Reversi Report

Name: 傅伟堡

SID：11812202

Part 1: Preliminaries

1.1 Problem Description

1.1.1 Introduction

Reversi is a strategy board game for two players, played on an 8×8 checkered board. The players take turns playing the game, and the chess pieces are divided into black and white. In the process of playing chess, if a piece of one side "clamps" the other's pieces, it can turn the clamped pieces over and become its own pieces. More important, a legal move in which at least one of the opponent's pieces is flipped. If a party has no children to play, they should stop once and let the other party go first. When neither sides has no legal move, the game ends and the player with more pieces wins.

1.1.2 Goal

In this project, we focus on designing a program that help us make choices and enable these decisions as smart as possible.

1.1.3 Algorithms

Acutually, there are many methods that can help us make decisions, including Minimax algorithm, MCTS algorithm, Convolutional Neural Network(CNN). In this project, I tried to use CNN and Minimax to solve.

On the one hand, when I tried to build a CNN to judge the probability of both sides winning in a situation, subjected to the limitation that the upload file size needs to be less than 10M, I have to reduce the number of layers of the neural network. Unfortunately, I found it difficult for the network to converge when the number of network layer is 3. In this case, I decided to give up using CNN to solve this problem.

On the other hand, I tried to use Minimax to solve this problem. In this process, I find the search level can only up to 2 due to the time limitation. In this case, I improved my algorithm with Alpha Beta pruning and the number of search levels is at least 3. Such that, I used this method as my final solution.

1.1.4 Software & Hardware

This project is written in *Python 3.7* with editor *PyCharm 2020.2.3*. By the way, *Tensorflow 2.1.0* and *Jupyter notebook* are used when I tried to implement CNN method.

1.2 Problem Application

The Reversi AI can help humans understand this game better, aid players to achieve higher performance and become more professional. Besides, we can apply this algorithmic idea to other confrontation games, which is considered a basic problem in AI.

Part 2: Methodology

2.1 Notation

Basically, there are three different states of a point in the chessboard, including empty, black and white. For our convience, we notate these three states 0, -1, 1 respectively. To make our notation clearlier, we use the notation $State(x, y) = s$ to represent the state of the point (x, y) in the chessboard is s, where $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $s \in \{-1, 0, 1\}$.

We define the following function to describle the state of current chessboard (notated as $BoardState(T)$), where $s_{x,y} = State(x, y)$, $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}$.

$$BoardState(T) = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} & s_{5,5} & s_{5,6} & s_{5,7} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} & s_{6,5} & s_{6,6} & s_{6,7} \\ s_{7,0} & s_{7,1} & s_{7,2} & s_{7,3} & s_{7,4} & s_{7,5} & s_{7,6} & s_{7,7} \end{pmatrix}$$

We use the notation $Stable(x, y) = v$ to denote whether the point (x, y) is stable, where $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $v \in \{True, False\}$. And we define $BoardStable(T)$ to describle the current chessboard stability.

$$BoardStable(T) = \begin{pmatrix} v_{0,0} & v_{0,1} & v_{0,2} & v_{0,3} & v_{0,4} & v_{0,5} & v_{0,6} & v_{0,7} \\ v_{1,0} & v_{1,1} & v_{1,2} & v_{1,3} & v_{1,4} & v_{1,5} & v_{1,6} & v_{1,7} \\ v_{2,0} & v_{2,1} & v_{2,2} & v_{2,3} & v_{2,4} & v_{2,5} & v_{2,6} & v_{2,7} \\ v_{3,0} & v_{3,1} & v_{3,2} & v_{3,3} & v_{3,4} & v_{3,5} & v_{3,6} & v_{3,7} \\ v_{4,0} & v_{4,1} & v_{4,2} & v_{4,3} & v_{4,4} & v_{4,5} & v_{4,6} & v_{4,7} \\ v_{5,0} & v_{5,1} & v_{5,2} & v_{5,3} & v_{5,4} & v_{5,5} & v_{5,6} & v_{5,7} \\ v_{6,0} & v_{6,1} & v_{6,2} & v_{6,3} & v_{6,4} & v_{6,5} & v_{6,6} & v_{6,7} \\ v_{7,0} & v_{7,1} & v_{7,2} & v_{7,3} & v_{7,4} & v_{7,5} & v_{7,6} & v_{7,7} \end{pmatrix}$$

We use $Weight(x, y) = w$ to denote the score of the point (x, y), where $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $v \in R$. And we define $WeightMap(T)$ de describle the current chessboard weight.

$$WeightMap(T) = \begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & w_{0,3} & w_{0,4} & w_{0,5} & w_{0,6} & w_{0,7} \\ w_{1,0} & w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} \\ w_{2,0} & w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} \\ w_{3,0} & w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} \\ w_{4,0} & w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & w_{4,5} & w_{4,6} & w_{4,7} \\ w_{5,0} & w_{5,1} & w_{5,2} & w_{5,3} & w_{5,4} & w_{5,5} & w_{5,6} & w_{5,7} \\ w_{6,0} & w_{6,1} & w_{6,2} & w_{6,3} & w_{6,4} & w_{6,5} & w_{6,6} & w_{6,7} \\ w_{7,0} & w_{7,1} & w_{7,2} & w_{7,3} & w_{7,4} & w_{7,5} & w_{7,6} & w_{7,7} \end{pmatrix}$$

We use $NextMoveList(T)$ to denote the list of valid move in the time $T + 1$, and $PreMove(T)$ to denote current move in the time T. Beside, we use $Evaluation(BoardState(T), player)$ to denote the score of the player in the current chessboard. More details, the evaluation score function is divided by five parts, including $MobilityScore(NextMoveList(T), BoardState(T), player)$,

$WeightScore(BoardState(T), player)$, $StableScore(BoardState(T), player)$, $NumberScore(BoardState(T), player)$, $ConnectScore(PreMove(T), player)$.

We use $BestMove(BoardState(T), player) = (i, j)$ to denote that the best policy for the current chessboard is to set position $(i. j)$ as our next step.

2.2 Data Structure

- candidate_list: A list of all valid movements of the program, and the last one is our next step by default.
- NextMoveList: A list of all valid movements in the next round.
- PreMove: A tuple of the position in the current round.
- NextboardList: A list of chessboard, the order is corresponding to NextMoveList.
- Opponent_NextMoveList: A list of all opponent's valid movements in the next round.
- DIR: A constant list of eight vectors representing eight different directions.
- DIR_o: A constant list of four vectors representing four different directions.
- BoardState: A list that represent the state of each position in the chessboard.
- BoardStable: A list that records whether the position is stable.
- BoardAccess: A list that records whether the position is accessed.

2.3 Model Design

2.3.1 Problem Formulation

The goal for our program is to calculate a bestSince the benefits of two players is opposite, and the goal for each player is to maximize their own benefit and minimize the opponent's benefit. That is to say, the advantage of one player is the disadvantage of the other. Here we just need to select a position which gives the most benefit for given chessboard. Actually, this definition is typically a Minimax algorithm problem.

2.3.2 Problem Solution

Running time limit is the biggest problem in this project, which is related to the the number of search level. However, pure Minimax algorithm will cost a lot of time, we use Alpha Beta pruning to reduce its algorithm complexity. The main idea is: we should return the current points when we find some good or bad points during our searching.

We assume that alpha is the maximum value of all the result in the maximum level and beta is the minimum value of all the result in the maximum level. Our goal is to find the maximum value and minimum value in this level then maintain alpha and beta. So if there are some values are smaller than the maximum value return from the minimum level or larger than the minimum value return from the maximum level, we should drop them out. In this case, we can solve this problem.

2.4 Detail of Algorithms

The algorithm we implement here is Minimax algorithm with Alpha Beta pruning. Acutally, my project consists of there parts: select all legal movements, minimax search and evaluation.

2.4.1 Select All Legal Movements

For a given chessboard, we need to select all legal movements first. Since we have known reversi formula, the idea is very simple. We just need to check all the positions in the chessboard and add the legal position into our $candidate\_list$. We trace all the position, and return the legal movements and its corresponding board. The pseudocode is as below:

```
Function GetAllLegalMovements(BoardState,mycolor):
    set validPlace, validBoard <- list()
    for i in range(BoardState.size):
        for j in range(BoardState[i].size):
            if there is no chess in BoardState[i][j] and we can place in this
position:
                newBoard = BoardState place (i,j)
                validPlace.append((i,j))
                validBoard.append(newBoard)
    return validPlace, validBoard
```

Then we need to check whether we can place (x, y) with $mycolor$ in the currentBoard. The idea is to traverse all eight directions and check one by one. The pseudocode is as below:

```
DIR = ((-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1))
Function CheckPlacement(x,y,board,mycolor):
    if (x,y) is not in board:
        return False
    set validflag = False and board[x][y] = mycolor
    for dir in range(len(DIR)):
        x_new <- x + DIR[dir][0]
        y_new <- y + DIR[dir][1]
        while (x_new,y_new) in board and board[x_new][y_new] == oponent_color:
            x_new <- x_new + DIR[dir][0]
            y_new <- y_new + DIR[dir][1]
        if (x_new,y_new) in board and board[x_new][y_new] == mycolor:
            while not (x_new == x and y_new == y):
                x_new <- x_new - DIR[dir][0]
                y_new <- y_new - DIR[dir][1]
                board[x_new][y_new] <- color
                if not(x_new == x and y_new == y):
                    validflag = True
    return validflag
```

2.4.2 Minimax Search

Since we have analyse the idea of Minimax before, we give the pseudocode here directly.

```
Function minimax(preMove,BoardState,mycolor,current_color,depth,alpha,beta):
    validPlace, validBoard = GetAllLegalMovements(BoardState, current_color)
    if depth == 0 or there is no validPlace:
        return Evaluation(preMove,validPlace,board,mycolor),[]
    if mycolor == current_color:
        set score_best <- -Infinite
        set move_best <- list()
        for i in range(len(validPlace)):
            current_score,_ = minimax(validPlace[i],validBoard[i],mycolor,-
current_color,depth-1,alpha,beta)
            if score_best < current_score:
                move_best <- validPlace[i]
                score_best <- current_score
            alpha <- max(alpha,score_best)
            if beta <= alpha:
                break
        return score_best, move_best
    else: // current_color == opponent
```

```
        set score_worst <- Infinite
        set move_best <- list()
        for i in range(len(validPlace)):
            current_score,_ = minimax(validPlace[i],validBoard[i],mycolor,-
current_color,depth-1,alpha,beta)
            if score_worst > current_score:
                move_best <- validPlace[i]
                score_worst <- current_score
            beta <- min(beta,score_worst)
            if beta <= alpha:
                break
         return score_worst, move_best
```

2.4.3 Evaluation

In fact, how to write evaluation function is a very important step, which is directly related to the performance of our algorithm. My evaluation consists of five part: MobilityScore, WeightScore, StableScore, NumberScore and ConnectScore.

MobilityScore

Mobility constrain the number of validPlaces in the later step. Here we define the unit of Mobility Score is the validPlaces number difference between me and opponent.

```
Function get_mobility(validPlace,board,mycolor):
    opponent_validPlace,_ <- GetAllLegalMovements(board,-mycolor)
    mobility <- len(validPlace) - len(opponent_validPlace)
    return mobility
```

WeightScore

In fact, some positions in the chessboard are much more valuable than others. For instance, four angles positions are very very important for us. So we define a WeightMap to represent the score in each position. And we define $WeightScore(T) = WeightMap(T) \times BoardState(T)$. Notice that, when our color is black (-1), we should return its absolute value.

```
Function get_WeightMap(board,mycolor):
    weight_map_score <- 0
    for i in range(len(board)):
        for j in range(len(board)):
            weight_map_score <- weight_map_score + board[i][j] * WeightMap[i][j]
    weight_map_score <- weight_map_score * mycolor
    return weight_map_score
```

StableScore

Actually, stable chess play a very important role in our project. The algorithm of finding stable chess is a little bit complex. The main idea is: We first find tha stable chess in the edge and then find others.

```
Function get_stable(board,mycolor):
    stable_num = 0
    corner = ((0,0),(0,7),(7,7),(7,0))
    stable = np.zeros((8,8),int)
    for four angles, if we place a chess on it:
        mark it as stable
```

```
        mark its connective chess as stable
    for inner chess, we have a dp function:
        stable[i][j] = stable[i-1][j]==1 and stable[i][j-1]==1 and (stable[i-1]
[j+1]==1 or stable[i+1][j-1]==1)
        stable[i][j] = stable[i-1][j]==1 and stable[i][j+1]==1 and (stable[i-1]
[j-1]==1 or stable[i+1][j+1]==1)
        stable[i][j] = stable[i][j-1]==1 and stable[i+1][j]==1 and (stable[i-1]
[j-1]==1 or stable[i+1][j+1]==1)
        stable[i][j] = stable[i+1][j]==1 and stable[i][j+1]==1 and (stable[i+1]
[j-1]==1 or stable[i-1][j+1]==1)
        stable[i][j] = stable[j-1][i]==1 and stable[j][i-1]==1 and (stable[j-1]
[i+1]==1 or stable[j+1][i-1]==1)
        stable[i][j] = stable[j-1][i]==1 and stable[j][i+1]==1 and (stable[j-1]
[i-1]==1 or stable[j+1][i+1]==1)
        stable[i][j] = stable[j][i-1]==1 and stable[j+1][i]==1 and (stable[j-1]
[i-1]==1 or stable[j+1][i+1]==1)
        stable[i][j] = stable[j+1][i]==1 and stable[j][i+1]==1 and (stable[j+1]
[i-1]==1 or stable[j-1][i+1]==1)

    Then count the number of chess which is marked as stable
    return stable_num
```

NumberScore

The number of chess is the key in this game, which is related to the result of the game. At the begining, we try to make our chess fewer. And at the end stage, we should try to make our chess as much as possible.

```
Function get_chessDiff(board,mycolor):
    my_chess, opponent_chess <- 0,0
    traverse the board and then statistics
    return my_chess - opponent_chess
```

ConnectScore

Before we place a chess, we check the number of connective places. If the number is odd, then it is advantages for us. Else, it is disadvantages for us. The method is to do a BFS.

```
Function get_smooth(i,j,board,mark):
    if (i,j) is not in board or board[i][j] has chess or mark[i][j] == 1:
        return 0
    mark[i][j] <- 1
    return 1 + get_smooth(i+1,j,board,mark) + get_smooth(i-1,j,board,mark) +
get_smooth(i,j+1,board,mark) + get_smooth(i,j-1,board,mark)
```

Part 3: Empirical Verification

3.1 Dataset

In fact, this project does not have a standard test dataset. In this case, I just use logics to check whether the available moves given by my program is correct or not. I checked correctness of the logic of the whole program for many times and confirmed that my program is correct. Additionally, to ensure my program is correct and my AI is powerful, I use the third Platform to

test my program: Botzone. Bozone is an online procedural confrontation platform, which can test my AI according to the Reversi game rules.

For the points race and the round robin, I also test my program in Botzone and judge whether the algorithms run normally as our expected during my AI is playing. In this stage, I begin to let the program with some AI which is very powerful to test our AI. When I find something is out of my expectation, I edit the hyperparameter of the program to mark my AI more intelligent.

3.2 Performance Measure

Unlike other projects we have done before, this project is hard to judge whether it is powerful. However, we have another methods to measure the performance of our project.

- Time using: We have explained before, my program will takes the full time (5s) in each step to maximize the search level dynamic. Though my AI will take many time in each step, my program will try its best to make the search level deeper.
- The search level: Actually, the number level will change dynamicly. In my implement, the range of search level is [3,7]
- Usability test: Basically, the measurement is related to the given platform. If our program is not usable, we cannot pass the test.
- The rank in the points race: Take up a large part in this project.
- The rank in the round robin: Take up a large part in this project.

3.3 Hyperparameter

3.3.1 WeightMap

At the begining, the hyperparameter in our WeightMap is as below:

$$WeightMap(T) = \begin{pmatrix} 1000 & -200 & 40 & 40 & 40 & 40 & -200 & 1000 \\ -200 & -500 & -7 & -7 & -7 & -7 & -500 & -200 \\ 40 & -7 & 3 & 2 & 2 & 3 & -7 & 40 \\ 40 & -7 & 2 & 1 & 1 & 2 & -7 & 40 \\ 40 & -7 & 2 & 1 & 1 & 2 & -7 & 40 \\ 40 & -7 & 3 & 2 & 2 & 3 & -7 & 40 \\ -200 & -500 & -7 & -7 & -7 & -7 & -500 & -200 \\ 1000 & -200 & 40 & 40 & 40 & 40 & -200 & 1000 \end{pmatrix}$$

When we place chess on one of the angles, we can set the nearby position safe. For example, if we play chess in $(0,0)$, then we change the weight in $(0,1),(1,0)$ and $(1,1)$. Then our WeightMap becomes:

$$WeightMap(T) = \begin{pmatrix} 1000 & 500 & 40 & 40 & 40 & 40 & -200 & 1000 \\ 500 & 500 & -7 & -7 & -7 & -7 & -500 & -200 \\ 40 & -7 & 3 & 2 & 2 & 3 & -7 & 40 \\ 40 & -7 & 2 & 1 & 1 & 2 & -7 & 40 \\ 40 & -7 & 2 & 1 & 1 & 2 & -7 & 40 \\ 40 & -7 & 3 & 2 & 2 & 3 & -7 & 40 \\ -200 & -500 & -7 & -7 & -7 & -7 & -500 & -200 \\ 1000 & -200 & 40 & 40 & 40 & 40 & -200 & 1000 \end{pmatrix}$$

The code is as follow:

```
Function ChangeWeightMap(board,mycolor):
    if board[0][0] == mycolor:
        board[0][1], board[1][0], board[1][1] = 500,500,500
    if board[0][7] == mycolor:
        board[0][6], board[1][7], board[1][6] = 500,500,500
    if board[7][7] == mycolor:
        board[7][6], board[6][7], board[6][6] = 500,500,500
    if board[7][0] == mycolor:
        board[6][0], board[7][1], board[6][1] = 500,500,500
```

3.3.2 Coeffient of Score

We split the stage into 6 part, and they have different coefficient. The coeffient in different stages are as below:

Stage	Mobility	Weight	Number	Stable	Connect
1-15	50	1	-20	500	0
16-30	150	1	-10	400	0
31-40	100	1	-5	300	200
41-50	80	1	0	250	160
51-60	50	0.5	50	150	100
61-64	50	0	200	100	100

Discussion

From the table, we can conclude that at the first stage, the stable chess is very important. At the mid stage, the number of mobility difference is very important. And at the end stage, the number difference is very important. Additionally, the connect score is about twice that of Mobility.

3.4 Experimental Result

In usability test: My AI passed all test cases, and the score is 100.

In points race: My AI rank is 31, and the score is 89.

In round robin: My AI rank is 21, and the score is 96.

3.5 Conclusion

- Advantage: My AI will make full use of time, and search more layers in each step as possible. Besides, the evaluation function is different accordinng to the stages.
- Disadvantage: The hyperameter is not strong enough.
- Experience: In this project, I learned the process of Minimax and knew the basic knowledge of AI. Also, I learned how to apply the Minimax Alogrithm into the reality.
- Deficiencies: Lack of knowledge of CNN.
- Possible improvement in the future: Learn more about CNN and try to adjust the hyperparameter to make the AI more powerful.

Part 4: References

[1] C. Frankland and N. Pillay, "Evolving game playing strategies for Othello," 2015 IEEE Congress on Evolutionary Computation (CEC), Sendai, 2015, pp. 1498-1504, doi: 10.1109/CEC.2015.7257065.