Report for Project Reversed Reversi

Huihui Huang
Department of Computer Science and Engineering
Southern University of Science and Technology
12010336
12010336@mail.sustech.edu.cn

I. INTRODUCTION

A. Problem Description

Reversed Reversi is a two-player chess game. The two players take turns placing white or black discs on the board. During a play, any discs of the opponent's color that are in a straight line and bounded by the disc just placed, then bounded discs of the opponent player's color are turned over to the current player's color. The game will continue until there is no empty spot on the board, and the winner would be the one whose number of discs existing on the board is smaller.

B. Problem Applications

The Reversed Reversi game is a zero-sum game. A zero-sum game is a mathematical representation in game theory and economic theory of a situation that involves two sides, where the result is an advantage for one side and an equivalent loss for the other. In other words, player one's gain is equivalent to player two's loss, therefore the net improvement in benefit of the game is zero. A zero-sum game is very common in our life, such as poker, chess, or reversi in games, economic benefits of low-cost airlines in saturated markets, etc., which are widely used.

C. Purpose

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. Besides, implementing the AI algorithm helps 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. PRELIMINARIES

A. Terminology

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

- Chessboard: a numpy array about chessboard information.
- Candidate List: a list of the legal moves for specific player and chessboard.
- Mobile disc: The mobile disc is a disc that can be placed in a valid position.
- Frontier disc: The frontier disc is adjacent to at least one empty square.
- Stable disc: The stable disc is that, no matter how the other disc move in the future, will not be reversed by others.
- Mobility The number of mobile discs.
- Frontier The number of frontier discs.
- Stability: The number of stable discs.
- Board Reward: Sum the discs value according to the weight of the chessboard.
- Discs Number: discs number is the number of discs of the corresponding color that are currently on the board.
- Game Tree: In the mini-max algorithm, because of the DFS algorithm a tree structure is formed, which is called the game tree.

B. Notation

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

- $S_{mobility}$: The score of mobility of the board.
- $S_{frontier}$: The score of frontier of the board.
- $S_{stability}$: The score of stability of the board.
- S_{board}: The score of board reward of the board.
- $S_{discsNum}$: The score of discs number of the board.
- 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_s^1: The weight of stability difference in the stage one.
- W_s^2: The weight of stability difference in the stage two.
- W_m^1: The weight of mobility difference in the stage one.
- W_m^2: The weight of mobility difference in the stage two.
- W_b: The weight of board reward difference.
- D_m: The difference between the number of self mobile discs and opponent.
- D_f: The difference between the number of self frontier discs and opponent.
- D_s: The difference between the number of self stable discs and opponent.
- $V_{self}(x, y)$: The sum of the self discs value according to the weight of the chessboard.
- $V_{opponent}(x, y)$: The sum of the opponent discs value according to the weight of the chessboard.

III. METHODOLOGY

A. Workflow

The proposed method of AI algorithm which determinate the behavior of the agent is divided into three steps. The first step is to get the current chessboard and identify the valid position to place the disc. The second step is to evaluate each

Algorithm 1 $miniMaxDecision$

Input: chessboard, color
Output: action

$bestScore \leftarrow -infinity$
$beta \leftarrow infinity$
$action \leftarrow empty\ tuple$
for each $act \in actionList$ **do**
    move and changes the board
    current depth increase by 1
    $value \leftarrow min(value, maxValue$
        $(chessboard, alpha, beta, color))$
    current depth decrease by 1
    if $value > bestScore$ **then**
        $bestScore \leftarrow value$
        $action \leftarrow act$
return $action$

valid position by the evaluate function. The third step is to order the position by their evaluated score from lowest to highest in the candidate list for the program will choose the last one as the position we want to place the discs.

Besides, the genetic algorithm is divided into four steps. The first is to create the population by some random value. The second step is to let those agents play with each other and each agent will get their evaluation score. The third step is to sort them from the highest to the lowest and mutate the parameter of the agent with a high score. The final step is to update the mutated agents as the new population and then repeat the second step until the times of reputation reach the number we set.

B. Detailed algorithm

1) AI Algorithm: As shown in algorithm 1 , 2 and 3. Based on the rules of Reversed Reversi Game, when the game begins 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()*, I use the mini-max algorithm combined with alpha-beta pruning to search the game tree. And with the test of time, I find that when the depth of the tree is four it can have the best trade-off between time and winning percentage.

When it comes to an evaluation in the mini-max algorithm, the function *evaluationFunciton()* in class AI is to estimate the value of the current player with the chessboard.

It consists of 5 parts and some parts are separated by the number of steps. When the number of steps is smaller than 30 it is stage 1 and when the number is larger than 30 it is stage 2. The following are detailed about the 5 parts:

- Mobility Score:

$$S_{mobility} = \begin{cases} W_m^1 \cdot D_m & if\ stage = 1 \\ W_m^2 \cdot D_m & if\ stage = 2 \end{cases}$$

Algorithm 2 $minValue$

Input: chessboard ,alpha, beta, color
Output: value

if $current\ depth > max\ depth$ **then**
    return $evaluationFunction(chessboard)$
if $actionList\ is\ empty$ **then**
    if $opponent\ actionList\ is\ empty$ **then**
        return $evaluationFunction(chessboard)$
    return $maxValue(chessboard, alpha, beta, -color)$
$value \leftarrow infinity$
for each $act \in actionList$ **do**
    move and changes the board
    current depth increase by 1
    $value \leftarrow min(value, maxValue$
        $(chessboard, alpha, beta, -color))$
    current depth decrease by 1
    if $value \leq beta$ **then**
        return $value$
    $alpha \leftarrow min(alpha, value)$
return $value$

Algorithm 3 $maxValue$

Input: chessboard ,alpha, beta, color
Output: value

if $current\ depth > max\ depth$ **then**
    return $evaluationFunction(chessboard)$
if $actionList\ is\ empty$ **then**
    if $opponent\ actionList\ is\ empty$ **then**
        return $evaluationFunction(chessboard)$
    return $minValue(chessboard, alpha, beta, -color)$
$value \leftarrow -infinity$
for each $act \in actionList$ **do**
    move and changes the board
    current depth increase by 1
    $value \leftarrow max(value, minValue$
        $(chessboard, alpha, beta, -color))$
    current depth decrease by 1
    if $value \geq beta$ **then**
        return $value$
    $alpha \leftarrow max(alpha, value)$
return $value$

- Frontier Score:

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

- Stability Score:

$$S_{stability} = \begin{cases} W_s^1 \cdot D_s & if\ stage = 1 \\ W_s^2 \cdot D_s & if\ stage = 2 \end{cases}$$

- Discs Number Score:

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

- Board Reward Score:

$$S_{board} = (\sum V_{self}(x,y) - \sum V_{opponent}(x,y)) \cdot W_b$$

2) Genetic Algorithm: A genetic Algorithm is used to search for an excellent evaluation function or evaluation parameters in a huge search space. I implement a genetic algorithm class, which packaged the complete genetic algorithm of each process function, including *generatingOriginPopulation()*, *compete()*, *fitness()*, *selection()*, *mutation()*, etc. The workflow of GA is shown in Fig. 1.

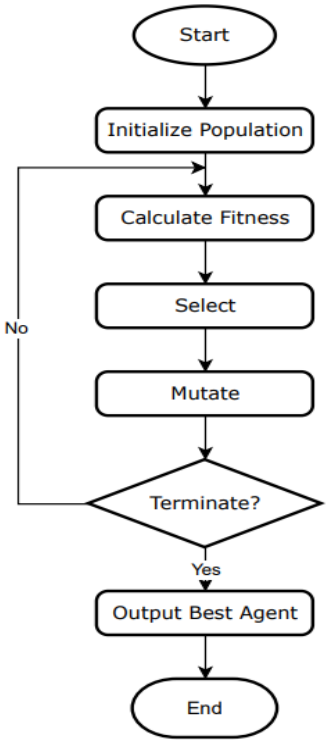

Fig. 1. Workflow of Genetic Algorithm

We can start the genetic algorithm by calling the *train()* function. In this function, we first call *generateOriginPopulation()* to generate an origin population randomly with a specific size such as 20. Then, we start a loop, which represents the generation times. For each loop, we call *fitness()* function, which will use the multi-process method to make each agent of the population fight against all other agents. After all the battles are completed, the number of wins and losses of each agent is counted, and the difference between the number of wins and the number of losses is taken as its fitness score. Also, it should print out the parameters of the agent with the highest fitness score.

To ensure that the algorithm time does not increase exponentially, I take the approach of keeping the number of populations constant. In the *selection()* function I select the first third of them as the best solutions and throw them to the new population. The remaining two-thirds consist of adding and subtracting random numbers to the parameters of the best solutions which can keep the quantity of the new population the same.

C. Analysis

1) Analysis on AI Algorithm: Assuming that *n* layers can be searched, the time-consuming of each layer search mainly comes from the evaluation function. The evaluation function needs to facilitate one pass through the board when calculating the value of the board and searching for stable discs, frontier discs, and mobile discs. Since it is a constant number of times, the search complexity of each layer is *O(1)*, so the search complexity of n layers is *O(n)*. The main parameter affecting the performance is the number of search layers. The larger the number of search layers, the better the effect will be, but the more time will be consumed, which requires a trade-off.

2) Analysis on Genetic Algorithm: The time consumed by the genetic algorithm mainly depends on the number of population in each generation and the number of iterations. Assuming that the population number of each generation is *s* and the number of iterations is *t*, then the time complexity of each generation is $O(s^2)$ and the total time complexity is $O(ts^2)$. As for the genetic algorithm, the two parameters mentioned above are also the determining factors. Theoretically, the larger the population size and the larger the number of iterations, the more likely it is to get the best solution, but the time consumption will also increase, which also requires a trade-off.

IV. Experiments

A. Setup

1) Software and Hardware: The project is developed using PyCharm as an 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 cloud server is CentOS8.2-Docker20-Ay0F with CPU: 2 cores Memory: 4GB.

2) Dateset: My chessboard values is a reference to this site [1], and the weight settings are obtained by the genetic algorithm. During the training of the genetic algorithm, I designed the following three modes to select the population because I did not know which selection mode could get the optimal solution better.

- Mode 1: Each time, the first one is selected as the best, which is directly put into the new population, and the remaining first half is mutated and put into the new population
- Mode 2: Each time, the first third is selected as the best and directly put into the population, and then it is added and subtracted by random number variation operation and also put into the new population.
- Mode 3: Each time, the first half is taken as the best, but instead of putting it directly into the new population, it is

added and subtracted by random number variation, and then put into the new population.

B. Results

Each mode is iterated 30 times with a population of 30. To test which mode is better, I let the population of each mode play as the black side against the other side. The number of games won is calculated and divided by the total number of games to get the winning rate. The result shows in TABLE I and concludes that the genetic method of mode 2 is better.

TABLE I
MODE WINNING RATE RESULT

Black \ White	Mode 1	Mode 2	Mode 3
Mode 1	-	34.35%	51.17%
Mode 2	76.89%	-	83.56%
Mode 3	43.71%	26.49%	-

In the following, I adopt the genetic method of mode 2 to obtain more results by increasing the number of iterations and the size of the populations. TABLE II shows how long it takes for me to iterate through mode 2 by setting different populations.

TABLE II
TIME COST DEPENDS ON POPULATION SIZE

Population Size	Time Cost(seconds)
10	12.47
20	40.12
30	91.66
50	282.64

Since I have a cloud server, I put the code on the cloud server and let the population iterate 200 times, and the individual weight values are as TABLE III.

TABLE III
RESULT OF EACH WEIGHT

W_m^1	W_m^2	W_f^1	W_f^2	W_s^1	W_s^2	W_p^1	W_p^2	W_b
2.51	1.88	5.75	3.03	0.75	4.38	1.56	5.21	2.22

C. Analysis

1) Effect of Hyper-parameters: Most of the parameters used in my algorithm are obtained by genetic algorithm, and the hyper-parameters are focusing on the population size and the iteration times. It can be seen from the experimental results that the larger the number of populations obtained each time, the more time is consumed. However, a large population can improve the probability of finding the optimal solution in a certain sense, so there is a trade-off to be found.

2) Results related to Methodology: My experiments are mainly aimed at genetic algorithms, and the main idea of genetic algorithms is based on the fact that as long as the number of iterations is enough, relatively optimal results can be obtained. Based on this theory, when I set the initial values, I directly use random numbers to initialize them. However, I find that if the way of crossover and mutation is wrong, the final result will be further and further away from the optimal solution. For example, in my mode 1, the parameter of the optimal solution increases with the number of iterations, which is unreasonable. Fortunately, I have also adopted other ways to select the optimal population and mutation, so that my genetic algorithm can get the optimal solution as much as possible.

V. CONCLUSION

A. The experience I learned

In this project, I learned to how to solve decision-making problems in zero-sum games by using Mini-max combined with the Alpha-beta pruning algorithm. Also, with continuous thinking and improvement of the algorithm, I found that the genetic algorithm could solve the problem of setting the evaluation function well. However, there are many ways to select the optimal population. To get the relatively optimal selection method, I set three modes for the selection method, to optimize my genetic algorithm.

B. Advantages of my algorithm

The advantage of my algorithm is that without the need for additional human participation I use the genetic algorithm to find the optimal parameters for my evaluation function, which largely improve the efficiency when optimize the algorithm.

C. Disadvantages of my algorithm

My algorithm had a very limited search depth of about 5-6 layers because of the limitation of the platform running time. Also, the implementation of my genetic algorithm is not very rigorous, for example, the initial value is set randomly, there is no crossover operation, and when mutating, random numbers are simply added or subtraction, which may cause me to lose the optimal solution.

D. Possible directions of improvements

In order to improve my search efficiency, I can consider turning the chessboard into binary or further optimize my algorithm. Also, 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 discs in a game is an important direction for improvement. Finally, the genetic algorithm can also be improved to some extent, such as adding crossover operation and carrying out more kinds of mutation when mutating.

REFERENCES

[1] S.Maguire, "Strategy Guide for Reversi & Reversed Reversi.", https://samsoft.org.uk/reversi/strategy.htm. (Accessed:October 30, 2022).

[2] Wikipedia contributors, "Minimax-Wikipedia, the free encyclopedia", https://en.wikipedia.org/w/index.phptitle=Minimax&oldid=1048398851 (Accessed:October 30, 2022).

[3] Wikipedia contributors, "Zero-sum game-Wikipedia, the free encyclopedia,", https://en.wikipedia.org/w/index.php?title=Zero-sum game& oldid=1050040981.(Accessed: October 30, 2022).