Artificial Intelligence of Reversed Reversi

November 4, 2021

Project1 Report of

CS303 Artificial Intelligence

11911839 NieYuhe

1 Preliminaries

1.1 Goal

Reversi[1] is a strategy board game for two player, playing game on an 8x8 uncheckered board. Reversed Reversi has almost same rules with reversi but only different in the object of the game is to have the fewest disc turned to display your color when the last playable empty square is filled.

In this project, the goal is to implement the AI algorithm of Reversed Reversi according to the interface requirements and submit it to the system as required for usability test and round robin in one month.

1.2 Algorithms

The algorithms listed below are used in this project:

- *MiniMax Algorithm with A-B pruning*: Main algorithm for the Reversed Reversi AI
- *Heuristic Algorithm with dynamic adjusting parameters*: Utility for MiniMax Algorithm
- *Genetic Algorithm*: Adjust parameters

Those algorithms are referred to the textbook[2] we learned.

1.3 Software and Hardware

1.3.1 Software

- Reversed Revesi AI code: *Python* (Editor: *Pycharm Professional 2021.1.1*)
- Adjusts parameters: *Jupyter Lab* (Editor: *Anaconda*)
- Online usability and round robin test: *SUSTech Reversed Reversi AI platform*
- Report writing: LaTeX (Editor: *Overleaf*)

1.3.2 Hardware

- Basic code and report writing: *Lenovo-Rescuer Inteli7-9760H CPU @ 2.60GHz(12CPUs), 2.6GHz*
- Adjusts parameters: *Server Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz(32CPUs)*

1.4 Application

The project can help us in the following applications:

- Adversarial AI competition: Gobang, Halma, Reversi, I-go and more advanced competition.
- Genetic algorithm can be applied to solve NP-hard problems: Route planning, Resource Allocation, etc.
- Limited Time Fast Effect AI competition: Competition held by TenCent, Alibaba, HuaWei, etc.
- Also you can build application for human-computer fighting of Reversed Reversi.

2 Methodology

2.1 Notation

Symbol	Description
w	weight board value
s	stability value
m	mobility value
k	kill value
cr	color ratio value
c	corner value
c_1	corner value evaluation part one
c_2	corner value evaluation part two
N	numbers of stages need to explore
b^*	effective branching factor
d	depth of the solution tree in the MiniMax algorithm
$T_{Minimax}$	time complexity caused by the MiniMax algorithm
T_u	time complexity caused by the calculation of utility
O_i $i \in \{w, s, m, k, c, cr\}$	time complexity caused by different heuristic strategy

Table 1: Notation List

2.2 Data Structure

There are three main data structures used in this project.

2.2.1 Move

Structure *Move*(Figure.1) is used for recording one of the next position which the MiniMax uses, it contains two parts:

Attributes:

- *pos*: a tuple (i,j), recording one selection of the candidate list
- *utility*: it is calculated according to current utility and the deeper utility encountered by MiniMax Algorithm

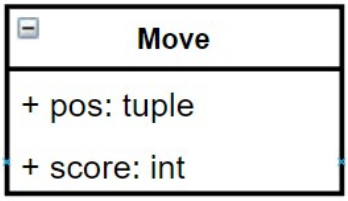

Figure 1: Move data structure

2.2.2 AI

Structure *AI*(Figure.2) is the interface that the project provided, it has several attributes and methods that need to implement.

Attributes:

- *chessboard*: a two dimensional 8x8 list that record the current chessboard
- *color*: a int number, in which -1 indicates COLOR_BLACK, 0 indicates COLOR_NONE and 1 indicates COLOR_WHITE

- *time_out*: a float number, recording the start time of one MiniMax Search
- *candidate_list*: a list of tuples, recording the valid positions according to current chessboard and current AI color
- *step*: a int number, recording the number of moves the board has made so far

Method:

- *go()*: AI pick valid positions according to the valid positions of the current chessboard and current AI color, and SUSTech Reversed Reversi AI platform will pick the last element of the candidate list as the next movement of this AI

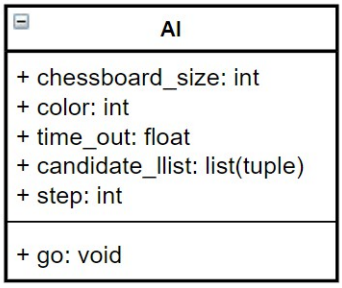

Figure 2: AI data structure

2.2.3 Game

Structure *Game*(Figure.3) is the offline AI contest class. It can add two AI class and let them play against each other, display each step and judge the winner.

Attributes:

- *user_black*: the AI class whose color is COLOR_BLACK
- *user_white*: the AI class whose color is COLOR_WHITE
- *chessboard*: a two dimensional 8x8 list that represent as the game chessboard
- *turn*: a int number, recording the current turn of two AI, in which -1 indicates COLOR_BLACK, 0 indicates COLOR_NONE and 1 indicates COLOR_WHITE

Methods:

- *play_chess()*: let one AI pick the next position it wants to go, print and refresh the chessboard
- *play_game()*: initial the game and loop until the game is completed
- *check_terminal()*: judge whether the game is over, and who is the winner

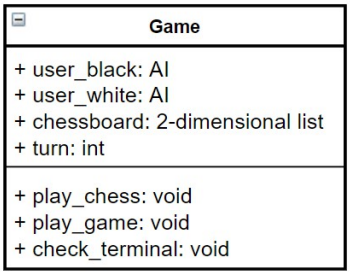

Figure 3: Game data structure

2.3 Model Design

2.3.1 Problem Formulation

This problem can be formulated into a search problem[3].

- *States*: The two-dimensional 8x8 board, and all the arrangement of pieces on the board
- *Initial state*: If AI's color is COLOR_BLACK, the initial state is the initial 8x8 board with 4 pieces placed on the middle, else the initial state is the state which opponent make the first move.
- *Actions*: Add some possible candidate positions to the candidate_list
- *Transition*: SUSTech Reversed Reversi AI platform will use the last element in candidate_list to create the next state each time after go() algorithm is finished.
- *Goal test*: Determine who has minimum pieces on the board when the game is finished
- *Path cost*: Utility that each time place a piece onto the board. (Implemented by Heuristic Algorithm)

2.3.2 Basic Flowchart

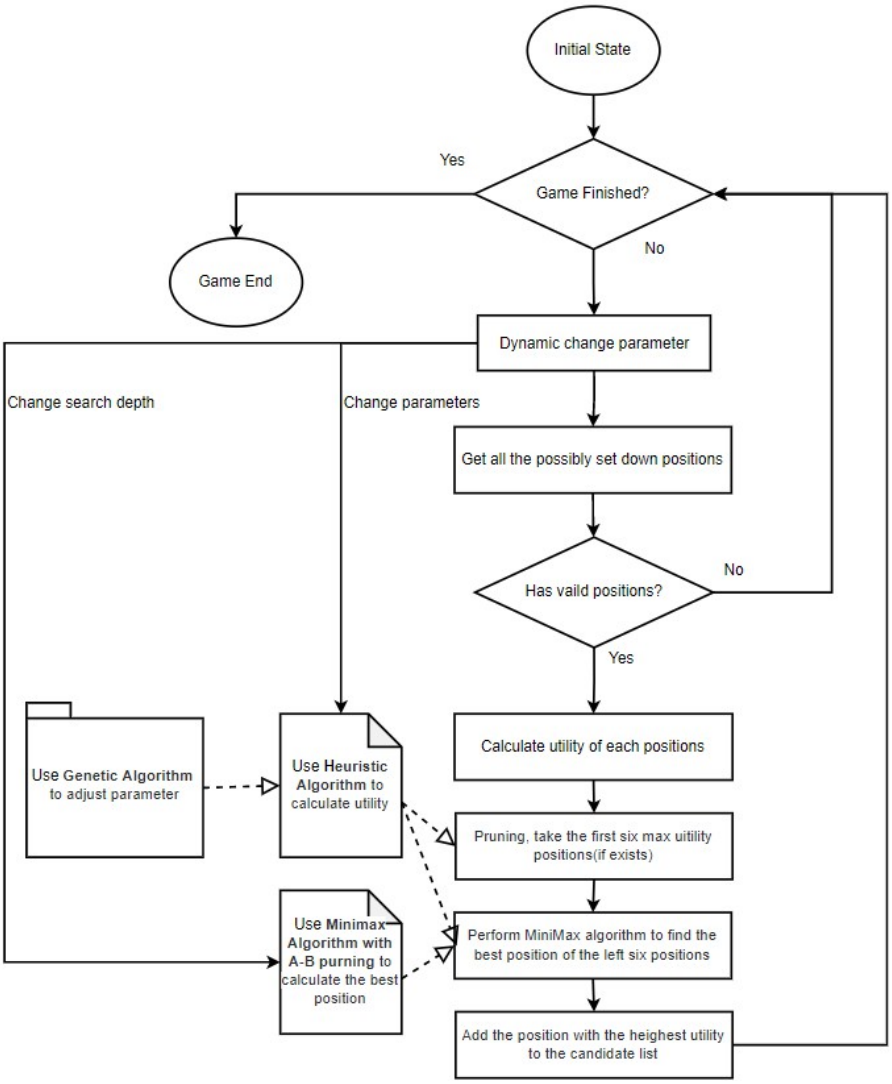

Figure 4: Reversi AI basic algorithm flowchart

2.4 Details of Algorithms

2.4.1 Get valid candidate list

The basic require of this project is to pass the usability cases which means AI must find valid positions to set down the piece. Basic methods are listed here:

Algorithm 1 GetPossiblePositions

Input: chessboard
Output: possibleSet

```
curPieces ← find all the pieces whose color equals to AI color;
create possibleSet;
for pos in curPieces do
    for dir in DIRECTIONS do
        directionSet ← EdgeSearch(chessboard, pos, dir);
        possibleSet ∪ directionSet;
    end for
end for
return possibleSet;
```

Algorithm 2 EdgeSearch

Input: chessboard, pos, dir
Output: directionSet

```
create directionSet;
while True do
    pos.X += dir.X;
    pos.Y += dir.Y;
    if pos.X or pos.Y cross the border then
        break;
    end if
    if not trigger and color in chessboard(pos) equals to COLOR_NONE then
        break;
    end if
    if trigger and color in chessboard(pos) equals to COLOR_NONE then
        directionSet ∪ pos;
    end if
    if color in chessboard(pos) equals to AI's color then
        trigger ← False;
    else
        trigger ← True;
    end if
end while
return directionSet;
```

2.4.2 MiniMax Algorithm with A-B pruning

Minimax algorithm is the main logic algorithm widely used in the Adversarial Search. This project implement the MiniMax Algorithm, and add more pruning to it in order to get "best candidate" in an reasonable time.

Algorithm 3 MaxiMize

Input: chessboard, depth, α, β, n
Output: bestMv

```
initialize bestMv from class Move();
if depth ≤ 0 then
    return bestMv
end if
possibleSet = GetPossiblePositions(chessboard);
purnCandidate = Select the first n utility(candidate);
for cand in purnCandidate do
    tempChessboard ← SetDownChess(cand);
    mv ← cand.getMove()
    tempMv ← MiniMize(tempChessboard, depth-1, α, β, n);
    mv.utility ← tempMv.utility;
    if mv.utility > bestMv.utility then
        bestMv ← mv;
    end if
    if mv.utility ≥ β then
        break;
    end if
    if mv.utility > α then
        α ← bestMv.utility
    end if
end for
return bestMv
```

Algorithm 4 Minimize

Input: chessboard, depth, α, β, n
Output: bestMv

```
initialize bestMv from class Move();
if depth ≤ 0 then
    return bestMv
end if
possibleSet = GetPossiblePositions(chessboard);
purnCandidate = Select the first n utility(candidate);
for cand in purnCandidate do
    tempChessboard ← SetDownChess(cand);
    mv ← cand.getMove();
    tempMv ← MaxiMize(tempChessboard, depth-1, α, β, n);
    mv.utility ← tempMv.utility;
    if mv.utility < bestMv.utility then
        bestMv ← mv;
    end if
    if mv.utility ≤ α then
        break;
    end if
    if mv.utility < β then
        β ← bestMv.utility;
    end if
end for
return bestMv;
```

2.4.3 Heuristic Algorithm

Heuristic algorithm is used to calculate the specific utility of one state. In this project, it not only contains some heuristic criteria, but also uses some dynamic parameters change strategy.

Algorithm 5 GetUtility

Input: chessboard, step
Output: utility

```
1: hyperPparameter ← list();
2: change hyperPparameter according to different period of step;
3: w ← GetWeightBoardValue(chessboard);
4: s ← GetStabilityValue(chessboard);
5: m ← GetMobilityValue(chessboard);
6: k ← GetKillValue(chessboard);
7: c ← GetCornerValue(chessboard);
8: cr ← GerColorValue(chessboard);
9: value ← list of(w, s, m, k, c, cr);
10: return value * hyperPparameter;
```

The following sections detail specific heuristic aspects inspired from Reversed Reversi game[4] and the Reversed Reversi strategy website[5]:

weight board value

Figure 5: Weight Board Value Criteria

A 8x8 two-dimensional board, each element represent one utility of a specific grid, If AI place it's piece in this grid, it will obtain the score of this grid. Thus we calculated all the pieces in the current chessboard and get the total score considering both AI and it's opponent.

stability value

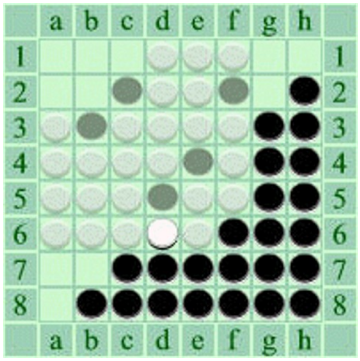

Figure 6: Stability Value Criteria

Stability is the number of pieces which you opponent can't not reverse. In Reversed Reversi, a player who has few stability is more likely to win.

mobility value

Figure 7: Mobility Value Criteria

Mobility is the number of piece which you can place, it can be easily calculate through get all the possible set down. This is a fluctuate value that could change during different period of game.

kill value

Figure 8: Kill Value Criteria

Kill is the number of pieces which you can reverse when you place the dice in the specific place. In Reversed Reversi, we prefer getting smaller kill value than the opponent.

color value

Color value is the numbers of oppnent's pieces minus AI's pieces, which is also the final evaluative criteria of the whole game. Whereas we will dynamiclly change the hyperparameter since during the early period, to capture some good place is more important than have a higher color value.

corner value

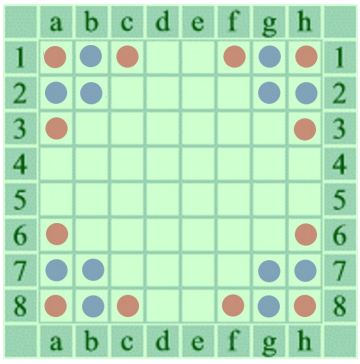

Figure 9: Corner Value Criteria

Corner value is part of the main heuristic strategy of the AI design. During the attempt in the AI competition, we conclude that some grid are more aggressive and have more possibility to let your opponent place more pieces. As Figure.9 shows, red grids are those we strongly reluctant to place our piece in. And blue grids are those we strongly recommend to conquer it as fast as possible.

3 Empirical Verification

3.1 Usability Test

3.1.1 Basic Test Cases

SUSTech Reversed Reversi AI platform provide ten basic test cases, which can prove the validity of method *GetPossiblePositions()*.

3.1.2 Self Battle

Class *Game()* is designed for the following reasons:

- Test the effectiveness of my heuristic algorithm.
- Show the progressiveness of each version of my AI design.
- Display each step when a AI choose a position to place its pieces.
- Check the invalid move.
- Let two AI class battle against each other and judge who is the winner.

This class is very important to tell me where is the bug and how much performance my code make.

3.2 Time Performance Measure

3.2.1 Time Complexity Analysis

The time complexity in performing MiniMax Algorithm is defined as follows:

- N: Numbers of stages need to explore
- b^*: Effective branching factor
- d: Depth of the solution tree

Since AI only search for up most depth 5 due to the time limit, in practice, A-B pruning performs inferiorly.

$$T_{MiniMax} = O(b^d)$$

In practice, I first pruned down the Minimax search space, taking only the top six utility positions from the candidate list of the first layer. Thus the average branching factor is constraint to 6, and the depth is up to 5. So there is $N = 6^5 = 7776$ stages that I need to explore in average. For each stages, AI needs to calculate it's utility. Suppose we need to check n grids. Then the time complexity of the utility is:

$$T_u = O_w(n) + O_s(n) + O_{mk}(n^2) + O_c(1) + O_{cr}(n) = O(n^2)$$

Thus the total time complexity during the search of function *go()* of AI is $O(b^d n^2)$.

3.2.2 Time Optimization

The *timeit()* method in Python is used to get the execution time of the code. I used it to test the *go()* method.

Here I list some of the time optimization strategies when I was revising my code:

- You can use two ways to search for the valid position. First one is to search begin from your pieces. Second is to search begin from a blank grid. Since in the later period, you have so many pieces on the board that search from the blank is more effective than the first way. And you can dynamically change your search strategy.
- *Numpy* is a high-performance Python data science package. It's built-in functions are more effective than basically using the basic statements like *for loop* in Python. Try to revise the code into using more numpy function when calculate the utility.
- *Numba* translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN. Use it when you have no choice but to use some *for loop*. But remember to split your *for loop* from the original function into a new part of function, since numba can not handle some complex cases.

3.2.3 Stop Calculation Trigger

In order to prevent from time-out, I also use a trigger to calculate the time once AI starts *go()*. If the calculate time is close to 4.9s, MiniMax algorithm will terminal and return the current best position. This is the ultimate means of the whole algorithm, and I try my best not to trigger it.

3.3 Intelligence Performance

3.3.1 Self Competition

Class *Game()* can register two AI class, let them have a competition and judge the winner. Each time when I improve my code, such as improving the dynamic adjusting hyparameters, the time cost, the search depth, I will record different versions. I have been created 9 versions before this project finished. And each time I tested whether the latter version can beat the previous version.

3.3.2 SUSTech Reversed Reversi AI platform

SUSTech Reversed Reversi AI platform is the main effective platform to measure the intelligence of my AI design. It can provide the following message:

- *Base Cases*: Verify that the code works (the AI correctly follows the rules of Reversed Reversi).
- *Friends Battle*: Search the student's ID you want to play and make a competition with him/her can also offer you a chance to verify your AI's performance. In addition, watching other AI's strategies provide me some new heuristic ways to improve my algorithm.
- *Rank*: Rank shows how you AI performed roughly. Offer you general status among the participants in this project.
- *Round Robin*: It is the most precise way to show that during this project how much students you beats and how much game your AI wins.

3.4 Adjust Hyperparameters

3.4.1 Hyperparameters Phases

During the project, I have tried several combinations of my hyperparameter. Basically, I have the 7 aspects mentioned before.(I break corner value into 2 parts, indicating my strategy.)

Stage1: Static Combinations

w	s	m	k	c1	c2	cr
1	30	10	10	200	500	10

Table 2: Static Hyperparamter Strategy

-500	200	-8	6	6	-8	200	-500
200	30	-16	3	3	-16	20	200
-8	-16	4	4	4	4	-16	-8
6	3	4	0	0	4	3	6
6	3	4	0	0	4	3	6
-8	-16	4	4	4	4	-16	-8
200	30	-16	3	3	-16	20	200
-500	200	-8	6	6	-8	200	-500

Table 3: Weight Board Value

Stage2: Dynamic Combinations

After I trying to use different hyper parameters in different phases of game. I revise my hyperparamters as below. And I didn't change the weight board value as the Table.3 shows.

phase	w	s	m	k	c1	c2	cr
1	1	30	10	10	200	500	10
2	1	30	30	20	100	300	50
3	1	100	30	100	70	200	300
4	1	300	30	100	50	150	500
5	1	500	30	100	50	150	999

Table 4: Dynamic Hyperparamter Strategy

The reason why I use this strategy to change hyperparameters will be introduced in Section 3.4.2.

Stage3: Genetic Algorithm in adjusting hyperparameters

The strategy in using Genetic Algorithm can be found in Section 3.4.3. In order to make GA algorithm converge quickly. I only split the whole game into three phases with 7x3 parameters. And the results are listed below:

phase	w	s	m	k	c1	c2	cr
1	1	36	0	6	36	44	50
2	1	40	0	50	2	11	33
3	1	39	25	8	1	50	39

Table 5: Genetic Algorithm Strategy

3.4.2 Dynamic Adjust Hyperparameters Strategy

In Section 2.4.3 we have talked about some heuristic aspects used and the dynamic change parameters over the period. In practice, I divided it into 7 periods. The main strategy is:

- In the early periods, I focus more on corner value. Since it is advantageous for AI to reach those position. And in the later of the game, the opponent has no choice but to places pieces in the vertex positions, which is very bad for opponent, because it will get a lot of stabilizers.
- In the middle periods, I gradually increase the importance of stability and color value, and gradually decrease the importance of corner value, since I want the opponent to get more stability position and maintain the number of AI's pieces in a fewer situation.
- In the final periods, I adjust the color value to be the most important parameter to be considered. The game's final target is to have the fewest pieces left in the board.

Also, when designing the weight board value, I also encourage the AI to reach favourable positions, and prevent AI entering the worse positions. Weight board value is something adjusting the action of AI globally, which means I will not adjust it dynamically.

3.4.3 Genetic Algorithm

Genetic Algorithm (GA) is used to adjust hyperparameters in a more rational ways. In this project, I used a integrate Python package called Geatpy[6] to perform GA. It is a well encapsulated package that can run GA and many of its varietas.

In Geatpy, there are only two things you need to do:

- Define the parameter you want to get and their domains.
- Define the fitness function, which is called *aimFunc* in actual.

The aimFunction is defined as followed:

```
function aimFunc(population):
    return fitness

fitness = new list(population.length)
for individual in population:
    parameter = get individual parameter
    create my_AI through class AI(parameter)
    for opponent in AI_pool:
        my_AI set color COLOR_BLACK
        opponent set color COLOR_WHITE
        create new_game through class Game(my_AI, opponent)
        fitness[my_AI] = new_game.winner is my_AI?
            fitness[my_AI]+=1:fitness[my_AI]-=1

        switch color of my_AI and opponent
        create new_game through class Game(opponent, my_AI)
        fitness[my_AI] = new_game.winner is my_AI?
            fitness[my_AI]+=1:fitness[my_AI]-=1
return fitness
```

The AI pool is supposed to be other student's AI in SUSTech Reversed Reversi AI platform. However, due to the limit of the platform and the poor network connection, I couldn't finished it on time. Instead, I define the fitness function as all the population play round robin themselves. However, it has poor convergence. This is the most regrettable part of this project, because the results of genetic algorithms are not as good as dynamic adjust hyperparameters.

3.5 Experimental Results

3.5.1 Basic Test Cases Verify

This project firstly pass all the cases, which proves the validity of method *GetPossiblePositions()*

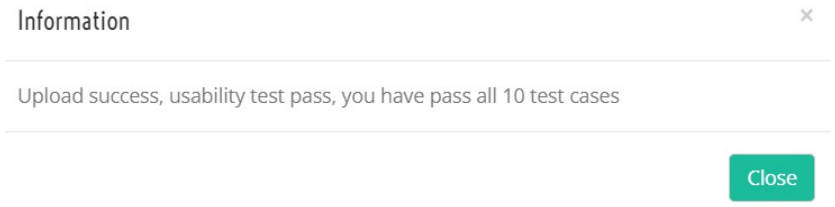

Figure 10: Use Case Passed Testify

3.5.2 Heuristic Algorithm Verify

The following shortcuts shows that heuristic algorithm really works.

Figure.11 shows that AI will conquer the advantageous position once it get chance in the early period.

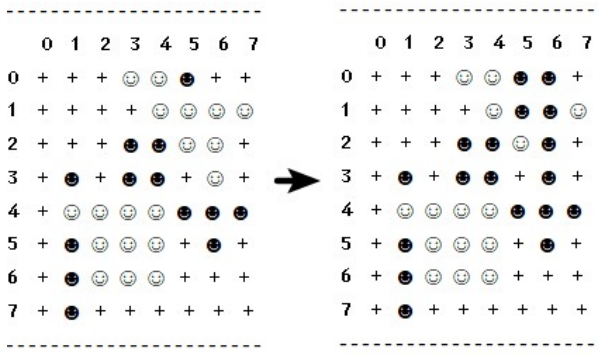

Figure 11: Conquer the Advantageous Position Quickly

Figure.12 shows that AI strongly avoids place piece in the disadvantage position until it has no other choice.

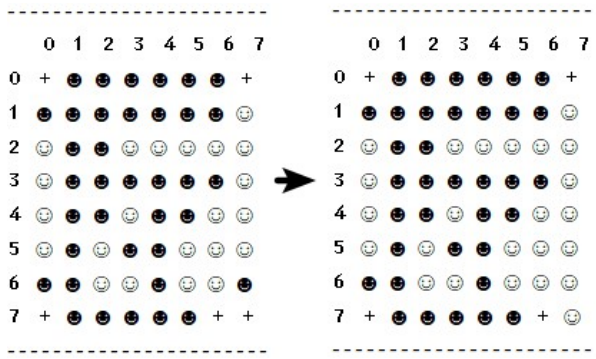

Figure 12: Avoid Place Piece in Disadvantage Positions

Figure.13 shows that AI will quickly select very attacking positions (Notice the three positions in the lower left) to let the opponent have no choice but to get the most disadvantageous position. We call this method —— "conquer 3 corners".

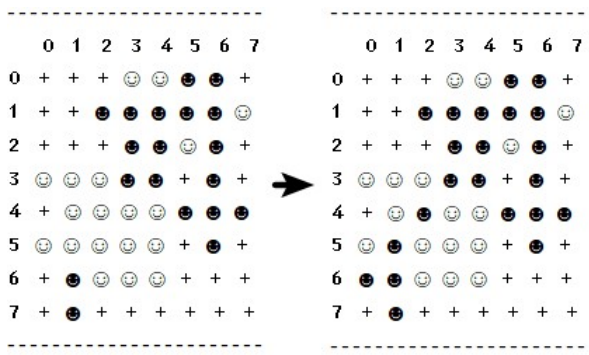

Figure 13: Conquer 3 Corners

3.5.3 Time Optimization Verify

Here is three comparison in using original Python, numba and numpy when I calculate the weight board value. It shows that the operational efficiency among those three are numpy >> numba >> origin:

```python
def get_weight_board_value(chessboard, COLOR):
    weight_board = [
        [-500, 200, -8, 6, 6, -8, 200, -500],
        [200, 30, -16, 3, 3, -16, 30, 200],
        [-8, -16, 4, 4, 4, 4, -16, -8],
        [6, 3, 4, 0, 0, 4, 3, 6],
        [6, 3, 4, 0, 0, 4, 3, 6],
        [-8, -16, 4, 4, 4, 4, -16, -8],
        [200, 30, -16, 3, 3, -16, 30, 200],
        [-500, 200, -8, 6, 6, -8, 200, -500]
    ]
    COLOR_OPPONENT = -COLOR
    player_score = 0
    for i in range(8):
        for j in range(8):
            if chessboard[i][j] == COLOR:
                player_score += weight_board[i][j]
            elif chessboard[i][j] == COLOR_OPPONENT:
                player_score -= weight_board[i][j]
    return player_score
```

```python
@jit(nopython=True)
def get_weight_board_value1(chessboard, COLOR):
    weight_board = [
        [-500, 200, -8, 6, 6, -8, 200, -500],
        [200, 30, -16, 3, 3, -16, 30, 200],
        [-8, -16, 4, 4, 4, 4, -16, -8],
        [6, 3, 4, 0, 0, 4, 3, 6],
        [6, 3, 4, 0, 0, 4, 3, 6],
        [-8, -16, 4, 4, 4, 4, -16, -8],
        [200, 30, -16, 3, 3, -16, 30, 200],
        [-500, 200, -8, 6, 6, -8, 200, -500]
    ]
    COLOR_OPPONENT = -COLOR
    player_score = 0
    for i in range(8):
        for j in range(8):
            if chessboard[i][j] == COLOR:
                player_score += weight_board[i][j]
            elif chessboard[i][j] == COLOR_OPPONENT:
                player_score -= weight_board[i][j]
    return player_score
```

```python
def get_weight_board_value2(chessboard, COLOR):
    weight_board = np.array([
        [-500, 200, -8, 6, 6, -8, 200, -500],
        [200, 30, -16, 3, 3, -16, 30, 200],
        [-8, -16, 4, 4, 4, 4, -16, -8],
        [6, 3, 4, 0, 0, 4, 3, 6],
        [6, 3, 4, 0, 0, 4, 3, 6],
        [-8, -16, 4, 4, 4, 4, -16, -8],
        [200, 30, -16, 3, 3, -16, 30, 200],
        [-500, 200, -8, 6, 6, -8, 200, -500]
    ])
    player_score = sum(np.multiply(weight_board, chessboard).flatten())
    if COLOR == -1:
        player_score = -player_score

    return player_score
```

Figure 14: Code difference among origin, numba and numpy

```
%timeit -n 5000 get_weight_board_value(chessboard, 1)
45.2 µs ± 1.47 µs per loop (mean ± std. dev. of 7 runs, 5000 loops each)

%timeit -n 5000 get_weight_board_value1(chessboard, 1)
The slowest run took 100.84 times longer than the fastest. This could mean that an intermediate result is being cached.
23.9 µs ± 54.7 µs per loop (mean ± std. dev. of 7 runs, 5000 loops each)

%timeit -n 5000 get_weight_board_value2(chessboard, 1)
15.3 µs ± 1.42 µs per loop (mean ± std. dev. of 7 runs, 5000 loops each)
```

Figure 15: Time cost between origin, numba and numpy

After the optimization of my code, I found the the time cost is effectively improved.

Version	Loops	Average time costs
AI1_6	100	202ms±4.75ms per loop
AI1_9	100	26.4ms±1.45ms per loop

Table 6: Time cost in different version of AI

3.5.4 Self Competition Verify

The following lists contains the versions I modified during the project:

- *AI1_1*: Pass all the base cases.
- *AI1_2*: Add MiniMax Algorithm with A-B pruning. Add heuristic algorithm to calculate utility.
- *AI1_3*: Modify aspects of heuristic algorithm.
- *AI1_4*: Add stop calculation trigger to control the time.
- *AI1_5*: Make pruning in MiniMax Algorithm. Only used top six utility of the valid positions in the following search.
- *AI1_6*: Fix some bugs in MiniMax Algorithm. Add more stronger heuristic algorithm "conquer 3 corners" to make the AI play games in a more strategic way.

- *AI1_7*: Add dynamic change hyperparameter strategy, break the game into seven periods. Add two ways to search for valid positions.
- *AI1_8*: Revise the AI code suitable for adjusting hyperparameters using genetic algorithms (But it doesn't work well).
- *AI1_9*: Reduce time cost by using numpy and numba to revice the code. Increase the search depth.

The most important versions of my AI is AI1_3, AI1_6 and AI1_9. I let them to make a battle. And the result seems that it really works. For example, AI1_9 can defeat AI1_7 and AI1_6, just as AI1_6 can defeat AI1_5.

3.5.5 SUSTech Reversed Reversi AI platform Verify

- *Usability Test*: As mentioned above, I have passed all the test cases.
- *Winning Percentage*: The winning percentage on the platform is probably 50%. And before the platform closed, I maintain the rank around 30-40.
- *Round Robin*: I finally get rank 62/160 in the round robin.

3.6 Conclusion

3.6.1 Advantages and Disadvantages of the Algorithms

In the whole projects, I have tried several algorithms to improve the intelligence of my AI – MiniMax algorithm, Heuristic algorithm and Genetic algorithm and have successfully implemented them. And I use numpy and numba to optimize the time cost, made it successfully from 200ms per call down to 20ms per call. However, there still exist some time cost problems. If I search in depth 5 in the whole game, then it will timeout during the middle period of the game since there contain more than 10 valid positions to be searched in average. Also the fitness function I defined in the Genetic algorithm is not behave well. Its convergency is bad.

3.6.2 Experience

This project really taught me a lot in understanding AI behavior and adversarial search. It made me implement algorithm in real scene, and provided a platform to battle with my classmate. The report I made also gave me another chance to reorganize my idea. All in all, I am appreciated for this experience.

3.6.3 Deficiencies and Possible Improvement Directions

Given me more time and more opportunity to have a competition in the platform, I will improve my Genetic Algorithm by letting my AI play against the whole class and get better hyperparameters. I will communicate with my classmates and learn their heuristic thinking and tuning methods to improve my own AI. And I will do more research in whether there exist any other algorithms in adversarial search.

References

[1] Wikipedia contributors, "Reversi — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Reversi&oldid=1048261233, 2021, [Online; accessed 31-October-2021].

[2] J. R. Stuart and N. Peter, *Artificial Intelligence A Mordern Approach*. Pearson Education ©, 2010.

[3] Wikipedia contributors, "Search problem — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Search_problem&oldid=1051862992, 2021, [Online; accessed 31-October-2021].

[4] B. Rose, *Othello A Minute to Learn, A Lifetime to Master*. United States Othello Association ©, 2005.

[5] M. Steve, "Strategy guide for reversi & reversed reversi," 2005. [Online]. Available: http://samsoft.org.uk/reversi/default.asp

[6] e. Jazzbin, "geatpy: The genetic and evolutionary algorithm toolbox with high performance in python," 2020.