Report for Project Reversed Reversi

Boyan Li
Department of Computer Science and Engineering
Southern University of Science and Technology
11912914
11912914@mail.sustech.edu.cn

I. Preliminaries

A. Problem Description

Reversed Reversi is a relatively simple board game. Players take turns placing disks on the board with their assigned color facing up. During a play, any disk of the opponent's color that are in a straight line and bounded by the disk just placed and another discs of the current player's color are turned over to the current player's color. The object of the game is to have the fewest discs turned to display your color when the last playable empty square is filled. The goal of the project is to implement the AI algorithm of Reversed Reversi according to the interface requirements, pass the usability testing and try to rank higher in the round robin.

B. Software and Hardware

The project is developed using PyCharm as integrated development environment based on Python language. The local testing platform is Windows 10 Professional Edition with Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz of 8 cores and 16 threads. The training platform is one node in Taiyi cluster with 2 Xeon Gold 6148 CPU @ 2.4GHz of 20 cores. The course testing platform is Xeon Gold 6240 @ 2.6GHz.

C. Algorithms

The project algorithms include Minimax Search, Alpha-Beta Pruning and Genetic Algorithm [1].

Minimax Search 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 [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, which is an adversarial search algorithm used commonly for machine playing of two-player games. I use the Minimax Search with Alpha-Beta Pruning to do the decision in the Reversed Reversi Game. Although Alpha-Beta Pruning can reduce the search space and speed up the search efficiency, the quality of the final decision by the search tree is largely dependent on the evaluation of the node value [3].

Traditionally, people can set up evaluation function based on the prior experience of games, but it is often difficult to guarantee the optimality of evaluation functions in this way. In computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. Thus, I use genetic algorithm to help me get a high-quality evaluation function to improve the performance of the Minimax Search Algorithm.

D. Problem Applications

In game theory and economic theory, a zero-sum game is a mathematical representation of a situation in which an advantage that is won by one of two sides is lost by the other. If the total gains of the participants are added up, and the total losses are subtracted, they will sum to zero. Zero-sum game is very common in our life, such as poker, chess or reversi in games, futures contracts and options in the market, etc., which are widely used [4].

In this project, we implement the AI algorithm of Reversed Reversi Game to help us understand the zero-sum game, learn the method to solve the zero-sum game problem, and finally choose the optimal strategy under a variety of choices.

II. Methodology

A. Notation

The list of all my notations used in my report is given below.

- (x, y): The square of x^{th} row and y^{th} column in the chessboard.
- C_{self}: The color of self, -1 for black, 1 for white.
- C_{oppo}: The color of opponent, which is equals $-C_{self}$.
- $R^1(x, y)$: The general reward of board square (x, y) in the stage one.
- $R^2(x, y)$: The general reward of board square (x, y) in the stage two.
- $A^1(x, y)$: The action reward of board square (x, y) in the stage one.
- $A^2(x, y)$: The action reward of board square (x, y) in the stage two.
- W_s: The weight of stability difference.
- W_f^1: The weight of frontier difference in the stage one.
- W_f^2: The weight of frontier difference in the stage two.
- W_p^1: The weight of pieces difference in the stage one.
- W_p^2: The weight of pieces difference in the stage two.
- $C(x, y)$: The chess color in the square (x, y), 0 for no chess, -1 for black chess, and 1 for white chess.
- D_s: The difference between the number of opponent stable pieces and self.

- D_f: The difference between the number of self frontier pieces and opponent.
- D_p: The difference between the number of self pieces and opponent.
- D_m: The maximum searching depth of Minimax algorithm.
- B_t: The chessboard in time t.
- C_t: The player color in time t.
- V: The evaluated value of the chessboard.
- S_{reward}: The score of general reward of the board.
- S_{action}: The score of action reward of the board.
- S_{stable}: The score of stable pieces of the board.
- $S_{frontier}$: The score of frontier pieces of the board.
- S_{pieces}: The score of pieces number of the board.
- $Stage$: The stage of the game, 1 for early stage, and 2 for final stage.

B. Data structure

The list of all my key data structures is given below.

- $chessboard$: a numpy array about chessboard information.
- $candidate\_list$: a list of the legal moves for specific player and chessboard.
- $params$: a dict including all the parameters about evaluation, including $R^1(x,y)$, $R^2(x,y)$, $A^1(x,y)$, $A^2(x,y)$, W_s, W_f^1, W_f^2, W_p^1, W_p^2.
- $Node$: a dataclass encapsulates board, color, search_type, depth, alpha, beta and value, as a node in the search tree.
- $search\_type$: a int value indicating the search_type of a node, MAX_SEARCH = 1 and MIN_SEARCH = -1.
- $directions$: a list including all eight directions from a square.
- $Game$: a class encapsulates all the game logic.
- $MinimaxSearchPlayer$: a class encapsulates the minimax with alpha-beta pruning algorithm.
- $FinalSearch$: a class encapsulates the minimax search in the final stage of one game. The goal is to try to search for a absolute winning strategy during the final stage (such as the last 10 steps in one game).
- GA: a class encapsulates complete genetic algorithm.

C. Model design

1) AI Algorithm: Based on the rules of Reversed Reversi Game, the game ends when the board is full or there is no legal moves for either player to play. Both players of the game take turns calling the $go()$ method in the AI class to return the $candidate\_list$, where the last element is the best move.

In function $go()$, the AI algorithm of the game varies according to the period of the game. I define the period of the game as the late period when the number of empty positions in the board does not exceed ten, otherwise it is the early period.

In the early stage of the game, I use minimax algorithm combined with alpha-beta pruning to search the game tree. Starting from the search depth of 3, I try to search a deeper depth with an infinite loop in limited time, and after each search, I add the current loop's optimal move to the $candidate\_list$.

In the final period of the game, I use two ways to get the best move: one is normal minimax algorithm with depth 2, the other is endgame strategy search (a modified minimax search trying to find an absolute winning strategy).If there not exist an absolute winning strategy, the best move is from former, otherwise from the latter.

When it comes to evaluation in minimax algorithm, the function $get\_node\_value()$ in class $MinimaxSearchPlayer$ is to estimate the value of the current node (current board). It consists of 5 parts:

- Board Reward Score:

$$S_{reward} = \begin{cases} \sum(-C(x,y) \cdot C_{self}) \cdot R^1(x,y) & \text{if stage=1} \\ \sum(-C(x,y) \cdot C_{self}) \cdot R^2(x,y) & \text{if stage=2} \end{cases}$$

- Action Reward Score:

$$S_{action} = \begin{cases} \sum_{self} A^1(x,y) - \sum_{oppo} A^1(x,y) & \text{if stage=1} \\ \sum_{self} A^2(x,y) - \sum_{oppo} A^2(x,y) & \text{if stage=2} \end{cases}$$

- Stable Pieces Score:

$$S_{stable} = W_s \cdot D_s$$

- Frontier Pieces Score:

$$S_{frontier} = \begin{cases} W_f^1 \cdot D_f & \text{if stage=1} \\ W_f^2 \cdot D_f & \text{if stage=2} \end{cases}$$

- Pieces Number Score:

$$S_{pieces} = \begin{cases} W_p^1 \cdot D_p & \text{if stage=1} \\ W_p^2 \cdot D_p & \text{if stage=2} \end{cases}$$

Finally, the evaluated total value of a node is:

$$V = S_{reward} + S_{action} + S_{stable} + S_{frontier} + S_{pieces}$$

2) GA Algorithm: Genetic Algorithm is used to search a excellent evaluation function (evaluation parameters) in huge search space. I implement a genetic algorithm class, which packaged the complete genetic algorithm of each process function, including $generate\_origin\_population()$, $translate\_population()$, $fitness()$, $selection()$, $compete()$, $cross\_over()$, $mutation()$, $train()$, etc. In my design, a specie has a piece of DNA. DNA is a fixed length binary coding sequence, in which genes are arranged, and each gene corresponds to a translated parameter, so a piece of DNA has a complete set of parameters required for a evaluation function.

Based on the goal of GA in this question, I set the length of DNA is 314, which includes 45 genes totally: 10 genes with length 7 bits representing R^1, 10 genes with length 7 bits representing R^2, 10 genes with length 7 bits representing A^1, 10 genes with length 7 bits representing A^2, 1 gene with length 8 bits representing W_s, 2 genes with length 6 bits representing W_f^1 and W_f^2, 2 genes with length 7 bits representing W_p^1 and W_p^2. It's worth nothing that because of the symmetry of the board's positions, there are only 10 really different positions on the entire 8 * 8 board. Thus, we only need to get 10 parameters in the positions, then we can get other positions by symmetry. This is very necessary, on the one hand to the evolution of constraints, on the other hand greatly reduce the search space.

We can start GA algorithm by calling $train()$ method. In this method, we firstly call $generate\_origin\_population()$ to generate an origin population randomly with specific size such as 60. Then, we start a loop, which represent the generation times. For each loop, we call $selection()$ to select half the population size excellent species to be alive. To be specific, in $selection()$ method, we call $fitness()$ method, which will use the multi-process method to make each specie of the population fight against all other species. After all the battles are completed, the number of wins and losses of each specie is counted, and the minimum value of the specie black and white wins is taken as its fitness, which can improve the robustness. At the same time, multi-processing is also very important idea to speed up the selection process. After $selection()$, we do $cross\_over()$. We random select two specie in the population and get their DNA, then for each bit in DAN we randomly select one from both to generate the child specie. Finally, we do mutation to the child's DNA based on the probability p_mutation and append it to the population.

D. Details of algorithms

Here are the details of algorithms mentioned above:

1) Game: Class $Game$ provides many interface about game logic.

- get_init_board (self). The function will return an initail chessboard.

Algorithm 1 get_init_board (self)

```
init_board ← np.zeros((8, 8))
init_board[3, 3] ← COLOR_WHITE
init_board[4, 4] ← COLOR_WHITE
init_board[3, 4] ← COLOR_BLACK
init_board[4, 3] ← COLOR_BLACK
return init_board
```

- get_reverse_list (self, board, move, color). Based on the board, move and current color to get the possible reversed pieces list.

Algorithm 2 get_reverse_list (self, board, move, color)

```
x, y ← move
reverse_list ← [ ]
if board[x, y]! = COLOR_NONE then
    return reverse_list
end if
for dx, dy in directions do
    dir_reverse_list ← [ ]
    dir_x, dir_y ← x + dx, y + dy
    dir_reverse_flag ← False
    while 0 <= dir_x < 8 and 0 <= dir_y < 8 do
        if board[dir_x, dir_y] == −color then
            dir_reverse_list.append((dir_x, dir_y))
            dir_x, dir_y = dir_x + dx, dir_y + dy
        else if board[dir_x, dir_y] == color then
            dir_reverse_flag ← True
            break
        else
            break
        end if
    end while
    if dir_reverse_flag and len(dir_reverse_list)! = 0 then
        reverse_list.extend(dir_reverse_list)
    end if
end for
return reverse_list
```

- $get\_next\_board(self, board, move, color)$ Based on the board, move and current color, return the next chessboard after move.

Algorithm 3 $get\_next\_board(self, board, move, color)$

```
board ← np.copy(board)
reverse_list ← self.get_reverse_list(board, move, color)
reverse_list.append(move)
for x, y in reverse_list do
    board[x, y] ← color
end for
if not self.has_legal_move(board, −color) then
    return board, color
else
    return board, −color
end if
```

- get_legal_moves (self, board, color). Based on the board and color, return the legal moves list.

Algorithm 4 get_legal_moves (self, board, color)

```
legal_moves ← set()
for i in range(8) do
    for j in range(8) do
        if board[i, j] == color then
            l ← self._get_legal_move_from_
                    location(board, (i, j), color)
            legal_moves.update(l)
        end if
    end for
end for
return list(legal_moves)
```

- has_legal_move (self, board, color). Based on the board and color, return whether the player has legal move.

Algorithm 5 has_legal_move (self, board, color)

```
for i in range(8) do
    for j in range(8) do
        if board[i, j] == color then
            flag ← self._check_legal_move_
                from_location(board, (i, j), color)
            if flag then
                return True
            end if
        end if
    end for
```

end for

- check_game_end (self, board). Based on the board, check the game is whether end.

Algorithm 6 check_game_end (self, board)

$b \leftarrow self.get\_legal\_moves(board, COLOR\_BLACK)$
$w \leftarrow self.get\_legal\_moves(board, COLOR\_WHITE)$
if $len(b) == 0$ and $len(w) == 0$ **then**
  $board\_sum \leftarrow np.sum(board)$
  if $board\_sum == 0$ **then**
    return $True, 0$
  else if $board\_sum > 0$ **then**
    return $True, -1$
  else
    return $True, 1$
  end if
else
  return $False, None$
end if

- _get_legal_move_from_location (self, board, location, color). Based on the board, location and color, get the next possible legal moves due to the specific location, the piece in the location input should be the same color as the color input.

Algorithm 7 _get_legal_move_from_location (self, board, location, color)

$x, y \leftarrow location$
$legal_moves \leftarrow set()$
for dx, dy in $directions$ **do**
  $dir\_x, dir\_y \leftarrow x + dx, y + dy$
  $has\_reverse\_piece \leftarrow False$
  while $0 <= dir\_x < 8$ and $0 <= dir\_y < 8$ **do**
    if $board[dir\_x, dir\_y] == -color$ **then**
      $has\_reverse\_piece \leftarrow True$
      $dir\_x, dir\_y = dir\_x + dx, dir\_y + dy$
    else if $board[dir\_x, dir\_y] == color$ **then**
      break
    else
      if $has\_reverse\_piece$ **then**
        $legal\_moves.add((dir\_x, dir\_y))$
        break
      end if
    end if
  end while
end for
return $legal\_moves$

- _check_legal_move_from_location (self, board, location, color). Based on the board, location and color, check the color has whether next possible legal moves, the piece in the location input should be the same color as the color input.

Algorithm 8 _check_legal_move_from_location (self, board, location, color)

$x, y \leftarrow location$
for dx, dy in $directions$ **do**
  $dir\_x, dir\_y \leftarrow x + dx, y + dy$
  $has\_reverse\_piece \leftarrow False$
  while $0 <= dir\_x < 8$ and $0 <= dir\_y < 8$ **do**
    if $board[dir\_x, dir\_y] == -color$ **then**
      $has\_reverse\_piece \leftarrow True$
      $dir\_x, dir\_y = dir\_x + dx, dir\_y + dy$
    else if $board[dir\_x, dir\_y] == color$ **then**
      break
    else
      if $has\_reverse\_piece$ **then**
        return $True$
        break
      end if
    end if
  end while
end for
return $False$

- get_front_pieces_num (self, board, view_color). Based on the board and view_color (self color), return the number of self frontier pieces and opponent frontier pieces.

Algorithm 9 get_front_pieces_num (self, board, view_color)

$self\_front\_num, self\_front\_num \leftarrow 0, 0$
for i in range(8) **do**
  for j in range(8) **do**
    if $board[i, j]! = COLOR_NONE$ **then**
      for k in range(8) **do**
        $x \leftarrow i + directions[k, 0]$
        $y \leftarrow j + directions[k, 1]$
        if $0 <= x < 8 and 0 <= y < 8andboard[x, y] == COLOR\_NONE$ **then**
          if $board[x, y] == view\_color$ **then**
            $self\_front\_num+ = 1$
          else if $board[x, y] == -view\_color$ **then**
            $oppo\_front\_num+ = 1$
          end if
        end if
      end for
    end if
  end for
end for
return $self\_front\_num, oppo\_front\_num$

2) MinimaxSearchPlayer: Class $MinimaxSearchPlayer$ encapsulates the minimax with alpha-beta pruning algorithm, which is the main algorithm of AI.

- __init__ (self, game, root_board, root_color, search_depth, params). Initialize the minimaxSearchPlayer with some important information.

Algorithm 10 __init__ (self, game, root_board, root_color, search_depth, params)

$self.game \leftarrow game$
$self.root\_board \leftarrow root\_board$
$self.root\_color \leftarrow root\_color$

```
self.search_depth ← search_depth
self.root_node ← Node(root_board, root_color,
        MAX_SAERCH, 0, −inf, inf, −inf)
self.root_child_node ← [ ]
self.root_legal_moves ← [ ]
self.reward_matrix_1 ←
        params[reward_matrix_1]
self.reward_matrix_2 ←
        params[reward_matrix_2]
self.action_matrix_1 ←
        params[action_matrix_1]
self.action_matrix_2 ←
        params[action_matrix_2]
self.stable_weight ←
        params[stable_weight]
self.front_1 ← params[front_1]
self.front_2 ← params[front_2]
self.num_1 ← params[num_1]
self.num_2 ← params[num_2]
```

- minimax_search (self, node). The minimax search algorithm with alpha-beta pruning.

Algorithm 11 minimax_search (self, node)

```
if node.depth == 1 then
    self.root_child_node.append(node)
end if
end, winner ← self.game.check_game_end(node.board)
if end then
    if winner == self.root_color then
        node.value = inf
    else
        node.value == −inf
    end if
    return node.value
end if
if node.depth == self.search_depth then
    node.value ← self.gat_node_value(node)
    return node.value
end if
next_moves ← self.game.
get_legal_moves(node.board, node.color)
if node.depth == 0 then
    self.root_legal ← next_moves
end if
if len(next_moves) == 0 then
    next_search_type ← −node.search_type
    if next_search_type == MAX_SEARCH then
        next_val ← −inf
    else
        next_val ← inf
    end if
    next_node ← Node(node.board, −node.color,
    next_search_type, node.depth+1, node.alpha,
    node.beta, next_val)
    back_value ← self.minimax_search
                (next_node)
    node.value ← back_value
    return node.value
end if
for move in next_moves do
    next_board, next_color ← self.game.
        get_next_board(node.board, move, node.color)
    if next_color! = node.color then
        next_search_type ← −node.search_type
    else
        next_search_type ← node.search_type
    end if
    if next_search_type == MAX_SEARCH then
        next_value ← −inf
    else
        next_value ← inf
    end if
    next_node ← Node(node.board, −node.color,
    next_search_type, node.depth+1, node.alpha,
    node.beta, next_val)
    back_value ← self.minimax_search
                (next_node)
    if back_value == inf then
        node.value ← back_value
        return node.value
    end if
    if node.search_type == MAX_SEARCH then
        node.value ←
        max(node.value, back_value)
        if node.value >= node.beta then
            return node.value
        end if
        node.alpha ←
        max(node.alpha, node.value)
    else
        node.value ←
        min(node.value, back_value)
        if node.value <= node.beta then
            return node.value
        end if
        node.beta ←
        min(node.beta, node.value)
    end if
end for
return node.value
```

- get_node_value (self, node). The evaluation function to get the value of a node.

Algorithm 12 get_node_value (self, node)

```
board ← node.board
self_pieces_num ← 0
oppo_pieces_num ← 0
for i in range(8) do
    for i in range(8) do
```

```
            if board[i, j] == self.root_color then
                self_pieces_num+ = 1
            else if board[i, j] == −self.root_color then
                oppo_pieces_num+ = 1
            end if
        end for
end for
pieces_num = self_pieces_num+
             oppo_pieces_num
game_stage ← int((pieces_num − 5)/30)
stable_matrix ← np.zeros((8, 8))
board_reward_score ← 0
action_reward_score ← 0
stable_score ← 0
front_score ← 0
pieces_num_score ← 0
Next if body is calculate the stable pieces number due to [0,0]
if board[0, 0]! = COLOR_NONE then
    check_color ← board[0, 0]
    width ← 0
    for i in range(8) do
        if board[0, i] == check_color then
            width ← i
            stable_matrix[0, i] ← 1
        else
            break
        end if
    end for
    for i in range(8) do
        if board[i, 0]! = check_color then
            break
        end if
        for j in range(width) do
            if board[i, j] == check_color then
                stable_matrix[i, j] ← 1
                width ← j
            else
                break
            end if
        end for
        if width == 0 then
            width ← 1
        end if
    end for
    depth ← 0
    for i in range(8) do
        if board[i, 0] == check_color then
            depth ← i
            stable_matrix[i, 0] ← 1
        else
            break
        end if
    end for
    for j in range(8) do
        if board[0, j]! = check_color then
            break
        end if
        for i in range(depth) do
            if board[i, j] == check_color then
                stable_matrix[i, j] ← 1
                depth ← i
            else
                break
            end if
            if depth == 0 then
                deoth ← 1
            end if
        end for
    end for
end if
    (We ignore the similar process to calculate the stable pieces number due to [0,7], [7,0] and [7,7])
self_stable ← 0
oppo_stable ← 0
for i in range(8) do
    for j in range(8) do
        if board[i, j] == self.root_color then
            if game_stage == 0 then
                board_reward_score− = self.reward_matrix1[i, j]
            else
                board_reward_score− = self.reward_matrix2[i, j]
            end if
            if stable_matrix[i.j] == 1 then
                self_stable+ = 1
            end if
        else if board[i, j] == −self.root_color then
            if game_stage == 0 then
                board_reward_score+ = self.reward_matrix1[i, j]
            else
                board_reward_score+ = self.reward_matrix2[i, j]
            end if
            if stable_matrix[i.j] == 1 then
                oppo_stable+ = 1
            end if
        end if
    end for
end for
stable_score ← (oppo_stable − self_stable) ∗ self.stable_weight
self_actions ←
self.game.get_legal_moves(board, self.root_color)
oppo_actions ←
self.game.get_legal_moves(board, −self.root_color)
for x, y in self_actions do
    if game_stage == 0 then
        action_reward_score+ = self.actions_matrix1[x, y]
```

```
    else
        action_reward_score+ = self.actions_matrix_2[x, y]
    end if
end for
for x, y in oppo_actions do
    if game_stage == 0 then
        action_reward_score− = self.actions_matrix_1[x, y]
    else
        action_reward_score− = self.actions_matrix_2[x, y]
    end if
end for
self_front, oppo_front ←
self.game.get_front_pieces_num
(board, self.root_color)
if game_stage == 0 then
    front_score ← (self_front − oppo_front) ∗ self.front_1
    pieces_num_score ← (self_pieces_num − oppo_pieces_num) ∗ self.num_1
else
    front_score ← (self_front − oppo_front) ∗ self.front_2
    pieces_num_score ← (self_pieces_num − oppo_pieces_num) ∗ self.num_2
end if
return board_reward_score + action_reward_score + stable_score + front_score + pieces_num_score
```

- get_action (self). The method to get a best move for AI based on the minimax algorithm.

Algorithm 13 get_action (self).

```
max_val ← −inf
best_move ← None
self.minimax_search(self.root_node)
for i, node in enumerate(self.root_child_node) do
    if max_val <= node.value then
        max_val ← node.value
        best_move ← self.root_legal_moves[i]
    end if
    if max_val == inf then
        break
    end if
end for
return best_move
```

3) FinalSearch: Class $FinalSearch$ encapsulates the modified minimax algorithm in the final stage of game, which will give an absolute winning strategy possibly.

- __init__(self, game, root_color). Initialize the final player.

Algorithm 14 __init__ (self, game, root_board, root_color, search_depth, params)

```
self.game ← game
self.root_color ← root_color
self.best_move ← None
```

- search(self, board, color, depth). The main method for searching. Depth is just used to maintain $self.best\_move$.

Algorithm 15 search(self, board, color, depth)

```
if color == self.root_color then
    legal_moves ← self.game.
    get_legal_moves(board, color)
    move_flag_list ← [ ]
    for move in legal_moves do
        next_board, next_color ← self.game.
        get_next_board(board, move, color)
        end, winner ← self.game
        check_game_end(next_board)
        if end then
            if winner == self.root_color then
                if depth == 0 then
                    self.best_move = move
                end if
                return 1
            else if winner == −self.root_color then
                move_flag_list.append(−1)
                continue
            else
                move_flag_list.append(0)
                continue
            end if
        end if
        flag ← self.search
        (next_board, next_color, depth + 1)
        move_flag_list.append(flag)
        if flag == 1 then
            if depth == 0 then
                self.best_move ← move
            end if
            return 1
        end if
    end for
    for i in range(len(move_flag_list)) do
        if move_flag_list[i] == 0 then
            if depth == 0 then
                self.best_move = legal_moves[i]
            end if
            return 0
        end if
    end for
    return -1
else
    legal_moves ← self.game.
    get_legal_moves(board, color)
    move_flag_list ← [ ]
    for move in legal_moves do
```

```
            next_board, next_color ← self.game.
            get_next_board(board, move, color)
            end, winner ← self.game
            check_game_end(next_board)
            if end then
                if winner == −self.root_color then
                    return -1
                else if winner == self.root_color then
                    move_flag_list.append(1)
                    continue
                else
                    move_flag_list.append(0)
                    continue
                end if
            end if
            flag ← self.search
            (next_board, next_color, depth + 1)
            move_flag_list.append(flag)
            if flag == −1 then
                return -1
            end if
        end for
        for flag in move_flag_list do
            if flag == 0 then
                return 0
            end if
        end for
        return 1
    end if
```

4) GA: GA is a class encapsulates complete genetic algorithm. It is used to get a good evaluation function in minimax search algorithm, which is very important for the performance of AI algorithm.

- __init__ (self, population_size, p_mutation). Initialize the config of GA.

Algorithm 16 __init__ (self, population_size, p_mutation)

```
assertpopulation_size%2 == 0
self.population_size ← population_size
self.p_mutation ← p_mutation
self.population_list ← [ ]
self.trans_population_list ← [ ]
```

- generate_random_dna (self). Generate a random DNA (binary sequence).

Algorithm 17 generate_random_dna (self)

```
dna ← empty string
for i in range(314) do
    dna+ = str(random.randint(0, 1))
end for
return dna
```

- generate_origin_population (self). Generate a randomly origin population.

Algorithm 18 generate_origin_population (self)

```
for i in range(self.population_size) do
    self.population_list.append
    ({dna : self.generate_random_dna(),
    generation : 0})
end for
```

- translate_gene (self, gene, negative=False). Translate gene from binary to decimal.

Algorithm 19 translate_gene (self, gene, negative=False)

```
gene_list ← list(gene)
gene_len ← len(gene)
value ← 0
for i in range(gene_len) do
    value+ = pow(2, i) ∗ int(gene_list[i])
end for
if negative then
    value− = pow(2, gene_len − 1)
end if
return value
```

- translate_specie (self, specie). Translate a binary specie to decimal specie.

Algorithm 20 translate_specie (self, specie)

```
dna ← specie[dna]
reward_1 ← [ ]
reward_2 ← [ ]
cur ← 0
for i in range(10) do
    reward_1.append(self.translate_gene(
    dna[cur : cur + 7], negative = True))
    cur+ = 7
end for
for i in range(10) do
    reward_2.append(self.translate_gene(
    dna[cur : cur + 7], negative = True))
    cur+ = 7
end for
action_1 ← [ ]
action_2 ← [ ]
for i in range(10) do
    action_1.append(self.translate_gene(
    dna[cur : cur + 7], negative = True))
    cur+ = 7
end for
for i in range(10) do
    action_2.append(self.translate_gene(
    dna[cur : cur + 7], negative = True))
    cur+ = 7
end for
stable_weight ← self.translate_gene(
dna[cur : cur + 8], negative = False)
cur+ = 8
front_1 ← self.translate_gene(
dna[cur : cur + 6], negative = True)
cur+ = 6
front_2 ← self.translate_gene(
dna[cur : cur + 6], negative = True)
```

```
cur+ = 6
num_1 ← self.translate_gene(
dna[cur : cur + 7], negative = True)
cur+ = 7
num_2 ← self.translate_gene(
dna[cur : cur + 7], negative = True)
cur+ = 7
assertcur == 314
```

(Now, based on symmetry of chessboard, we can get reward_matrix_1, reward_matrix_2, action_matrix_1, action_matrix_2 from reward_1, reward_2, action_1 and action_2, the process is ignored here.)

```
return
    {reward_matrix_1 : reward_matrix_1,
    reward_matrix_2 : reward_matrix_2,
    action_matrix_1 : action_matrix_1,
    action_matrix_2 : action_matrix_2,
    front_1 : front_1,
    front_2 : front_2,
    num_1 : num_1,
    num_2 : num_2}
```

- translate_population (self). The method is to translate the population to decimal population.

Algorithm 21 translate_population (self)

```
self.trans_population_list.clear()
for specie in self.population_list do
    self.trans_population_list.append
    (self.translate_specie(specie))
end for
```

- fitness (self). The method is to calculate the fitness of each specie in population_list.

Algorithm 22 fitness (self)

```
random.shuffle(self.population_list)
self.translate_population()
cores ← multiprocessing.cpu_count()
withmultiprocessing.Pool
(processes = cores) as p
result ← p.map(self._compete,
range(self.population_size))
for i in range(self.population_size) do
    self.population_list[i][fitness] ← result[i][0]
    self.population_list[i][win] ← result[i][1]
    self.population_list[i][lose] ← result[i][2]
    self.population_list[i][draw] ← result[i][3]
end for
self.population_list ← sorted(
self.population_list, key =
lambdas : s[fitness], reverse = True)
```

- selection (self). The method is to select half excellent population.

Algorithm 23 selection (self)

```
self.fitness()
self.population_list ← self.population_list[:
self.population_size//2]
self.population_size// = 2
assertself.population_size =
len(self.population_list)
for specie in self.population_list do
    specie[generation]+ = 1
end for
```

- _compete(self, specie_index, search_depth=1). The method is to do competing within species and return fitness info.

Algorithm 24 _compete (self)

```
win ← 0
lose ← 0
draw ← 0
win_black ← 0
win_white ← 0
start_time ← time.time()
self_specie ← self.trans_population_list[specie_index]
for i in range(self.population_size) do
    if i! = specie_index then
        oppo_specie ←
        self.trans_population_list[i]
        cur_color ← COLOR_BLACK
        self_color ←
        random(COLOR_BLACK, COLOR_WHITE)
        oppo_color ← −self_color
        game ← Game(8)
        board ←
        game.get_init_board()
        end, winner ←
        game.check_game_end(board)
        while not end do
            if cur_color == self_color then
                self_ai ←
                MinimaxSearchPlayer(game, board,
                self_color, search_depth, self_specie)
                move ← self_ai.get_action()
                board, cur_color ←
                game.get_next_board(board, move, self_color)
            else
                oppo_ai ←
                MinimaxSearchPlayer(game, board,
                oppo_color, search_depth, oppo_specie)
                move ← oppo_ai.get_action()
                board, cur_color ←
                game.get_next_board(board, move, oppo_color)
            end if
            end, winner ←
            game.check_game_end(board)
        end while
        if winner == self_color then
            win+ = 1
            if self_color == COLOR_BLACK
then
```

```
            win_black+ = 1
        else
            win_white+ = 1
        end if
    else if winner == oppo_color then
        lose+ = 1
    else
        draw+ = 1
    end if
  end if
end for
return min(win_white, win_black), win, lose, draw
```

- cross_over (self). The method is to do the cross-over operator to the population.

Algorithm 25 cross_over (self))

```
for i in range(self.population_size) do
    random select
    specie1, specie2 from self.population
    self.population_list.append
    (self._cross_over(specie1, specie2))
end for
self.population_size* = 2
```

- _cross_over (self, specie1, specie2). The method is to do the cross-over operator to the two species.

Algorithm 26 _cross_over (self, specie1, specie2)

```
new_dna ← empty string
dna_1 ← list(specie1[dna])
dna_2 ← list(specie2[dna])
for i in range(len(dna_1)) do
    new_dna+ = random(dna_1[i], dna_2[i])
end for
new_specie ← dna : new_dna, generation : 0
self.mutation(new_specie)
return new_specie
```

- mutation (self, specie). The method is to do the mutation operator to the specie.

Algorithm 27 mutation (self, specie)

```
dna ← list(specie[dna])
for i in range(len(dna)) do
    if random.random() < self.p_mutation then
        if dna[i] == 1 then
            dna[i] ← 0
        else
            dna[i] ← 1
        end if
    end if
end for
```

- train (self, total_generation). The method is main method to start GA algorithm.

Algorithm 28 train (self, total_generation)

```
self.generate_origin_population()
for i in range(total_generation) do
    self.selection()
    self.cross_over()
end for
```

5) AI: AI is a class represents a game agent, which is the main role of the game and interacts with the game mainly through $go()$ method. It is worth nothing that the $go()$ method is effectively decorated with a $timeout\_decorator(4.8)$ that limits the time without a timeout.

- $go(self, chessboard)$. The method is to interact with the game.

Algorithm 29 go (self, chessboard)

```
self.candidate_list.clear()
left_number ← len(np.where(chessboard ==
COLOR_NONE)[0])
if left_number <= 10 then
    legal_moves ←
    self.game.get_legal_moves(chessboard, self.color)
    if legal_movesisempty then
        return
    end if
    self.candidate_list.extend(legal_moves)
    player ←
    MinimaxSearchPlayer(self.game, chessboard,
    self.color, 2, params)
    best_action ← player.get_action()
    self.candidate_list.append(best_action)
    player ← FinalSearch(self.game, self.color)
    player.search(chessboard, self.color, 0)
    if player.best_moveisnotNone then
        self.candidate_list.append(player.best_move)
    end if
else
    legal_moves ←
    self.game.get_legal_moves(chessboard, self.color)
    if legal_movesisempty then
        return
    end if
    self.candidate_list.extend(legal_moves)
    search_depth ← 3
    while True do
        if search_depth >= 10 then
            break
        end if
        player ←
        MinimaxSearchPlayer(self.game, chessboard,
        self.color, search_depth, params)
        best_action ← player.get_action()
        self.candidate_list.append(best_action)
        search_depth+ = 1
    end while
end if
```

III. Empirical verification

A. Dataset

For the Reversed Reversi Game, I do not have a dataset ready to test my code. In addition to the cases in the usability test, I also test the correctness and completeness of my code in two ways else. First, I implement an arena which can support two opponents to play against each other, display the game process and save the log to local file. It can help me detect possible code errors in matches by playing enough games with the AI and random algorithms. To speed up the game and save time, I set the depth of the search tree to one layer, which quickly ended the game without changing my purpose. By playing enough games, I can theoretically detect all the possible anomalies. When an exception is caught, I look at the log for that match to fix the code. Second, I constructed some special chessboard and positions to check the correctness of the code, such as side pieces, corner pieces and the situation where there is only one empty space in the chessboard.

In order to improve the performance of the algorithm, I not only evaluated the performance of the algorithm through the game on the platform of AutoPlay and Playto, but also realized the automatic evaluation and selected the optimal algorithm parameter model by modifying the genetic algorithm. Specifically, the genetic algorithm will run a round robin against all individuals in each generation and evaluate each individual's win rate statistically, saving the individuals with the highest win rate in each generation. After recording a certain generation, I can round robin the optimal individuals of each generation again, so as to select the optimal individuals of these generations and achieve the goal of improving the performance of the algorithm. So even if Playto is turned off, I can still effectively improve my algorithm and improve its performance, thanks to the characteristics of genetic algorithms.

B. Performance measure

I evaluated the performance of the algorithm in a local round robin, which was done on a local computer, Windows 10 Professional Edition with Intel(R) Core(TM) I7-10700K CPU @ 3.80GHz of 8 cores and 16 Threads By adding decorators to the $go()$ function, I limited the maximum time of each step to 4.8 seconds, which is to say, the worst-case time to 4.8 seconds, and my algorithm averaged about 0.7 seconds when agreeing to search 3 levels.

My code has passed the usability test and I was ranked 8[th] in both the points and the round robin about 92.4% win ratio. My algorithm can beat me and my classmates, too.

C. Hyperparameters

Most of the parameters used in my algorithm are obtained by genetic algorithm, and the hyperparameters are only the limited running time and the enabling time of the terminal search algorithm. Since the battle platform limits the single step chess time to 5 seconds, I limit the maximum running time of the algorithm to 4.8 seconds. On the one hand, I consider making full use of the time to search, and on the other hand, I also keep 0.2 seconds to avoid misjudgment timeout. The endgame search algorithm needs to search from the current situation to the end game, so it can only be used at the end of the game. I tuned the endgame search and enabled it in the remaining 12, 10 and 8 steps, and found that 10 moves worked best.

The remaining parameters in the algorithm are derived from the genetic algorithm. Finally, the parameters I uploaded are listed as follows:

- R^1. The reward matrix at the game stage one.

$$\begin{bmatrix} 47 & -35 & 9 & 23 & 23 & 9 & -35 & 47 \\ -35 & -25 & 48 & 25 & 25 & 48 & -25 & -35 \\ 9 & 48 & 6 & -45 & -45 & 6 & 48 & 9 \\ 23 & 25 & -45 & -53 & -53 & -45 & 25 & 23 \\ 23 & 25 & -45 & -53 & -53 & -45 & 25 & 23 \\ 9 & 48 & 6 & -45 & -45 & 6 & 48 & 9 \\ -35 & -25 & 48 & 25 & 25 & 48 & -25 & -35 \\ 47 & -35 & 9 & 23 & 23 & 9 & -35 & 47 \end{bmatrix} \tag{1}$$

- R^2. The reward matrix at the game stage two.

$$\begin{bmatrix} -18 & -22 & 39 & -5 & -5 & 39 & -22 & -18 \\ -22 & 62 & 16 & -41 & -41 & 16 & 62 & -22 \\ 39 & 16 & -10 & -35 & -35 & -10 & 16 & 39 \\ -5 & -41 & -35 & 16 & 16 & -35 & -41 & -5 \\ -5 & -41 & -35 & 16 & 16 & -35 & -41 & -5 \\ 39 & 16 & -10 & -35 & -35 & -10 & 16 & 39 \\ -22 & 62 & 16 & -41 & -41 & 16 & 62 & -22 \\ -18 & -22 & 39 & -5 & -5 & 39 & -22 & -18 \end{bmatrix} \tag{2}$$

- A^1. The action reward matrix at the game stage one.

$$\begin{bmatrix} 55 & 56 & 39 & 25 & 25 & 39 & 56 & 55 \\ 56 & 27 & -11 & 60 & 60 & -11 & 27 & 56 \\ 39 & -11 & 52 & 3 & 3 & 52 & -11 & 39 \\ 25 & 60 & 3 & -41 & -41 & 3 & 60 & 25 \\ 25 & 60 & 3 & -41 & -41 & 3 & 60 & 25 \\ 39 & -11 & 52 & 3 & 3 & 52 & -11 & 39 \\ 56 & 27 & -11 & 60 & 60 & -11 & 27 & 56 \\ 55 & 56 & 39 & 25 & 25 & 39 & 56 & 55 \end{bmatrix} \tag{3}$$

- A^2. The action reward matrix at the game stage two.

$$\begin{bmatrix} -22 & 12 & 43 & 56 & 56 & 43 & 12 & -22 \\ 12 & -51 & 49 & 41 & 41 & 49 & -51 & 12 \\ 43 & 49 & 55 & 36 & 36 & 55 & 49 & 43 \\ 56 & 41 & 36 & -60 & -60 & 36 & 41 & 56 \\ 56 & 41 & 36 & -60 & -60 & 36 & 41 & 56 \\ 43 & 49 & 55 & 36 & 36 & 55 & 49 & 43 \\ 12 & -51 & 49 & 41 & 41 & 49 & -51 & 12 \\ -22 & 12 & 43 & 56 & 56 & 43 & 12 & -22 \end{bmatrix} \tag{4}$$

- W_s: 202. The weight of difference between stable pieces.
- W_f^1: -52. The weight of difference between frontier pieces at the game stage one.
- W_f^2: -27. The weight of difference between frontier pieces at the game stage two.

- W_p^1: -44. The weight of difference between pieces number at the game stage one.
- W_p^2: -51. The weight of difference between pieces number at the game stage two.

D. Experimental results

I have passed all the 10 test cases in the usability test. After that, in the point race I got 8$^{\text{th}}$ rank. And in the last round robin I also got 8$^{\text{th}}$ rank.

In order to save time, I applied for a 40-core node to run genetic algorithm on Taiyi cluster. See Fig. 1. Log files are saved during the run, as shown in Fig. 2. In the case of population size of 60, the running speed is about 1 minute per generation.

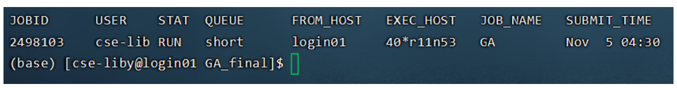

Fig. 1. Single node with 40 cores

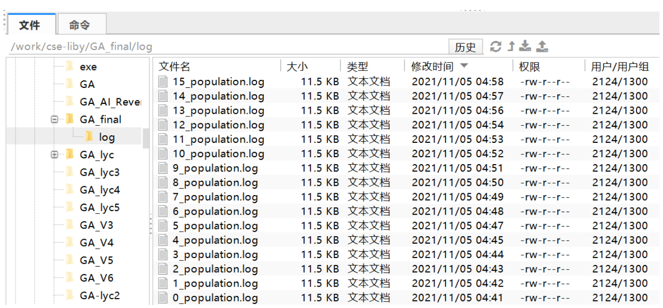

Fig. 2. The log files

To test the effectiveness of the genetic algorithm, I saved individual data of 300 generations, and the top 30 optimal individuals in each generation were saved to the local log file, as shown in Fig. 3. Now, I randomly select an individual from each generation as a representative, and play 6 games with the random algorithm, including 3 black games and 3 white games, and calculate the corresponding winning rate as the level indicator of this generation. As shown in Fig. 4, we can see that with the increase of genetic algebra, the level of individuals is constantly improving.

E. Conclusion

1) Advantages and disadvantages of my algorithm: In this project, I used Minimax combined with Alphabeta pruning as my decision algorithm, and used genetic algorithm to provide good evaluation function (evaluation parameters) for decision algorithm. Through the combination of the two, high performance can be achieved.

As an important search algorithm in zero-sum games, Minimax algorithm has been significantly used in many games, and

Fig. 3. The log file content

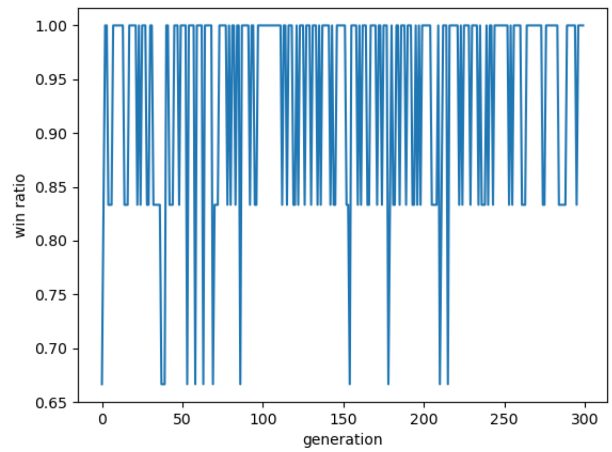

Fig. 4. The win ratio

Alpha-beta pruning can speed up the former search efficiency. However, the quality of the algorithm largely depends on the search depth and evaluation function. Generally speaking, the deeper the search depth, the better the evaluation function and the higher the performance of the algorithm. The setup of evaluation functions is often difficult. The advantage of my algorithm is that the evaluation function is obtained by using genetic algorithm without the need for additional human participation, which largely overcomes the difficulties and problems caused by the evaluation function.

The disadvantage of the algorithm is that due to the limitation of single step time, the searchable depth of the search tree is affected by the algorithm implementation and language efficiency, and efficient language and excellent algorithm optimization can improve the process.

2) The experience I learned: In this project, I learned to use Minimax combined with Alphabeta pruning algorithm to solve decision-making problems in zero-sum games. At the same time, in the process of continuous thinking and improvement of the algorithm, I found that genetic algorithm could solve the problem of setting evaluation function well. However, the efficiency of the initial version of genetic algorithm was very low, so I used Python multi-process module for parallel operation, which greatly accelerated the evolution speed.

In addition, IN the genetic algorithm, I also tried a variety of evaluation methods, such as checkerboard terrain evaluation, stability sub evaluation, frontier sub evaluation, number of pieces evaluation, and phased evaluation, and finally made the algorithm performance significantly improved.

3) Deficiencies: Although the evaluation function in the search algorithm was cleverly solved by the genetic algorithm, it was ultimately limited by the platform running time, and my algorithm had a very limited search depth of about 3-4 layers by mid-game. The efficiency of the search algorithm itself is less optimized, if the replacement table, hash and other ways, it is possible to improve the search efficiency of the algorithm, to achieve a deeper search depth.

4) Possible directions of improvements: At present, the algorithm is divided into two stages of parameters, the value of each parameter is 7-8 binary, representing a limited range of values. I suspect that the ability to evaluate functions could be further improved by dividing up more stages of the game and increasing the range of values for each parameter. Alternatively, a set of evaluation functions that dynamically change with the number of pieces in a game is an important direction for improvement.

REFERENCES

[1] S. Mirjalili, *Genetic Algorithm*. Cham: Springer International Publishing, 2019, pp. 43–55. [Online]. Available: https://doi.org/10.1007/978-3-319-93025-1_4

[2] Wikipedia contributors, "Minimax — Wikipedia, the free encyclopedia," 2021, [Online; accessed 4-November-2021]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1048398851

[3] M. Buro, "An evaluation function for othello based on statistics," *Technical Report 31, NEC Research Institute*, 1997.

[4] Wikipedia contributors, "Zero-sum game — Wikipedia, the free encyclopedia," 2021, [Online; accessed 4-November-2021]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Zero-sum_game&oldid=1050040981