# AI Algorithm for Reversed Reversi

Yichen Li
*Department of Computer Science*
*Southern University of Science and Technology*
11912433
11912433@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.

This goal of the project is to help us learn and practice game-tree searching. This project requires us to find better evaluation function and optimize Minimax algorithm to search more depths.

The algorithms I have tried in this project are AlphaZero, Minimax with alpha-beta pruning. At the beginning, I used AlphaZero algorithm with different networks but I gave up with the reason of time and space constraints [1]. Then I used Minimax algorithm with alpha-beta pruning. To train better evaluation function, I also used Genetic Algorithm(GA). [2]

This project is written in Python with IDE PyCharm. The local hardware environment is Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz 3.79 GHz. The training environment is Taiyi(40 cores/node). The running platform is a server provided by the teachers.

### B. Problem applications

As we all know, reversi is a popular game around the world. Meanwhile, the research of AI for Reversi has a long history. And Reversed Reversi is very similar to reversi. Our research of Reversed Reversi can help to optimize old algorithms and explore new game-tree searching algorithms on the basis of previous studies.

## 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 chess color of my self. 1 for white, $-1$ for black, 0 for none.
- $c_R$: The color of root node.
- $C_R$: The children node of root node stored in MinimaxPlayer.
- $D$: The difference piece number between mine and opponent.
- $d$: The search depth in Minimax algorithm.
- $INF$: Infinity.
- $L_c$: The length of chrome.
- $L_g$: The length of gene.
- $F_r$: Relative frontier.
- $F_{self}$: My frontier.
- $F_{oppo}$: The opponent's frontier.
- $M_r$: Relative mobility.
- $M_{self}$: My mobility.
- $M_{oppo}$: The opponent's mobility.
- $M_P$: Minimax player.
- $N$: The total number of chess.
- $N_{self}$: The number of my chess.
- $N_{oppo}$: The number of opponent's chess.
- $nINF$: Negative infinity.
- $P$: The period of current chessboard, it is 0 when $N < 35$, otherwise it is 1.
- $P_m$: The probability of gene mutation.
- $pop$: The population dictory.
- $R$: The root node stored in MinimaxPlayer with current board, color, depth is 0, search type is MAX_SEARCH, $\alpha$ is $nINF$, $\beta$ is $INF$, and value is $nINF$.
- $S_r$: Relative stability.
- $S_{self}$: My stablility.
- $S_{oppo}$: The opponent's stability.
- $S_P$: Search Player.
- $S_{pop}$: The population size.
- $S_{select}$: The selection size.
- $SW_r$: Relative weigt map sum.
- $SW_{self}$: My weight map sum.
- $SW_{oppo}$: The opponent's weight map sum.
- $T$: The training time.
- $V$: The list of legal moves.
- $V_R$: The list of legal moves of root node store in MinimaxPlayer.
- $W\_stab$: The weight of stability in evaluate function.
- $W\_front\_start$: The weight of frontier in evaluate function at the beginning.
- $W\_front\_end$: The weight of frontier in evaluate function in the end.
- $W\_diff\_start$: The weight of difference in evaluate function at the beginning.
- $W\_diff\_start$: The weight of difference in evaluate

function at the beginning.

## B. Data structure

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

- candidate_list (list): a list of the legal moves.
- directions (list): a list of eight directions $[(1,1), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1), (0,1)]$.
- V_map_start (list): a list of weight map at the beginning.
- V_map_end (list): a list of weight map in the end.
- mobility_start (list): a list of mobility map at the beginning.
- mobility_end (list): a list of mobility map in the end.

Here are the details of data:

- Vmap_start. The weight map at the beginning.

$$\begin{bmatrix} -17 & -43 & 25 & 23 & 23 & 25 & -43 & -17 \\ -43 & -25 & 56 & 25 & 25 & 56 & -25 & -43 \\ 25 & 56 & 7 & -45 & -45 & 7 & 56 & 25 \\ 23 & 25 & -45 & -49 & -49 & -45 & 25 & 23 \\ 23 & 25 & -45 & -49 & -49 & -45 & 25 & 23 \\ 25 & 56 & 7 & -45 & -45 & 7 & 56 & 25 \\ -43 & -25 & 56 & 25 & 25 & 56 & -25 & -43 \\ -17 & -43 & 25 & 23 & 23 & 25 & -43 & -17 \end{bmatrix} \tag{1}$$

- Vmap_end. The weight map in the end.

$$\begin{bmatrix} 46 & -40 & -5 & -9 & -9 & -5 & -40 & 46 \\ -40 & -2 & 1 & -3 & -3 & 1 & -2 & -40 \\ -5 & 1 & 54 & -35 & -35 & 54 & 1 & -5 \\ -9 & -3 & -35 & 21 & 21 & -35 & -3 & -9 \\ -9 & -3 & -35 & 21 & 21 & -35 & -3 & -9 \\ -5 & 1 & 54 & -35 & -35 & 54 & 1 & -5 \\ -40 & -2 & 1 & -3 & -3 & 1 & -2 & -40 \\ 46 & -40 & -5 & -9 & -9 & -5 & -40 & 46 \end{bmatrix} \tag{2}$$

- mobility_start. The mobility map at the beginning.

$$\begin{bmatrix} 47 & 56 & -25 & 9 & 9 & -25 & 56 & 47 \\ 56 & 58 & -45 & 48 & 48 & -45 & 58 & 56 \\ -25 & -45 & 36 & 33 & 33 & 36 & -45 & -25 \\ 9 & 48 & 33 & -45 & -45 & 33 & 48 & 9 \\ 9 & 48 & 33 & -45 & -45 & 33 & 48 & 9 \\ -25 & -45 & 36 & 33 & 33 & 36 & -45 & -25 \\ 56 & 58 & -45 & 48 & 48 & -45 & 58 & 56 \\ 47 & 56 & -25 & 9 & 9 & -25 & 56 & 47 \end{bmatrix} \tag{3}$$

- mobility_end. The mobility map in the end.

$$\begin{bmatrix} 34 & 41 & 43 & 56 & 56 & 43 & 41 & 34 \\ 41 & -52 & 36 & -24 & -24 & 36 & -52 & 41 \\ 43 & 36 & -9 & 5 & 5 & -9 & 36 & 43 \\ 56 & -24 & 5 & -36 & -36 & 5 & -24 & 56 \\ 56 & -24 & 5 & -36 & -36 & 5 & -24 & 56 \\ 43 & 36 & -9 & 5 & 5 & -9 & 36 & 43 \\ 41 & -52 & 36 & -24 & -24 & 36 & -52 & 41 \\ 34 & 41 & 43 & 56 & 56 & 43 & 41 & 34 \end{bmatrix} \tag{4}$$

- stability_weight: 218
- frontier_weight_start: -52
- frontier_weight_end: -27
- diff_weight_start: -52
- diff_weight_end: -51

## C. Model design

Based on the given chessboard, each player takes turns to do a move until the game is end. each move is get by function $go()$ to find all legal moves, and figure out the best move in $candidate\_list$.

To find all legal moves, I use the function $get\_legal\_moves()$ in $go()$. It judges all positions of chessboard by The function $is\_legal\_move()$, which is to judge whether a move is a legal move.

To get the best move, I consider to divide it into two part by the piece number on the board. The first part is before 54 pieces while the second part is after 54 pieces, for the reason that I can search to the end of the game in last 10 layers during 5 second.

The first part is Minimax algorithm called class $MinimaxPlayer$. I design a loop to make full use of 5 second. In each loop, the search depth will increase, I will call $get\_action()$ in class $MinimaxPlayer()$ to get the best move. In $get\_action()$, $minimax()$ is to search for the given search depth. Function $getValue()$ in $minimax()$ is to get value of node by $getMapWeightSum()$, $get\_stability()$, $get\_mobility()$, $get\_frontier()$, $get\_diff()$. These functions are used to get the sum of each location based on static weight table, get stability of current player and current chessboard, get mobility of current player and current chessboard, get frontier of current player and current chessboard and get piece number difference between current player and other player respectively.

- Static weight squares:

$$SW_r = SW_{self} - SW_{oppo} = \sum_{(i,j)} W(i,j) \cdot C(x,y) \cdot c_{self}$$

- Relative stability:

$$S_r = -S_{self} + S_{oppo}$$

- Relative mobility:

$$M_r = M_{self} - M_{oppo}$$

- Relative Frontier:

$$F_r = F_{self} - F_{oppo}$$

- Absolute difference:

$$D = N_{self} - N_{oppo}$$

And the formula of the evaluation function is:

$$E(L_i) = \sum C_i \cdot W_i = C_w \cdot SW_r + C_m \cdot M_r + C_f \cdot F_r + C_d \cdot D + C_s \cdot S_r$$

The second part is the combination of Minimax and final search. In the last 10 layers, current chessboard can be searched to the end of the game based on test on running platform. Thus, In this part, I first let the class $MinimaxPlayer$ use funtion $minimax()$ to search for 2 depth. Then I use class $SearchPlayer$ to call functiion $search()$ to implement final search algorithm. Then I can get the best move by $best\_move$ stored in class $SearchPlayer$.

The algorithm to get the location based weight table and weights of stability, mobility, frontier and difference is Genetic Algorithm(GA). $genetic_algorithm()$ in class $GA$ will simulate $selection()$ and $crossover()$. In function $selection()$, it simulates $compete()$ in its population and sorts by wins to select half of population size. In function $compete()$ I call $do\_compete()$ to implement multiprocessing. To maintain population size, in function $crossover()$, it calls function $do_crossover()$ to simulate chromatid exchanges and calls function $mutation()$ to simulate gene mutation. To simulate gene, I convert decimal data to binary. Funtion $transform()$ uses function $transform\_gene()$ to transform all binary data into decimal. During every loop of evolution, I output every individual's information in log directory. Thus, I can select one of them as my final evaluation data.

## D. Details of algorithms

Here are the details of algorithms mentioned above:

- go (cb). The main function to generate a candicate list, which is a list of legal moves and the best move is at the last of the list. The input "cb" represents for the current chessboard.

**Algorithm 1** go(cb)

```
1:  $V \leftarrow get\_legal\_moves(cb, c_{self})$
2:  if $len(V) = 0$ then
3:      return
4:  end if
5:  $candicate\_list \leftarrow V$
6:  if $N < 54$ then
7:      $d \leftarrow 3$
8:      while $True$ do
9:          if $d >= 20$ then
10:             break
11:         end if
12:         create MinimaxPlayer $M_P$ with $d$
13:         $move \leftarrow M_P.get\_action()$
14:         append $move$ into $candidate\_list$
15:         $d \leftarrow d + 1$
16:     end while
17: else
18:     $d \leftarrow 2$
19:     create MinimaxPlayer $M_P$ with $d$
20:     $move \leftarrow M_P.get\_action()$
21:     append $move$ into $candidate\_list$
22:     create SearchPlayer $S_P$
23:     $search(cb, c_{self}, 0)$
24:     $move \leftarrow S_P.best\_move$
25:     if $move$ then
26:         append $move$ into $candidate\_list$
27:     end if
28: end if
```

- get_legal_moves(cb,c). To generate legal moves of current player. The input "cb" represents for the current chessboard, "c" for the current player color, and return the list of legal moves.

**Algorithm 2** get_legal_moves(cb,c)

```
1: $moves \leftarrow []$
2: for $i \leftarrow 1$ to $7$ do
3:     for $j \leftarrow 1$ to $7$ do
4:         if $is\_legal\_move(cb, c, (i, j))$ then
5:             append $i$ into $moves$
6:         end if
7:     end for
8: end for
9: return $moves$
```

- is_on_board(x, y). To judge whether the move (x,y) is on the chessboard. The input "(x,y)" represents for the move to make, and returns true if it is on the board, otherwise returns false.

**Algorithm 3** is_on_board(x, y)

```
1: return $0 <= x < 8$ and $0 <= y < 8$
```

- is_legal_move(cb, c, move). To judge whether the move (x,y) is a legal move. The input "cb" represents for the current chessboard, "c" for the current player color, "(x,y)" for the move to make, and returns true if it is legal, otherwise returns false.

**Algorithm 4** is_legal_move(cb, c, move)

```
1:  $x\_s, y\_s \leftarrow move$
2:  if not $(is\_on\_board(x\_s, y\_s)$ and $cb(x\_s, y\_s) = c)$ then
3:      return False
4:  end if
5:  for $x\_d, y\_d$ in directions do
6:      $x \leftarrow x\_s + x\_d$
7:      $y \leftarrow y\_s + y\_d$
8:      if not $is\_on\_board(x, y)$ or $cb[x][y]! = -c$ then
9:          continue
10:     end if
11:     $x \leftarrow x + x\_d$
12:     $y \leftarrow y + y\_d$
13:     while $is\_on\_board(x, y)$ do
14:         if $cb[x][y] = 0$ then
15:             break
16:         else if $cb[x][y] = c$ then
17:             return True
18:         end if
19:         $x \leftarrow x + x\_d$
20:         $y \leftarrow y + y\_d$
```

```
21:         end while
22:     end for
23:     return False
```

- is_game_end(cb). To judge whether the game is over. The input “cb” represents for the current chessboard, and returns (end, winner), end is true is game is over, winner is the player color.

**Algorithm 5** is_game_end(cb)

```
1: if $len(get\_legal\_moves(cb, 1))$ != $0$ and $len(get\_legal\_moves(cb, -1))! = 0$ then
2:     return $0, 0$
3: end if
4: $cnt \leftarrow np.sum(cb)$
5: if $cnt > 0$ then
6:     return $1, -1$
7: else if $cnt < 0$ then
8:     return $1, 1$
9: else
10:     return $1, 0$
11: end if
```

- get_next_state(cb, c, move). To get the next board and player. The input “cb” represents for the current chessboard, “c” for the current player color, “move” for the move to make, and returns next board and next color.

**Algorithm 6** get_next_state(cb, c, move)

```
1: $b \leftarrow np.copy(cb)$
2: $next\_c \leftarrow do\_move(b, move, c)$
3: return $b, next\_c$
```

- do_move(cb, move, c). To do the move on chessboard. The input “cb” represents for the current chessboard, “move” for the move to make, “c” for the current player color,and returns next color.

**Algorithm 7** do_move(cb, move, c)

```
1: $rv\_list \leftarrow get\_reverse\_list(cb, move, c)$
2: append $move$ into $rv\_list$
3: for $i, j$ in $rv\_list$ do
4:     $cb[i][j] \leftarrow c$
5: end for
6: if $len(get\_legal\_moves(cb, -c)) = 0$ then
7:     return $c$
8: else
9:     return $-c$
10: end if
```

- get_reverse_list(cb, move, c). To get the reverse list. The input “cb” represents for the current chessboard, “move” for the move to make, “c” for the current player color,and returns next color.

**Algorithm 8** get_reverse_list(cb, move, c)

```
1: $x\_s, y\_s \leftarrow move$
2: $rv\_set \leftarrow set()$
3: for $x\_d, y\_d$ in directions do
4:     $x \leftarrow x\_s + x\_d$
5:     $y \leftarrow y\_s + y\_d$
6:     if not $is\_on\_board(x, y)$ or $cb[x][y]! = -c$ then
7:         continue
8:     end if
9:     $rv\_temp\_list \leftarrow [(x, y)]$
10:     $x \leftarrow x + x\_d$
11:     $y \leftarrow y + y\_d$
12:     while $is\_on\_board(x, y)$ do
13:         if $cb[x][y] = 0$ then
14:             break
15:         else if $cb[x][y] = -c$ then
16:             append $(x, y)$ into $rv\_temp\_list$
17:             $x \leftarrow x + x\_d$
18:             $y \leftarrow y + y\_d$
19:         else
20:             update $rv\_temp\_list$ to $rv\_set$
21:             break
22:         end if
23:     end while
24: end for
25: return $list(rv\_set)$
```

- get_init_board(). To get the init board. It returns an init board.

**Algorithm 9** get_init_board()

```
1: $b \leftarrow np.zeros(8, 8)$
2: $b[3][3] \leftarrow 1$
3: $b[3][4] \leftarrow -1$
4: $b[4][3] \leftarrow -1$
5: $b[4][4] \leftarrow 1$
6: return $b$
```

- get_action(). To get the best move from MinimaxPlayer. It returns a best move.

**Algorithm 10** get_action()

```
1: $minimax(R)$
2: $max\_v \leftarrow nINF$
3: $best\_move \leftarrow V_R[0]$
4: for $i, chld$ in $enumerate(C_R)$ do
5:     if $chld.value > max\_v$ then
6:         $max\_v \leftarrow chld.value$
7:         $best\_move \leftarrow V_R[i]$
8:     end if
9:     if $chld.value = INF$ then
10:         return $best\_move$
11:     end if
12: end for
13: return $best\_move$
```

- minimax(node). To execute Minimax algorithm to find the best point of the next move. The input “node” represents for node with its board, color, depth, search type, $\alpha$, $\beta$ and value, and returns the value of node.

**Algorithm 11** minimax(node)

```
$end, winner \leftarrow is\_game\_end(node.board)$
if end then
    $node.value \leftarrow INF * winner * c_R$
    return $node.value$
end if
if $node.depth = d$ then
    $node.value \leftarrow getValue(node)$
    return $node.value$
end if
$V \leftarrow get\_valid\_moves(node.board, node.color)$
if $node.depth = 0$ then
    $V_R \leftarrow V$
end if
if $len(V) = 0$ then
    if $node.search\_type = MAX\_SEARCH$ then
        $next\_v \leftarrow INF$
    else
        $next\_v \leftarrow nINF$
    end if
    create next Node subNode with
    $node.board, -node.color,$
    $node.depth + 1, -node.search\_type,$
    $node.\alpha, node.\beta, next\_v$
    if $node.depth = 0$ then
        append $subNode$ into $C_R$
    end if
    $node.value \leftarrow minimax(subNode)$
    return $node.value$
end if
for $move$ in $V$ do
    $next\_b, \_ \leftarrow get\_next\_state(node.board,$
    $node.color, move)$
    if $node.search\_type = MAX\_SEARCH$ then
        $next\_v \leftarrow INF$
    else
        $next\_v \leftarrow nINF$
    end if
    create next Node subNode with
    $node.board, -node.color,$
    $node.depth + 1, -node.search\_type,$
    $node.\alpha, node.\beta, next\_v$
    if $node.depth = 0$ then
        append $subNode$ into $C_R$
    end if
    $utility \leftarrow minimax(subNode)$
    if $utility = INF$ then
        $node.value \leftarrow INF$
        return INF
    end if
    if $node.search\_type = MAX\_SEARCH$ then
        $node.value \leftarrow max(node.value, utility)$
        if $node.value >= node.\beta$ then
            return $node.value$
        end if
        $node.\alpha \leftarrow max(node.value, node.\alpha)$
    else
        $node.value \leftarrow min(node.value, utility)$
        if $node.value <= node.\alpha$ then
            return $node.value$
        end if
        $node.\beta \leftarrow min(node.value, node.\beta)$
    end if
end for
return $node.value$
```

- getValue(node). To evaluate the value of node. The input "node" represents for the node to evaluate, and returns the value of node.

**Algorithm 12** getValue(node)

```
$b \leftarrow node.board$
$mapWeightSum \leftarrow getMapWeightSum(b, c_R)$
$stability \leftarrow (get\_stability(b, -c_R) - get\_stability(b, c_R)) * W\_stab$
$mobility \leftarrow get\_mobility(b, c_R)$
if $period = 0$ then
    $frontier \leftarrow (get\_frontier(b, c) - get\_frontier(b, -c)) * W\_front\_start$
    $diff \leftarrow get\_diff(b, c) * W\_diff\_start$
else
    $frontier \leftarrow (get\_frontier(b, c) - get\_frontier(b, -c)) * W\_front\_end$
    $diff \leftarrow get\_diff(b, c) * W\_diff\_end$
end if
return $mapWeightSum + stability + mobility + frontier + diff$
```

- getMapWeightSum(board, color). Calculate the weightmap dot board dot color.

- get_stability(board, color). Calculate the sum of the color piece that can not be reversed at any time. Therefore, it always exits at corner and edges.

- get_mobility(board, color). Calculate the length of the color player's legal moves.

- get_frontier(board, color). Calculate the sum of the color piece that has no piece in its eight directions.

- get_diff(board, color). Calculate the difference between the color piece number and the opponent's piece number.

- search(cb, c, depth). To final search to the end and find winning strategy. The function input "cb" represents for the current chessboard, and "c" for the current player color returns the value that calculated as the metrics mentioned in the Model design.

**Algorithm 13** search(cb, c, depth)

```
end, winner ← $is\_game\_end(node.board)$
if end then
    if $winner = c_R$ then
        return 1
    else if $winner = 0$ then
        return 0
    else
        return -1
    end if
end if
$V \leftarrow get\_valid\_moves(b, c)$
$return\_v \leftarrow []$
for $move$ in $V$ do
    $next\_b, next\_c \leftarrow get\_next\_state(b, c, move)$
    $value \leftarrow search(nexr\_b, next\_c, depth + 1)$
    append $value$ into $return\_v$
    if $c = c_R$ then
        if $value = 1$ then
            if $depth = 0$ then
                $best\_move \leftarrow move$
            end if
            return 1
        end if
    else
        if $value = -1$ then
            return -1
        end if
    end if
end for
for $i \leftarrow 0$ to $len(return\_v)$ do
    if $return\_v[i] = 0$ then
        if $c = c_R$ and $depth = 0$ then
            $best\_move \leftarrow V[i]$
        end if
        return 0
    end if
end for
if $c = c_R$ then
    return -1
else
    return 1
end if
```

- genetic_algorithm(). To train the population with GA.

**Algorithm 14** genetic_algorithm()

```
$pop \leftarrow []$
for $i \leftarrow 0$ to $S_{pop} - 1$ do
    $idv \leftarrow dict()$
    $chr \leftarrow []$
    for $j \leftarrow 0$ to $L_c - 1$ do
        $gene \leftarrow []$
        for $k \leftarrow 0$ to $L_g - 1$ do
            append $random.randint(0, 1)$ to $gene$
        end for
        append $gene$ to $chr$
    end for
    $idv[\text{“}chrome\text{”}] \leftarrow chr$
    $idv[\text{“}generation\text{”}] \leftarrow 1$
    $transform(idv)$
    append $idv$ into $pop$
end for
for $i \leftarrow 0$ to $T - 1$ do
    $selection()$
    $crossover()$
end for
```

- transform(individual). Transform the binary data into decimal and store them in the population dict.

- selection(). To simulate selection.

**Algorithm 15** selection()

```
$compete()$
$pop \leftarrow sorted(pop, key = lambdaidv : idv[\text{“}win\text{”}], reverse = True)$
$pop \leftarrow pop[: S_{select}]$
for $idv$ in $pop$ do
    $idv[\text{“}generation\text{”}] \leftarrow idv[\text{“}generation\text{”}] + 1$
end for
```

- compete(). To simulate competition between population.

**Algorithm 16** compete()

```
for $i \leftarrow 0$ to $S_{pop} - 1$ do
    $win, lose, draw \leftarrow do\_compete(i)$
    $pop[i][\text{“}win\text{”}] \leftarrow win$
    $pop[i][\text{“}lose\text{”}] \leftarrow lose$
    $pop[i][\text{“}draw\text{”}] \leftarrow draw$
end for
```

- do_compete(idv_i). To simulate competition between one individual and population. The input idv_i is the index of the individual to compete with population, and returns the individual's win, lose and draw times.

**Algorithm 17** do_compete()

```
$win \leftarrow 0$
$lose \leftarrow 0$
$draw \leftarrow 0$
for $i \leftarrow 0$ to $S_{pop} - 1$ do
    if $i! = idv\_i$ then
        if $random.randint(0, 1) = 1$ then
            $c \leftarrow 1$
        else
            $c \leftarrow -1$
        end if
        $b \leftarrow get\_init\_board()$
        $end, winner \leftarrow is\_game\_end(b)$
        $cur\_c \leftarrow -1$
        while $notend$ do
            create a player $p$ with $M_P$
            $cur\_c \leftarrow do\_move(b,$
            $p.get\_action(), cur\_c)$
```

```
18:         $end, winner \leftarrow is\_game\_end(b)$
19:       **end while**
20:       **if** $winner = c$ **then**
21:         $win \leftarrow win + 1$
22:       **else if** $winner = -c$ **then**
23:         $lose \leftarrow lose + 1$
24:       **else**
25:         $draw \leftarrow draw + 1$
26:       **end if**
27:     **end if**
28: **end for**
29: **return** $win, lose, draw$
```

- crossover(). To simulate crossover between population.

**Algorithm 18** crossover()

```
1: **for** $i \leftarrow 0$ to $S_{select} - 1$ **do**
2:   $j \leftarrow np.random.choice(range(S_{select}))$
3:   **while** $i = j$ **do**
4:     $j \leftarrow np.random.choice(range(S_{select}))$
5:   **end while**
6:   $new\_idv \leftarrow do\_crossover(pop[i], pop[j])$
7:   append $new\_idv$ into $pop$
8: **end for**
```

- do_crossover(idv1, idv2). To simulate crossover between individuals. The input "idv1" and "idv2" represent for two individual to crossover, and returns new individual.

**Algorithm 19** do_crossover(idv1, idv2)

```
1: $new\_idv \leftarrow dict()$
2: $chr \leftarrow []$
3: **for** $i \leftarrow 0$ to $L_c - 1$ **do**
4:   $x \leftarrow random.randint(0, 1)$
5:   **if** $x = 0$ **then**
6:     append $mutation(idv1[`chrome'][i])$ into $chr$
7:   **else**
8:     append $mutation(idv2[`chrome'][i])$ into $chr$
9:   **end if**
10: **end for**
11: $new\_idv[“chrome”] \leftarrow chr$
12: $new\_idv[“generation”] \leftarrow 1$
13: $transform(new\_idv)$
14: **return** new_idv
```

- mutation(gene). To simulate gene mutation. The input "gene" represents for the gene to mutation, and returns new_gene.

**Algorithm 20** mutation(gene)

```
1: $new\_gene \leftarrow []$
2: **for** $i \leftarrow 0$ to $L_g - 1$ **do**
3:   **if** $random.random() < P_m$ **then**
4:     **if** $gene[i] = 0$ **then**
5:       append 1 into $new\_gene$
6:     **else**
7:       append 0 into $new\_gene$
8:     **end if**
9:   **else**
10:    append $gene[i]$ into $new\_gene$
11:  **end if**
12: **end for**
13: **return** $new\_gene$
```

## III. Empirical verification

### A. Dataset

In the usability test, I not only use the test cases provided, but also design some other cases to test my code. At the first time, after I wrote the game code, I designed two random players to test whether my game logic code is right. Both sides of the game get legal moves $V$ and directly do move on the $V[0]$. During the game, I can check the legal moves, reverse chess on the board and the final winner to judge my game logic.

After finishing all algorithms, during the training of my evaluation function, I let my AI fight to the random player to observe whether my evaluation function is becoming better, which is a great method to check my GA, minimax and evaluation.

During the training, I observe my evaluation function and judge it artificially. For example, it is obvious that the stability is especially important, and the frontier weight is negative when it is $F_{self} - F_{oppo}$. If my stability weight is always small or frontier weight is always positive, my algorithm may have problems.

At the last, I submit my code to the platform and use "Playto" to improve my evalution function. For example, I try different evaluation matrix to pick the best rank one. Or I change my evaluation object, such as using sigmoid instead of dividing the game into begin and end to compare their performance. Besides, I also use platform to test how depth I can search to without overtime. so that I can find my final search limit to get better. During each game, I will download my log details and try to find out a better strategy and find my AI's shortages, so that I can better improve my AI.

After the "Playto" function is disabled, I will let my different AI do compete and choose the best one. Genetic Algorithm has already included compete, so I can quickly get the highest winning ratio or longest survival generation individual in each generation. Besides, I will compete with my classmates and random player to sum up experience. I can also let it play to my AlphaZero to compare their performance.

### B. Performance measure

I test my AI on my computer and Taiyi platform. I let it do compete with random player, AlphaZero, different evaluation function and classmates. My computer test environment is Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz 3.79 GHz with 8 cores and 16 threads. Taiyi platform test environment is 40 cores.

I test my AI on platform to find the maximum depth I can search. I find that when the depth is 3, it will always be on time. But when depth is 4, it will sometimes face overtime. So I use a loop to increase depth gradually to make full use of time and the average of each turn is 4 seconds. And I also find that the last 10 layers I can search to the end without overtime, so I can use final search in the last 10 steps. Apart from these, I also use timeout_decorator to control my AI's time.

I have passed 10 test cases in the usability test. My rank of code in the round robin is 3.

## C. Hyperparameters

I use the parameters in my evaluation function. In the "Data structure" subsection, I mentioned my final parameters. I use genetic algorithm (GA) to improve my parameters. In each generation, I have 60 populations, different population means different evaluation function. Then 60 populations will do compete with each other, and I use multyprocessing to accelerate the competitions. I will select 30 populations according to their winning numbers. Then the rest 30 populations will crossover with each other, and they may occur gene mutation. After 80 generations, I can enter 30. After 200 generations, I can enter top 10. At the end, I choose 3000 generations, and I enter 3.

## D. Experimental results

I have passed 10 test cases in the usability test. My win_count is 393 and lose_count is 18. My winning persentage on the platform is 95.6%. And the rank in the last round robin I got is 3.

## E. Conclusion

*1) Analysis of the experimental results:* According to my rank in the last round robin, the experimental results are pretty good. My AI can basically defeat ordinary Minimax, but AlphaZero is more than my AI's match on the same condition.

*2) Advantages and disadvantages of my algorithm:* I fully combined GA with Minimax, which makes the evaluation function much better. It is much better than artificial evaluation matrix. Besides, I also consider the weight during different period, which can make evaluation function dynamically. However, My evaluation fuction is still too simple and maybe I can use some nonlinear function instead of different periods such as sigmoid. In addition, I can consider more factors such as parity. If possible, I can increase my population size and selection size, which can increase diversity. Besides, I can optimize my algorithms to improve speed so that I can search for more depths.

*3) The experience I learned:* I need to stabilize my mind when meet failed models and declining ranking and I should be more adhere to my thought. Until the last one week of deadline, I try AlphaZero but due to time and space constraints, I find that although AlphaZero could enter top 10 but still had difficulty in competing with higher-rank Minimax AI. Besides, the time of training a model is very long and the result may be not good. So I gave up optimizing AlphaZero and turned to GA and Minimax. However, the rank in the last round robin turns out that AlphaZero is the best.(AlphaZero ranks 1 and the winning persentage is 99.5%). Therefore, if I have more confidence and immerse myself in optimizing the model and algorithms, I may have better AI.

*4) Deficiencies:* Maybe I can use some nonlinear function to simulate dynamic weight such as sigmoid. And I can consider more aspects such as parity. Besides, I can optimize my algorithms to improve the speed, so that I can search for more depths.

*5) Possible directions of improvements:* Explore better evaluation function, set of period and some other linear or nonlinear functions. Or use another algorithm to get dynamic adjustment strategy of evaluation function.

Learn to optimize AlphaZero algorithm such as Numpy underlying language, model reduction and so on.

## References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[2] K. A. Cherry, "An intelligent othello player combining machine learning and game specific heuristics," 2011.