Report for Project 1: An Intelligent Agent Playing Reversi

Zhan Zhuang 11811721
Computer Science and Engineering
Southern University of Science and Technology
11811721@mail.sustech.edu.cn

1. Preliminaries

This report focuses on the algorithm and implement of an AI program that plays a Reversi (Othello) game, as well as the way to tune parameters for a variety of heuristic strategies with program results and performance. The program has been developed as the first project for the course of CS303 Artificial Intelligence, held in the Fall 2020 at Southern University of Science and Technology.

Reversi is a two-player deterministic zero-sum board game where the winner always has more discs at the end of the game. To be specific, the winner is the player who has more discs of his colour than his opponent at the end of the game. This will happen when neither of the two players has a legal move. Generally at this stage all 64 squares are occupied [1]. So the total states for Reversi are about 10^{28}, and there are a maximum of 60 moves between black and white. Further more, it is a perfect information game because the states are all fully observable.

This project used the 8x8 board (Othello board [1]) as a two-dimensional array, with each element as a value from the {-1,0,1}, corresponding to black, empty, and white respectively. Our goal is to train a agent to play the game of Reversi with a group of search methods. To achieve this goal, in my experiments, I tried the game many times on the web platform and in Python programs, with good results finally.

1.1. Software

This project is all written in Python 3.9 with PyCharm and the WZebra 4.2.4 [2] is helpful to test the performance of my program.

1.2. Algorithm

I used the combination of minimax tree search with alpha-beta pruning as the main algorithms in this project. The weight of evaluation function is also an important part in the work, so I used iteration trial to select and adjust the set of value.

1.3. Problem applications

With this program, we can try two modes which are human vs agent and agent vs agent to implement an AI game. Through repeated analysis and comparison, human players can learn the new ideas of Reversi based on this program. For artificial intelligence developer, the program can generate its training set and perform self-combat, algorithm optimization or neural network tuning.

2. Methodology

2.1. Notation

The primary notations used in this report are listed in **Table 1**.

TABLE 1. NOTATIONS

Symbol	Definition
$EVAL(p)$	the evaluation of a state of player p
$C-squares$	the squares at a2, a7, b1, b8, g1, g8, h2, and h7
$X-squares$	the squares at b2, b7, g2, and g7
$E(pos)$	the evaluation of the chessboard position pos
$D(p)$	the number of discs of player p
$M(p)$	the number of possible moves available to a player p
$S(p)$	the number of discs can't be flipped of player p
$F(p)$	the number of potential flips of player p
$self$	the player on our part
$opponent$	the player on the other part
sum_discs	the number of discs have been played on the board
lim_time	the time limit, the default for this project is 5 seconds
w cur_time	the time used in this turn
$score(i)$	the score of heuristic method i
w_i	the weight of heuristic method i

2.2. Architecture

2.2.1. Data structure. The primary data structure used in this report are listed in **Table 2**.

TABLE 2. DATA STRUCTURE

Function	Definition
chessboard	a two-dimensional array represents an 8x8 size checkerboard
candidates	a list of the most valuable positions which have found
actions(p)	a list of the available position the player p can set

2.2.2. Model design.

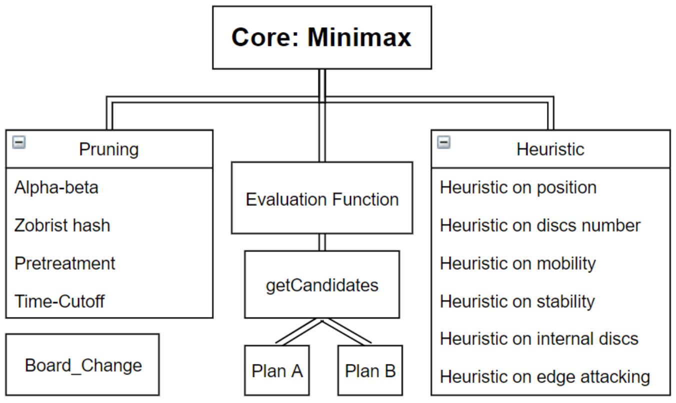


Figure 1. The Model Design

The most common search algorithm for this zero-sum game, in which each player takes turns executing his or her turn, is minimax. This algorithm needs to evaluate the state of the game, so it needs an evaluation function to judge the effect of black and white chess. For the evaluation function, as shown in the Figure 1, we adopt six heuristic strategies and adjust the weight of each strategy through iterative optimization. For the balance of search depth and time, we adopt four pruning algorithms to ensure the deep search of the state in the feasible region within the limited search time .

2.3. Detail of Algorithm

2.3.1. Minimax algorithm.

Minimax is an algorithm based on search that finds the minimum of the greatest likelihood of failure which is used in games and programs such as backgammon and go chess, in which two players take turns performing one step at a time. However, Minimax is a pessimistic algorithm that assumes that each step of the opponent will lead us in the direction of a pattern that has the least theoretical value from the current point of view, also the opponent has perfect decision-making ability. Therefore, our strategy should be to choose the best that the other side can achieve in our worst case to make the other side's perfect decision cause me the least loss. In this Reversi project, the pseudocode of this algorithm can be described as **Algorithm 1** .

2.3.2. Alpha-beta pruning.

] Alpha-beta pruning with two bounds that are passed along during the calculation that restrict the set of possible solutions based on the search tree. The algorithm corresponds to this branch-and-bound idea. [3] Specifically, the β is the minimum upper bound of possible solutions and the α is the maximum lower bound of possible solutions. So when any new solution value v is out of the range $\alpha \leq v \leq \beta$, its result needn't to be considered. In this Reversi project, the pseudocode of this algorithm can be developed from Minimax algorithm as **Algorithm 2**

Algorithm 1 Minimax(self, chessboard, player)

Input: *chessboard*: The current chessboard, {-1,0,1} for black, blank and white;
player: The current player: 1 for self, -1 for opponent;
Output: opt, v: A position with value that is the best choice;

```
1:  $actions \leftarrow getCandidates(chessboard, player)$
2:  $opt \leftarrow null$
3:  if $actions.length = 0$ then
4:      return $evaluation(chessboard, player), null$
5:  end if
6:  if $player = self.player$ then
7:      v = -Infinity
8:      for act in actions do
9:          newboard = move(chessboard, player, act)
10:         $v', null \leftarrow Minimax(newboard, -player)$
11:         $v = max\{v, v'\}$
12:         $opt = act$
13:     end for
14:     return v
15:     v = Infinity
16:     for act in actions do
17:         newboard = move(chessboard, player, act)
18:         $v', null \leftarrow Minimax(newboard, -player)$
19:         $v = min\{v, v'\}$
20:         $opt = act$
21:     end for
22:     return opt, v
23: end if
```

2.3.3. Pretreatment and Time-Cutoff pruning.

In order to search for more depth in limited time, we used the method of pruning according to time and the pretreatment of candidates.

For the depth of the initial Minimax search, according to the s (sum_discs) and $actions(p)$, it can be set in the way: .

$$depth = \begin{cases} s/4-6 & s > 46 \\ 3 & actions(p) > 14 \ \& \ s \leq 46 \\ 4 & 7 < actions(p) \leq 14 \ \& \ s \leq 46 \\ 5 & 5 < actions(p) \leq 7 \ \& \ s \leq 46 \\ 6 & 3 < actions(p) \leq 5 \ \& \ s \leq 46 \\ 7 & actions(p) \leq 3 \ \& \ s \leq 46 \end{cases} \tag{1}$$

Before each iteration of Minimax search, we directly calculated the evaluation functions for all possible solutions. Then we sort them and add a search range *scale* to do pruning .

For the *scale* of the candidates we will use, according to the time remaining t, which can be calculated by $(lim\_time - cur\_time)$ it can be set in the way: .

$$scale = \begin{cases} 1 & t > 4 \\ 3 & 2.5 < t \leq 4 \\ 4 & 2.1 < t \leq 2.5 \\ 5 & 0.4 < t \leq 2.1 \\ 7 & t \leq 0.4 \end{cases} \tag{2}$$

Algorithm 2 MinimaxWithAlphabetaPruning(self, chessboard, player, min, max, d)

```
Input: $chessboard$: The current chessboard, {-1,0,1} for black, blank and white;
    $player$: The current player: 1 for self, -1 for opponent;
    $min$: The lower bound has already been seen;
    $max$: The upper bound has already been seen;
    $d$: The depth of search tree;
Output: $opt, v$: A position with value that is the best choice;
1: $actions \leftarrow getCandidates(chessboard, player)$
2: if $actions.length = 0$ or $d = 0$ then
3:     return evaluation(chessboard, player)
4: end if
5: if $player = self.player$ then
6:     v = min
7:     for act in actions do
8:         newboard = move(chessboard, player, act)
9:         $null, v' \leftarrow Minimax(newboard, -player, v, max, d-1)$
10:        $v = max\{v, v'\}$
11:        $opt = act$
12:        if $v > max$ then
13:            return null, max
14:        end if
15:    end for
16:    return v
17:    v = max
18:    for act in actions do
19:        newboard = move(chessboard, player, act)
20:        $null, v' \leftarrow Minimax(newboard, -player, min, v, d-1)$
21:        $v = min\{v, v'\}$
22:        $opt = act$
23:        if $v < min$ then
24:            return null, min
25:        end if
26:    end for
27:    return opt, v
28: end if
```

2.3.4. Cutoff-test pruning.

To save search time, the search function is aborted if either the opponent or we are stuck with no moves or if all candidate actions are in the X-Squares. .

2.3.5. Zobrist hash pruning.

In this part, my methods and ideas come from these articles [4] [5] .

2.4. Evaluation function

In order to determine which step is the best solution, the evaluation function is used to analyze the score of each feasible solution, which is obtained by the above optimized search algorithm. The evaluation function EVAL is obtained by adding the following six heuristics according to different weights, so that, $EVAL = \sum_{i=1}^{6} w_i score(i)$.
.

2.4.1. Heuristic on position.

TABLE 3. POSITION VALUE TABLE

	A	B	C	D	E	F	G	h
1	3000	-80	2	5	5	25	-100	3000
2	-80	-1500	-5	-5	-5	-5	-1500	-80
3	25	-5	15	3	3	15	-5	25
4	10	-5	3	3	3	3	-5	10
5	10	-5	3	3	3	3	-5	10
6	25	-5	15	3	3	15	-5	25
7	-80	-1500	-5	-5	-5	-5	-1500	-100
8	3000	-120	27	5	5	20	-100	3000

The simplest heuristic function is use an position value table as shown in **Table 3**. In order to avoid placing the C-squares and X-squares and trying to preempt corners, we changed the value of these positions significantly.
So let Score(1) be the score according to the heuristic on position: $Score(1) = \sum_{pos} E(pos)chessboard[pos]$.

2.4.2. Heuristic on discs number.

This heuristic function is also brief, which is the difference between the discs number of two players. Considering that Reversi's individual discs are relatively unstable in the prophase and metaphase, I increment this weight only in the final phase. In fact, if you use this heuristic strategy alone, you have a lower chance of winning without enough stable discs. Let Score(2) be the score according to the heuristic on discs number: $Score(2) = D(self) - D(opponent)$.

2.4.3. Heuristic on mobility.

It's important to note that if a player cannot play legal moves (in other words his mobility is equal to zero), he is obliged to pass the turn, that is a serious disadvantage for him and a big advantage for the opponent, who probably will win the game [6]. In addition, greater mobility means more chances to search for key action. In this program, I calculate this strategy by calculating the candidate actions and let Score(3) be the score according to the heuristic on mobility: $Score(3) = M(self) - M(opponent)$.

2.4.4. Heuristic on stability.

The stable discs mean that the disc which will never be flipped by the opponent. Obviously, an corner is always a stable disc, and when a corner is occupied by our side, the discs along the edges and the internal discs in two edges of the corner will also become stable discs. The number of stable discs directly determines the final score, so it is a key evaluation factor. So let Score(4) be the score according to the heuristic on stability: $Score(4) = S(self) - S(opponent)$.

2.4.5. Heuristic on internal discs.

Because we always play the Reversi in a expand from the center of the board to the edges way, the disc that is flipped first is always the outermost disc, in other words, the outermost discs are more likely to be flipped. Therefore, keeping fewer external pieces and keeping our discs as close to the internal trend as possible helps to gain the turnover opportunities and reduce the mobility of opponent. So let Score(5) be the score according to the heuristic on stability: $Score(5) = -F(self) + F(opponent)$.

2.4.6. Heuristic on edge attacking.

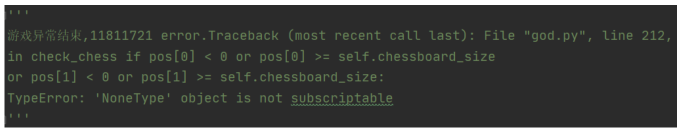
For operations involving edges and corners that are important, I use an array to record a finite number of cases and increase the reward or penalty values by ternary encoding them. When the algorithm calculates operations that take into account edges and corners, that part adds an item to the evaluation function to make the program intelligent. Take any edge like edge a, the ai from a1 to a8 and assign it to 3 to the i power. The following parameters are the sum of ternary code of the specific position obtained after training. If a move is completed, it produces a key corresponding to the key in the dictionary, then its score will increase the corresponding score.

```
find = {
     -6560: 20, -2186:  7, -6558:  7,
     -2184:  5, -726:   5, -2178:  5,
      3886: -8,  4354: -8, -4351: -8,
     -3157: -8, -3565: -5, -2263: -5,
     -4294: -6, -2266: -6, -2193:  7,
     -1459:  7, -3055: -3, -3109: -3,
     -3648: -5,  -736: -5,  3648: -5,
       736: -5,  4351:  7,  3157:  7,
     -2209: -5, -3403: -5, -2185: -9,
     -4371: -9,  2185: -8,  4371: -8,
     -2194: -6, -3646: -6,  2194:  4,
     -3646:  4,  2188: -4, -4210: -7,
     -4318: -7, -3481: -4, -4315: -4,
      3481: -4,  4315: -4,  4210: -7,
      4318: -7, -4372: -2,     0:-20,
      2187: -9,     1: -8,     2: -9,
      1458: -9,     6: -9,   486: -8,
        18: -8, -2350:  6, -2242:  6,
      2350:  8,  2242:  8,  2206:  7,
      2604:  7, -2206:  3, -2604:  3,
      6319:  6,  2213:  6,     4: -8,
      2916: -8, -3154: -7, -3622: -7,
     -3643:  6, -4369:  6, -6556:  5,
     -3644:  5,  2193:  7,  1459:  7,
     -4370: -7, -5830: -7, -3517: -8,
     -2107:  9, -3775: -8,  3024: -8,
       112: -8, -3766:-10, -3274: -9,
      4295: -9,  4453: -9, -2265:  9,
}
```

3. Empirical Verification

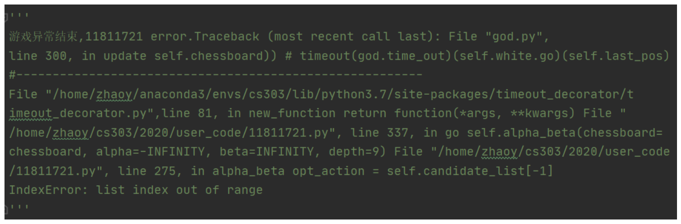
3.1. Dataset

For feasibility test, in addition to passing the ten tests on the platform, I also found two untested feasibility errors by playing against other players on the platform. This



```
'''
游戏异常结束,11811721 error.Traceback (most recent call last): File "god.py", line 212,
in check_chess if pos[0] < 0 or pos[0] >= self.chessboard_size
or pos[1] < 0 or pos[1] >= self.chessboard_size:
TypeError: 'NoneType' object is not subscriptable
'''
```

Figure 2. Feasibility Error 1

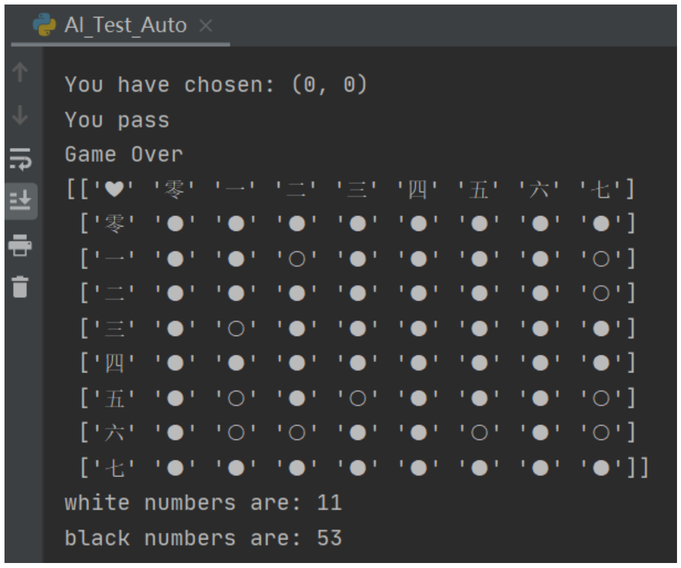


```
'''
游戏异常结束,11811721 error.Traceback (most recent call last): File "god.py",
line 300, in update self.chessboard)) # timeout(god.time_out)(self.white.go)(self.last_pos)
#-------------------------------------------------------
File "/home/zhaoy/anaconda3/envs/cs303/lib/python3.7/site-packages/timeout_decorator/t
imeout_decorator.py",line 81, in new_function return function(*args, **kwargs) File "
/home/zhaoy/cs303/2020/user_code/11811721.py", line 337, in go self.alpha_beta(chessboard=
chessboard, alpha=-INFINITY, beta=INFINITY, depth=9) File "/home/zhaoy/cs303/2020/user_code
/11811721.py", line 275, in alpha_beta opt_action = self.candidate_list[-1]
IndexError: list index out of range
'''
```

Figure 3. Feasibility Error 2

error is due to a type error for ”None” and an empty array. And the following are two bugs I found as Figure2 and Figure 3 .

For the points race, I saved more than 30 important matches with players on the platform. By observing and analyzing the cause of the failure and using the WZebra program to learn the optimal solution, the parameters were adjusted .



```
You have chosen: (0, 0)
You pass
Game Over
[['♥' '零' '一' '二' '三' '四' '五' '六' '七']
 ['零' '●' '●' '●' '●' '●' '●' '●' '●']
 ['一' '●' '●' '○' '●' '●' '●' '●' '○']
 ['二' '●' '●' '●' '●' '●' '●' '●' '○']
 ['三' '●' '○' '●' '●' '●' '●' '●' '●']
 ['四' '●' '●' '●' '●' '●' '●' '●' '●']
 ['五' '●' '○' '●' '○' '●' '●' '●' '○']
 ['六' '●' '○' '○' '●' '●' '○' '●' '○']
 ['七' '●' '●' '●' '●' '●' '●' '●' '●']]
white numbers are: 11
black numbers are: 53
```

Figure 4. Auto-Fight Test

For the round-robin race, I wrote an auto-fight program that lets my multiple AI programs to fight against

TABLE 4. PART OF AUTO-FIGHT TEST RESULTS

Test	Winner	Explanation
AI1 VS AI2	AI1	AI1 is no Zobrist hash
AI2 VS AI3	AI2	AI2 is no time limited
AI1 VS AI3	AI1	AI3 is reference group

TABLE 5. WEIGHT OF EVALUATION FUNCTION

Parameter	Value	Explanation
w_1	4	position
w_2	6	discs number
w_3	97	mobility
w_4	1150	stability
w_5	43	internal discs
w_6	215	edge attacking
$w_1(endgame)$	0	position
$w_2(endgame)$	100	discs number
$w_3(endgame)$	67	mobility
$w_4(endgame)$	1500	stability
$w_5(endgame)$	0	internal discs
$w_6(endgame)$	175	edge attacking

each other, as shown in the figure 4, and choose the one that has the advantage. The following Table 4 is a result of the auto-fight program. .

3.2. Hyperparameters

Through the introduction in the previous section, I used the ways of playing with players, saving and analyzing chess games, playing against myself and playing against standard procedures to carry out the reference adjustment. Finally, the super parameters trained are shown in the Table 5.

3.3. Result

All the test cases my program had passed in the usability test, my rank in the points race is 18 and my rank in the round robin is 78.

3.4. Analysis

In the points race stage, my program had a good performance, but in the round robin stage, it showed that my program did not have good stability and robustness, or it had a big problem in the optimization of efficiency. So I've summed up some of the strengths and weaknesses of my program.

3.4.1. Advantage.

The evaluation function is comprehensive and has a better decision-making method for special corner and edge cases.

The search depth and width are dynamically changed according to the remaining time, and the hash permutation table is used to save the situation, which can effectively reduce the search time.

3.4.2. Disadvantage.

The adjustment of parameters in the program is subjective. So some heuristic algorithms, such as ant colony algorithm and genetic algorithm, should be used for optimization.

The weight of the evaluation function is difficult to change dynamically with the situation. So it is impossible to make a valid judgment in all cases.

Minimax's limitations. For example, Monte Carlo method can be used to search deeper layers, but also more aware of the optimal solution.

3.4.3. Future.

Due to the tight time at the later stage, this project submitted a version that was not particularly excellent in a hurry in the end of the round robin stage. When I wrote the experiment report, it was also delayed to the last hour, and there were many things that were not improved and mentioned. What I regret most is that I did not make reasonable use of time to adjust parameters. If I used the algorithm package of genetic algorithm to adjust the weight between strategies or adopted monte Carlo method, I believe there will be better results.

In the next project, it is very important to prepare and set the plan in advance. The second is to determine the algorithm to be used and understand the advantages and disadvantages of the algorithm as well as the difficulties.

4. Acknowledgments

First of all, I would like to express my gratitude to our Computer Science and Technology department and Teacher Ke TANG, Xiaofen LU and Zhao YAO. I am also deeply indebted to a lot of previous research can be found on the Internet, which really raise me up.

References

[1] K.-F. Lee and S. Mahajan, "The development of a world class othello program," *Artificial Intelligence*, vol. 43, no. 1, pp. 21 – 36, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/000437029090068B

[2] G. Andersson, "Wzebra (othello program)," 2004.

[3] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293 – 326, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370275900193

[4] B. Bouzy and C. Tutorial, "Old-fashioned computer go vs monte-carlo go," in *IEEE Symposium on Computational Intelligence in Games (CIG)*, 2007.

[5] J. Mehat and T. Cazenave, "Monte-carlo tree search for general game playing," *Univ. Paris*, vol. 8, 2008.

[6] J. Festa and S. Davino, "" iago vs othello": An artificial intelligence agent playing reversi." in *PAI@ AI* IA*, 2013, pp. 43–50.