.

The pseudo code(3) for calculating the evaluation score is as follows.

Algorithm 3 Calculate the candidate list

calculate the $playermatrixvalue$
calculate the evaluation matrix of opponent after renew the matrix $opponentmatrixvalue$
$matrix\_value = (playermatrixvalue - opponentmatrixvalue)$
calculate the $stable\_degree$
calculate the $move\_degree$
if $current\_state$ smaller or equal than 20 **then**
  $score \leftarrow matrix\_value + 20 * stable\_degree + move\_degree$
else if $current\_state$ bigger than 20 and smaller than 45 **then**
  $score \leftarrow matrix\_value + 20 * stable\_degree + 40 * move\_degree$
else
  $score \leftarrow matrix\_value * 30 + stable\_degree + move\_degree$
end if=0

I got the best parameter ratio from many practice and battles.

Calculate Stability Degree. When calculating stability, calculate a stability value from 0 - 4 for each point. Start searching from the four directions of each point to both ends, and both ends find the first coordinate position that is not the current player.

If at least one of these two coordinates is out of bounds, the stability is increased by one. (It means that the chess piece cannot be flipped in this direction). If these two coordinate points are opponent's pieces, the stability is increased by one. This also shows that this coordinate cannot be flipped by the opponent.

2.4.4. Minimax search. Whether it is brute force search or alpha beta pruning search, the minimax algorithm is the basis of the program. The minimax algorithm believes that the opponent will always choose the move that is most unfavorable to us. Based on this, all the moves that the opponent may take are deduced. Its pseudo code is as algorithm 4.

Algorithm 4 Minimax search

the input is $Chessboard$ and $Player$
if $layer$ is 0 **then**
  $evaluation\_score \leftarrow Eva(Chessboard, Player)$
  return $evaluation\_score$.
end if
if $Player$ is me **then**
  for each element in cal_List **do**
    get $new\_chessboard$ from element.
    remember the $biggest\_value$ of $Minimaxsearch(new\_chessboard, Opponent)$
    return the $biggest\_value$
  end for
end if
if $Player$ is opponent **then**
    for each element in cal_List **do**
      get $new\_chessboard$ from element.
      remember the $smallest\_value$ of $Minimaxsearch(new\_chessboard, me)$
      return the $smallest\_value$
    end for
end if=0

3. Empirical Verification

3.1. Dataset: What data did I used?

The main way I test code is to play against others on the online platform provided by the course. After each match, I will analyze the board as carefully as possible and consider which move is not good and which move is good. In the usability test, I also set up many examples to test the robustness of my code. As shown in the figure.3, I built a lot of test codes to verify the status of my program.

For example, I read data from the log of a chess game to judge whether the move is good or bad.

I also look for other Othello programs on the Internet to improve my own chess skills.

After the play to function was closed, I chose to invite friends to play games with my program and use other people's Othello programs to optimize my code. This picture(figure.4) shows an Othello test program I often use. [3]

```
    ]
]
# chessboard_case2 这个变量储存了从log中读取的60个棋盘
chessboard_case2 = chessboard_alalysis.get_chessboard_list("log.txt")

# testObject = AI.AI(8, AI.BLACK, 5)

now_test = 50
testAI = AI.AI(8, AI.WHITE, 5)
# testObject.go(chessboard_case2[now_test])
testLibrary.print_chessboard(chessboard_case2[now_test])

# for chessboard in chessboard_case2:
#     print(testObject.evaluation_function(chessboard,-1),end=' ')
#     print(testObject.get_stable_degree(chessboard,-1))

# for i in range(0,60):
# time_start = time.time()
testAI.go(chessboard_case2[now_test])
# time_end = time.time()
testLibrary.print_chessboard(chessboard_case2[now_test])
    # print('time is ', time_end - time_start,' s')
print(testAI.candidate_list)
```

Figure 3. Some Test Code

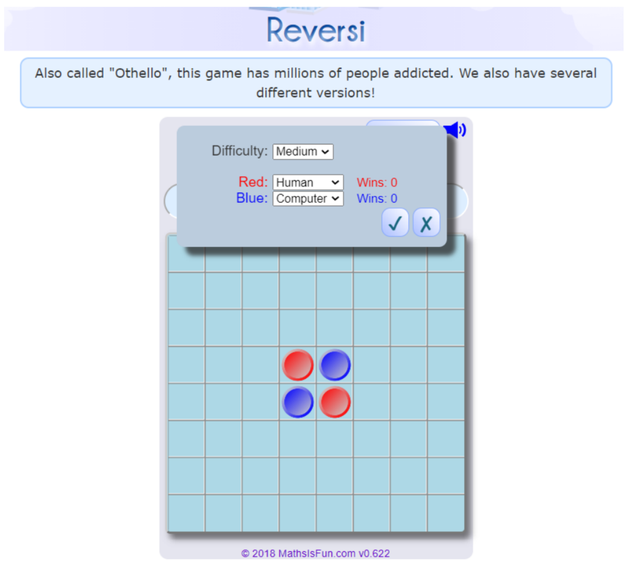


Figure 4. An Online Othello Program

3.2. Performance measure

Most of the time, I measure the performance of my program based on the log returned from an online game of Othello. Every online match will return an informative log file that includes the time spent on each step.

If after each game, my number of timeouts does not exceed three times, I think this program has no major problems in performance

My code has passed the usability test. The rank of your code in the points race 79/120, the rank of your code in the round robin is 82/120. Most of the time, it can defeats my roommates. The number of times it wins yourself or your roommates is about 20.

3.3. Hyperparameters

In my program, there are mainly five parameters that affect program performance.I adjust the parameters based on my judgment of the game. I got a big improvement from this book [4].

The first is the time node. Different time nodes are set to determine the early, mid and late game. Determining the game time stages is important for determining different evaluation formulas.

The second is the specific value of the decision matrix (evaluation matrix). In response to different situations, my program will automatically adjust the value of the decision matrix to better adapt to the current situation.

The third to fifth parameters are the weighted values of specific factors in the evaluation function. It is exactly the x, y, z mentioned above.For usability test, the points race and the round robin, I used the same parameters.

In most cases, I have been adjusting my parameters based on the game situation with others. When the current pieces number is too small, I will increase the weight of the number of pieces in the next game. When the movability is too small to affect the progress of the game, I will increase the weight of it. Adjusting parameters is a complex and non-formula process. It is difficult to describe the specific mathematical basis.

3.4. Experimental result

3.4.1. Usability test. I passed all the usability test.

3.4.2. rank in points race. It is 79/120.

3.4.3. rank in the round robin. It is 82/120.

3.5. Conclusion

I learned a lot from this project.

The minimax algorithm is a very good decision algorithm for computers. But he also has his own limitations. For example, when facing a bad opponent, you may not always be able to play very well. Therefore, the minimax algorithm is definitely not the algorithm used by the top artificial intelligence programs.

But the minimax algorithm can derive and calculate chess decisions in a way that exceeds the limits of humans. From a strategic point of view, it is a very good algorithm.

If possible, I will use machine learning methods to optimize my evaluation function, and get a relatively good program through a lot of confrontation training.

Acknowledgments

Thanks to teacher Tang Ke's artificial intelligence course, I learned a lot from it. Thanks to teacher Zhao Yao for the experimental class, I got a lot of concrete inspiration from it. Thanks to my roommate for helping me practice my program, from the easy victory at the beginning to almost impossible in the end.

References

[1] R. E. Korf and D. M. Chickering, "Best-first minimax search: Othello results," in *AAAI*, 1994, pp. 1365–1370.

[2] B. Shahzad, L. R. Alssum, and Y. Al-Ohali, “Selection of suitable evaluation function based on win/draw parameter in othello,” in *2012 Ninth International Conference on Information Technology - New Generations*, 2012, pp. 802–806.

[3] MathsIsFun.com, “Online Reversi AI,” 2020, https://www.mathsisfun.com/games/reversi.html.

[4] B. Rose, *Othello:A Minute to Learn... A Lifetime to Master*. Trademarks of Anjar Co., 2005.