Report for Reversed Reversi Project

王标

11910216

1. Preliminaries

1.1 Problem description

Reversed Reversi

Reversed Reversi is a funny chess game. Its victory conditions is opposite to Reversi.

The board for black and white is an 8 by 8 square board. Play in the middle of a space, not on intersections as in go. At the beginning of the board in the middle of the two white two black four pieces placed across, black is always first. The winner is counted by the number of pieces when the game is over, and the one with less pieces wins.

In other words, My code should try to lose in the Reversi game.

Goal

What i should do is to **write an intelligent game program** to win in the Reversed Reversi game.

Algorithms

In this chess game, I use minimax and Alpha-Beta Pruning Algorithm to search the best move by predicting the next some moves. I use evaluation function to judge each move by scoring the positions of the pieces in a chess game and considerations for other factors like mobility, edge piece.

Software

I only use **Pycharm** to write python code , and test my code in the war platform Reversed Reversi.

1.2 Problem application

In the real world, AI encounters so many different problems that there is no single rule, no single function that can help explain its behavior. Chess AI is just a very early exercise for artificial intelligence.

It does give us the general idea of searching for all possible scenarios, judging the merits of the situation and choosing the best one.

Through the study of the complete information game, we have more experience to think about how to face the incomplete information game. Finally, the application of artificial intelligence in real life scenarios such as unmanned driving, urban resource mobilization, etc.

2. Methodology

2.1 Notation

- (x,y) means the position of point in the chessboard
- E(x,y) means the score of chessboard after putting the chess in the position (x,y)
- K_1 means the coefficient of E(x,y)
- K_2 means the coefficient of the mobility
- k_3 means the coefficient of the edge piece
- M_w means the mobility of white
- M_b means the mobility of black
- N_w means the number of edge pieces of white
- N_b means the number of edge pieces of black
- W means the sum of score the positions of chess

$$W = -chessboard \times wight$$

2.2 Data structure

- In the Reversed Reversi chessboard, use 1 , -1 ,and 0 represent white chess , black chess and empty.

  ```
  COLOR_BLACK=-1
  COLOR_WHITE= 1
  COLOR_NONE = 0
  ```

-
  ```
  chessboard_size
  # the length of square
  ```

-
  ```
  color
  # represent the player in this turn wheather white or black
  ```

-
  ```
  candidate_list
  # the moves the player can go
  ```

-
  ```
  chessbord
  # the list 8*8 respresent the chessboard
  ```

-
  ```
  place
  # the move the player finally decide to go
  ```

- Every position has its weight to calculate the score of the chessboard.

```
weight = [
    [100, -5, 10,  5,  5, 10, -5, 100],
    [-5, -45,  1,  1,  1,  1, -45, -5],
    [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],
    [-5, -45,  1,  1,  1,  1, -45, -5],
    [100, -5, 10,  5,  5, 10, -5, 100]
]
```

2.3 Model Design

2.3.1 function

```
class AI(object):
  def __init__(self, chessboard_size, color, time_out)
   # initialization
  def go(self, chessboard)
   # decide the move
  def is_on_board(x, y)
   # judge wheather (x,y) in the chessboard
  def turn_over(chessboard, x ,y ,color)
   # The new chessboard I get, if I put my chess in the (x,y)
  def can_go(chessboard, color)
   # judge wheather color can make a move
  def is_terminal(chessboard, color)
   # judge wheather game is terminal
  def step(chessboard, color)
   # count the mobility of color
  def chessnumber(chessboard, color)
   # count number of the chesses in the chessboard number
  def alphabeta_search(chessboard, color,  d)
   # d is the layer
  def calculate_edge(chessboard, color)
   # count the number of edge pieces
  def calculate(chcessboard, color)
   #evaluation function
```

2.3.2 Solution

The chess game is divided into three stages: early stage, middle stage and later stage by chessnumber().

early stage : chessnumber()< 24

middle stage : 24<=chessnumber()<58

later stage : chessnumber()>= 58

early stage

- I use alpha-beta_search(chessboard, color, d) and set d = 3.
- I try to take up some important positions.
- Evaluation function will focus on score of the position.

middle stage

- I use alpha-beta_search(chessboard, color, d) and set d = 4
- I try to limit the opponent's mobility.
- Evaluation function will focus on score of mobility.

later stage

- I use alpha-beta_search(chessboard, color, d) to search the finally result
- I choose the move that I win in the finally result

2.3.3 evaluation function

if color is block , we change the place of N_w and N_b /   M_w and M_b.

$$E(x,y) = k_1 \times W + k_2 \times (M_w - M_b) + k_3 \times (N_w - N_b)$$

early edge

$$k_1 = 5, k_2 = 3, k_3 = 3$$

middle edge

$$k_1 = 3, k_2 = 7, k_3 = 3$$

later stage

judge the chessboard result

if color wins

$$E(x,y) = infinity$$

if color loses

$$E(x,y) = -infinity$$

2.4 Detail of algorithms

2.4.1 alpha-beta_search(chessboard, color, d)

```
def alphabeta_search(chessboard, player,  d):

    def max_value(chessboard, player, alpha, beta, depth):
        if is_terminal(chessboard, player) or depth == 0:
            return calculate(chessboard, player)
        v = -infinity
        for a in can_go(chessboard, player):
            a_board = turn_over(chessboard, a[0], a[1], player)

            v2= min_value(a_board, -player, alpha, beta, depth-1)
            if v2 > v:
                v = v2
                alpha = max(alpha, v)
            if v >= beta:
                return v
        return v

    def min_value(chessboard, player, alpha, beta, depth):
        if is_terminal(chessboard, player) or depth == 0:
            return calculate(chessboard, -player)
        v = infinity
        for a in can_go(chessboard, player):
            a_board = turn_over(chessboard, a[0], a[1], player)
            v2 = max_value(a_board, -player, alpha, beta, depth-1)

            if v2 < v:
               v = v2
               beta = min(beta, v)
            if v <= alpha:
                return v
        return v

    if step(chessboard)>58:
        d = 64 - step(chessboard)
    elif step(chessboard)>24:
         d = 4
    else:
        d = 3

    best_score = -infinity-1
    beta = infinity
    best_action = None

    for a in can_go(chessboard, player):
        a_board = turn_over(chessboard, a[0], a[1], player)
        v = min_value(a_board, -player, best_score, beta, d-1)
        if v > best_score:
            best_score = v
            best_action = a

    return best_action
```

2.4.2 calculate(chessboard, color)

```
def calculate(chessboard, color):

  if chessnumber(chessboard)< 24 :
    value = 5 * (weight * chessboard) + 3*(mobility(color) - mobility(-color)) +
3*(edge(color) - edge(-color))

  if 24=<chessnumber(chessboard)<58:
    value = 3 *(weight*chessboard) + 5 *(mobility(color) - mobility(-color))+ 3*
(edge(color) - edge(-color))

  if chessnumber(chessboard)>=58:
      if winner(color):
            value = infinity
      else
            value = -infinity
         return value
```

3. Empirical Verification

3.1 Dataset

- In the early days, I used usability test to determine whether my can_go () function could run correctly and whether candicate_list is right. I also set up some test cases.
- After usability tests, I download the game data fighting with other people on the AI platform. Set the input on the code to debug the code. Use Pycharm debug pattern to check every step running on my code. And set up special situations for testing.
- If I can't use Playto function, I'll write a script that allows my code to be tested on other platforms. Or write a web game of Reversed Reversi, put my code into man-machine mode and invite others to play. Either way, I can collect data and evaluate my evaluation function.

3.2 Performance

I only used rank wins and losses to judge the performance of my code, and I tried many search levels, setting them to the maximum number of levels that would not time out. I often invite my roommates to play against my code. Through some suggestions from my roommate, I made some improvements. When I split the game into three phases, the performance of my code improved dramatically.

3.3 Hyper parameters

- **Weight**

  The weight for each position on the board is important, indicating the position's importance to the situation. It would have been more helpful to get some position early.
- **search layer**

  Search speed is an important factor in chess ai comparison.The more search layers of your code, the more likely you are to win the game.
- **stage**

  The game is divided into several stages and processed separately to make the code more intelligent. I only have three phases, and if I could divide the game phases more carefully, the code would have a pattern for each phase. The odds will go up a lot.
- **mobility**

An important strategy in Reversed Reversi is to reduce the number of positions the opponent can make, forcing the opponent to make a bad move. So it's an important part of the code to consider mobility.

3.4 Experimental results

- The test case i pass in lab_1 is 10.
- My rank in the AI rank list is 172th.
- My rank in the round robin is 165th.

3.5 conclusion

- Based on my experimental results, my code is not doing very well. The reason is that my evaluation function doesn't make the code think the way I want it to.
- The advantage of my algorithm may be the stage analysis of the game.
- The disadvantage of my algorithm is that the evaluation is too rough and many of the weights are not quite correct.
- From this project, I have a rough understanding of artificial intelligence in the complete information game, and also feel the charm of AI algorithm. It's not easy to get the code to work and execute the strategy I've laid out.
- I think my evaluation function has a lot of room for improvement and should not be a simple linear relationship. For the weight test, it should be possible to use mechanical learning to adjust, fitting better weight. I can consider the situation of the chess game in more detail and code the implementation of multiple strategies. My search algorithm is also the simplest model, and there are many ways to optimize speed.

4. Reference

[1] Tronicke, M. (2020). Othello. Shakespeare Bulletin, 38(1), 121.

[2] Shoham, Y., & Toledo, S. (2002). Parallel randomized best-first minimax search. \emph{Artificial Intelligence, 137}(1-2), 165-196.