GOMOKU

Project 1 Report of

CS303 Artificial Intelligence

Department of Computer Science and Engineering

BY

Zhang Yifan

11711335

Table of Contents

1. Preliminaries

1.1 Problem Description

Gomoku is an abstract strategy board game and is also called Five in a row. Players must alternate turns to place a stone of their color (either black or white) on an empty intersection on the board, and the winner is the first player to form an unbroken chain of five stones horizontally, vertically or diagonally. Gomoku originated from Japan, and the name comes from Japanese gomokunarable. Go means five, and moku is a counter word for pieces and narable means line-up. Comparing to Renju, Gomoku has two kinds which are called freestyle Gomoku and standard Gomoku.

In this project, we focus primarily on freestyle Gomoku which refers to the game regulation that there are no forbidden moves. After analyzing, I believe this means the player which black color has a more significant advantage since he plays first and it’s more difficult for the other player to stop him/her from forming the unbroken chain of five black stones.

1.2 Problem Application

Artificial intelligence is a newly-developed and highly comprehensive frontier science of rapid development. Gambling and chess is one of the major artificial intelligence research areas. Gomoku here is AI applied in chess. This project involves reasoning, decision-making and planning, and is designed to think and play like a human-being. This area is a very popular topic around the world, and to my knowledge, there are several international tournaments worldwide and some famous virtual players such as Yixin, Goro and so on.

1.3 Software & Hardware

This project is written in Python 3.6 with editor JetBrains Pycharm.

1.4 Algorithms

This project uses the algorithm that mainly scores positions and keeps a record of the scores of each available position and chooses the best location at present. Different scores are assigned for different compositions of different colors, and many situations are taken into consideration, so this algorithm can accurately predict and choose the best move, which makes it very competitive even to the ones with game theory analysis.

2. Methodology

In this part, I will introduce the representation of the algorithm, the structure of my program, and the applied parameters in this project.

2.1 Representations

2.1.1 Notation

The goal is to get the score of each available position and choose the most beneficial one to be the next move. We represent each position as move, where move[0] means the x-coordinate of the board and move[1] means the y-coordinate. At each position,

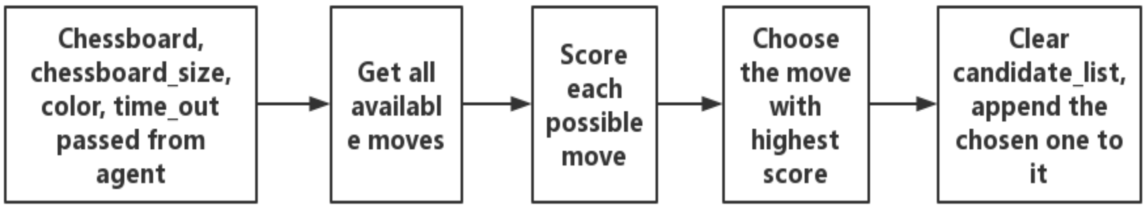
$$chessboard[x][y] = \begin{cases} 1, & this\ position\ is\ occupied\ by\ white\ move \\ 0, & this\ position\ is\ empty \\ -1, & this\ position\ is\ occupied\ by\ black\ move \end{cases}$$

According to the composition each color can form, each position is given a sum_score which is the sum of score of current color and the score of the opposite color. Then this program will append the position with the highest score to the candidate_list and return it to the platform.

2.1.2 Data Structure

NAME	**MEANING**
Candidate_list	A list of the most valuable positions which have been found
Possible_moves[]	A list of all the empty positions on the current chessboard
Scores[]	A list of the scores of the positions in possible_moves[]
Max_score	The highest score in scores[]
Best_move	The chosen position, which has the max_score

2.2 Model Design


Chessboard, chessboard_size, color, time_out passed from agent
Get all availabl e moves
Score each possible move
Choose the move with highest score
Clear candidate_list, append the chosen one to it

2.3 Details of Algorithm

2.3.1 Available moves

This function searches all the empty positions on the board from the center of the board to the edge. Chessboard[x][y] == 0 means this position is empty.

The pseudo code is as following:

```
Function get_possible_moves (chessboard, size):
possible_moves = []
For i = 0 to 7:
    IF i=0: points_c = center point
    ELSE:   points_c = points on the i-th circle from the center
    append each point to possible_moves

Return possible_moves
```

2.3.2 Get Compositions

This is the most complex function in this project, which calculates the scores of each available position.

First, we need to know if our color occupies this position, what composition we can achieve. We need to check four directions, ***ROW, COLUMN, FROM LEFT LOWER TO UPPER RIGHT, FROM UPPER LEFT TO LOWER RIGHT.*** Then we need to know what composition the opponent can achieve if the opponent occupies this, we also need to check all four directions.

Now let's take ROW for an example and see how this calculates the scores.

First, I want to explain some variables:

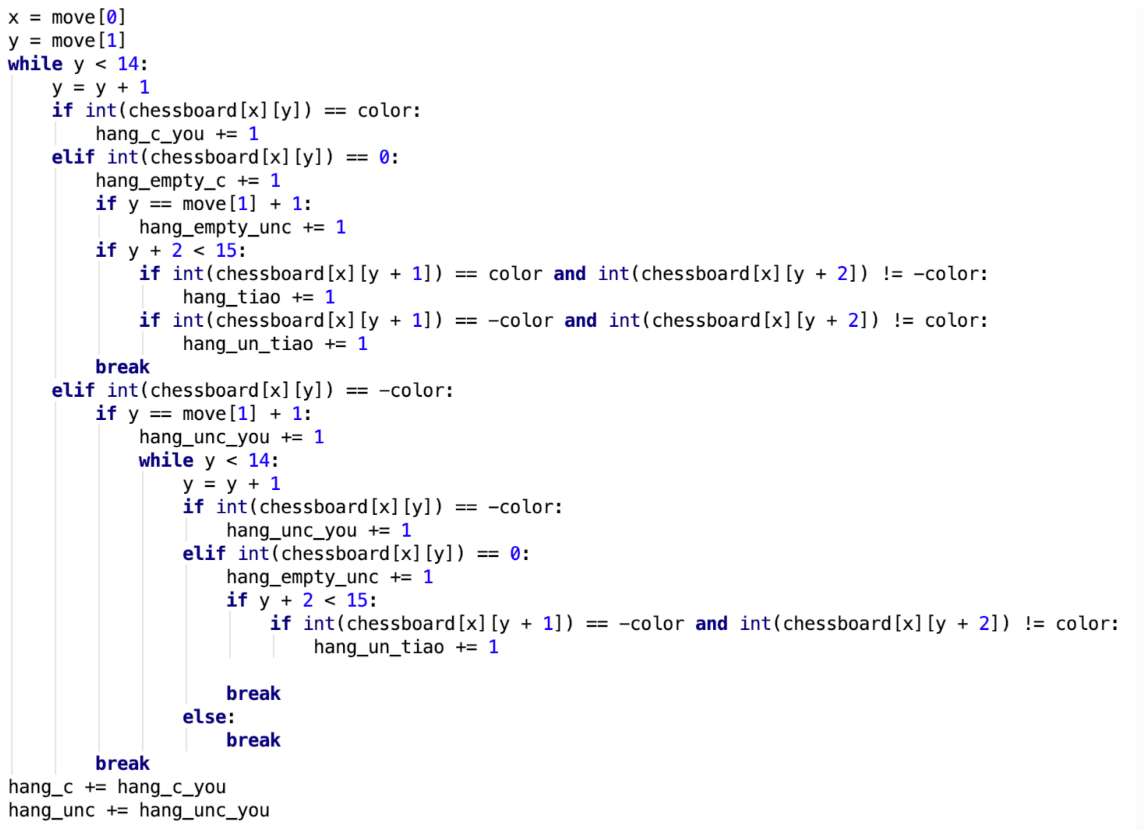
hang_c: how many our pieces are linked if it's occupied by current color
hang_c_zuo: how many pieces of current color are there on the left
hang_c_you: how many pieces of current color are there on the right
hang_unc: how many other pieces are linked if it's occupied by opposite color
hang_unc_zuo: how many pieces of opposite color are there on the left
hang_unc_you: how many pieces of opposite color are there on the right
hang_empty_c: how many sides of current color pieces are empty
hang_empty_unc: how many sides of opposite color pieces are empty
hang_tiao: whether there are pieces of current color that skip an empty position
hang_un_tiao: whether there are pieces of opposite color that skip an empty position

The source code:

```
while y > 0:
    y = y - 1
    if int(chessboard[x][y]) == color:
        hang_c_zuo += 1
    elif int(chessboard[x][y]) == 0:
        hang_empty_c += 1
        if y == move[1] - 1:
            hang_empty_unc += 1
        if y - 2 > -1:
            if int(chessboard[x][y - 1]) == color and int(chessboard[x][y - 2]) != -color:
                hang_tiao += 1
            if int(chessboard[x][y - 1]) == -color and int(chessboard[x][y - 2]) != color:
                hang_un_tiao += 1
        break
    elif int(chessboard[x][y]) == -color:
        if y == move[1] - 1:
            hang_unc_zuo += 1
            while y > 0:
                y = y - 1
                if int(chessboard[x][y]) == -color:
                    hang_unc_zuo += 1
                elif int(chessboard[x][y]) == 0:
                    hang_empty_unc += 1
                    if y - 2 > -1:
                        if int(chessboard[x][y - 1]) == -color and int(chessboard[x][y - 2]) != color:
                            hang_un_tiao += 1
                    break
                else:
                    break
        break
hang_c = hang_c_zuo
hang_unc = hang_unc_zuo
hang_unc += 1
hang_c += 1
```

This part checks the composition on the left of the current position.

As in the source code, after checking the number of pieces on the left of the position, hang_c should be 1 more than the hang_c_zuo since it includes the position itself.



```
x = move[0]
y = move[1]
while y < 14:
    y = y + 1
    if int(chessboard[x][y]) == color:
        hang_c_you += 1
    elif int(chessboard[x][y]) == 0:
        hang_empty_c += 1
        if y == move[1] + 1:
            hang_empty_unc += 1
        if y + 2 < 15:
            if int(chessboard[x][y + 1]) == color and int(chessboard[x][y + 2]) != -color:
                hang_tiao += 1
            if int(chessboard[x][y + 1]) == -color and int(chessboard[x][y + 2]) != color:
                hang_un_tiao += 1
        break
    elif int(chessboard[x][y]) == -color:
        if y == move[1] + 1:
            hang_unc_you += 1
            while y < 14:
                y = y + 1
                if int(chessboard[x][y]) == -color:
                    hang_unc_you += 1
                elif int(chessboard[x][y]) == 0:
                    hang_empty_unc += 1
                    if y + 2 < 15:
                        if int(chessboard[x][y + 1]) == -color and int(chessboard[x][y + 2]) != color:
                            hang_un_tiao += 1

                    break
                else:
                    break
        break
hang_c += hang_c_you
hang_unc += hang_unc_you
```

This part checks the composition on the right of the current position. The tricky part is that the execution upper part code changes values of x and y, so their values should be reassigned.

Then we need to check the other three directions. The algorithms are similar, but the boundary conditions and variations of coordinates are different.

2.3.3 Parameters

After checking the compositions, then we need to collect data from all directions and start calculating scores.

According to the number of pieces, color, whether sides are empty, the scores are assigned as follows:

Number of pieces	5	4	4	3
Number of empty sides	2/1/0	2	1	2
Current color	1000000	10000	3000	1700
Opposite Color	30000	4000	1800	1000

Number of pieces	3	2	2	1	1
Number of empty sides	1	2	1	2	1
Current color	50	70	20	5	2
Opposite Color	40	60	10	3	1

And then start the scoring, we still take **ROW** as an example:

```
if hang_c >= 5:
    sum_score += 1000000
    if hang_c_zuo > 0 and hang_c_you > 0:
        sum_score += 3000000
elif hang_empty_c > 0:
    num_c[(4 - hang_c) * 2 + 2 - hang_empty_c] += 1
if hang_unc >= 5:
    sum_score += 30000
    if hang_unc_zuo > 0 and hang_unc_you > 0:
        sum_score += 50000
elif hang_empty_unc > 0:
```

2.3.4 Special considerations

1. Take **ROW** as an example, if hang_tiao > 0, we need to add it to the hang_c. And if hang_un_tiao > 0, we need to add it to the hang_unc. But they should only be add once, which means if there are skip pieces on the both ends, we only count once.

```
if hang_tiao == 2:
    hang_tiao = 1
hang_c += hang_tiao
```

2. ***num_c*** and ***num_unc*** are used to store the number of compositions, if some composition occurred more than twice in four directions, we should only count twice since it doesn't influence the possibility of winning if it's twice or more than twice.

```
for i in range(0, 8):
    if num_c[i] <= 2:
        sum_score += scores_c[i] * num_c[i]
    else:
        sum_score += scores_c[i] * 2
    if num_unc[i] <= 2:
        sum_score += scores_unc[i] * num_unc[i]
    else:
        sum_score += scores_unc[i] * 2
```

3. If the current position is in the middle of other pieces, it should have additional score to differentiate from the other positions.

```
if hang_c == 4 and hang_empty_c == 2 and hang_c_zuo > 0 and
hang_c_you > 0:
        sum_score += 5000
if hang_unc == 4 and hang_empty_unc == 2 and hang_unc_zuo > 0
and hang_unc_you > 0:
        sum_score += 2000
if hang_c == 3 and hang_empty_c == 2 and hang_c_zuo > 0 and
hang_c_you > 0:
        sum_score += 700
if hang_unc == 3 and hang_empty_unc == 2 and hang_unc_zuo > 0
and hang_unc_you > 0:
        sum_score += 400
```

3. Empirical Verification

3.1 Empirical Performance

3.1.1 Platform Test

This program passed all 25 test cases on the Gomoku platform. And it also did well in the qualifying tournament.

3.1.2 Parameter adjusting

To adjust the parameters to achieve the best performance, I analyzed the importance of different compositions, including how many times they appeared, and I concluded some relationships of each parameter.

Number of pieces	5	4	4	3
Number of empty sides	2/1/0	2	1	2
Current color	a	c	e	g
Opposite Color	b	d	f	h

Number of pieces	3	2	2	1	1
Number of empty sides	1	2	1	2	1
Current color	i	k	m	o	q
Opposite Color	j	l	n	p	r

1. $2a > a > 2b > b > 2c$ and $c > 2d$ and $2e$ and $2f$ and $d > e > f$

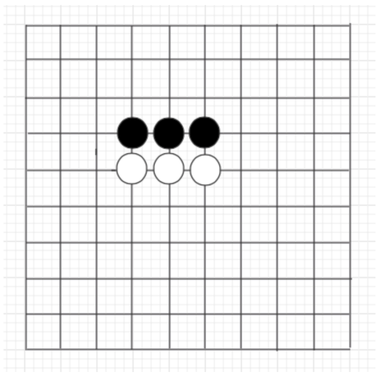
2. $2g > g > 2h > f > h$

3. $2g > e$

3.1.3 Sample Test

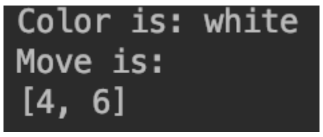
I wrote a test program to print out the move my algorithm chooses and see if they are correct.

TEST CASE 1

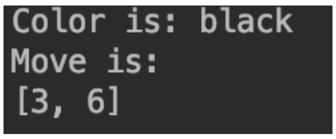


When given this board, and we test the algorithm with both colors:

The program chooses:



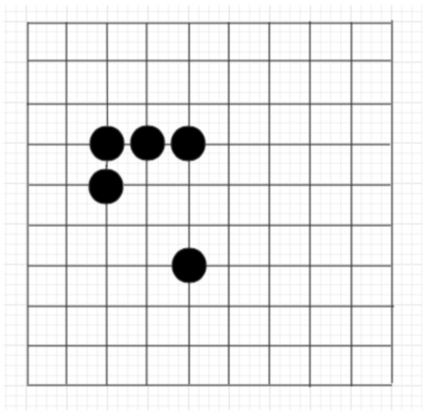
```
Color is: white
Move is:
[4, 6]
```



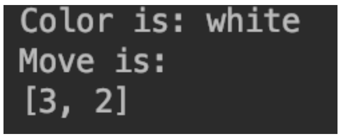
```
Color is: black
Move is:
[3, 6]
```

as shown in this case, in order to win, the algorithm would first form a HUOSI instead of stopping the opposite from forming one.

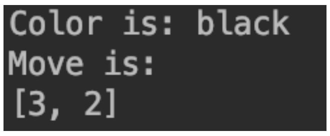
TEST CASE 2



The output:



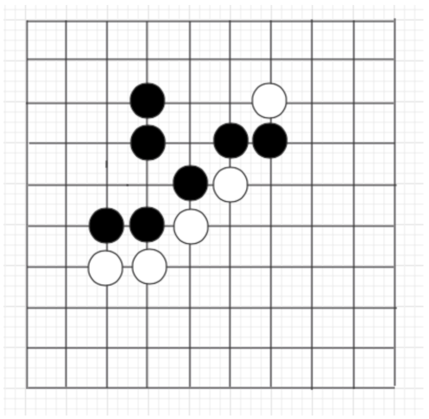
```
Color is: white
Move is:
[3, 2]
```



```
Color is: black
Move is:
[3, 2]
```

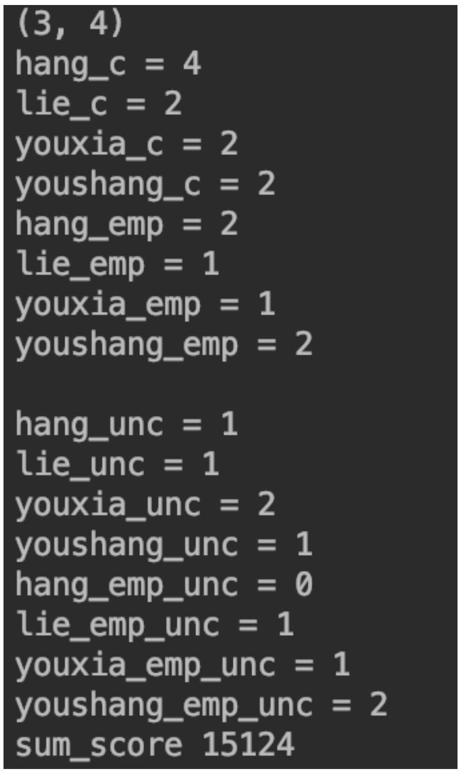
Now, regardless of the color, [3,2] is the best move since it can either win or stop the opponent from winning.

TEST CASE 3

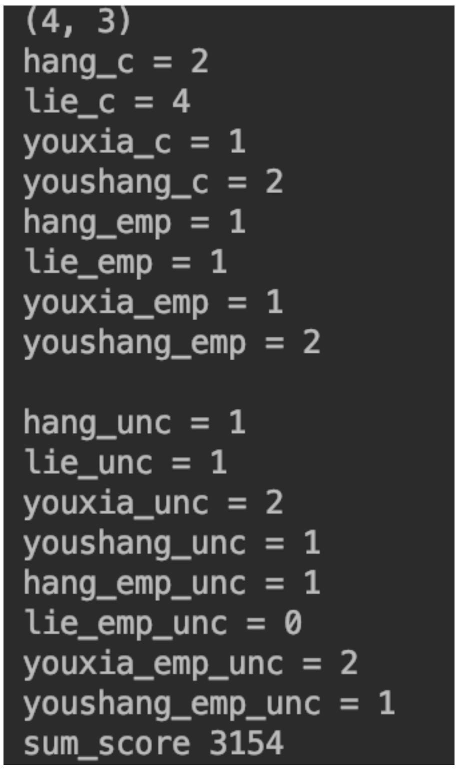


When giving this chessboard, regardless of the color, [3,4] is the best move. Now we print the scores of some significant moves and see how this program works:

If the current color is black:



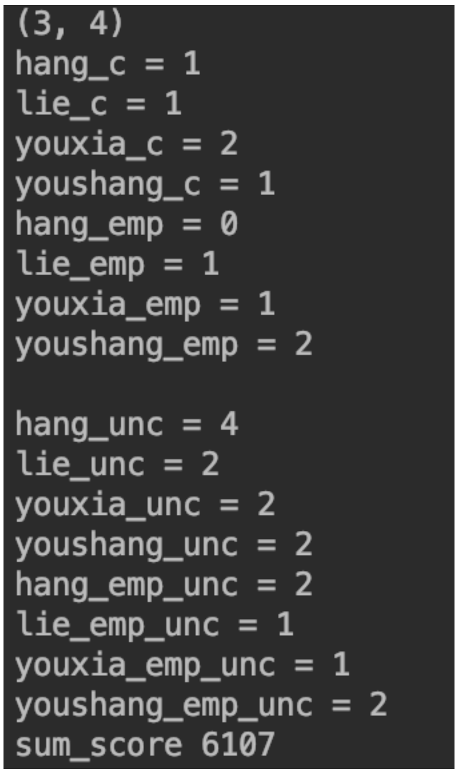
```
(3, 4)
hang_c = 4
lie_c = 2
youxia_c = 2
youshang_c = 2
hang_emp = 2
lie_emp = 1
youxia_emp = 1
youshang_emp = 2

hang_unc = 1
lie_unc = 1
youxia_unc = 2
youshang_unc = 1
hang_emp_unc = 0
lie_emp_unc = 1
youxia_emp_unc = 1
youshang_emp_unc = 2
sum_score 15124
```



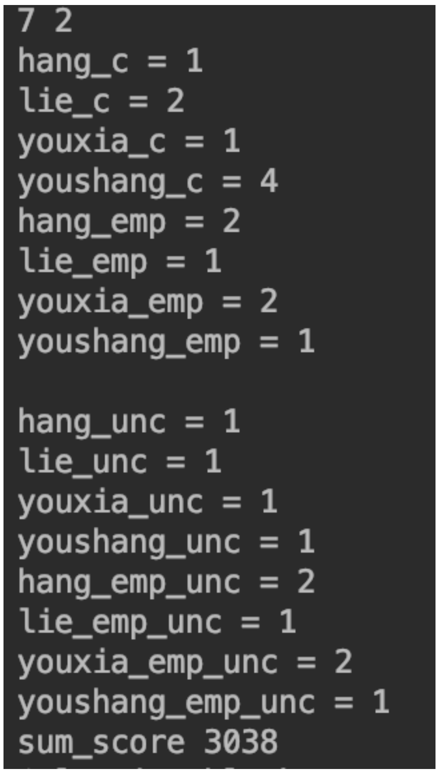
```
(4, 3)
hang_c = 2
lie_c = 4
youxia_c = 1
youshang_c = 2
hang_emp = 1
lie_emp = 1
youxia_emp = 1
youshang_emp = 2

hang_unc = 1
lie_unc = 1
youxia_unc = 2
youshang_unc = 1
hang_emp_unc = 1
lie_emp_unc = 0
youxia_emp_unc = 2
youshang_emp_unc = 1
sum_score 3154
```

It will choose [3,4] to forming the must-win situation. At [4,3], it can also form four linked black pieces, but one side is not empty and might be discouraged.

If the current color is white:



```
(3, 4)
hang_c = 1
lie_c = 1
youxia_c = 2
youshang_c = 1
hang_emp = 0
lie_emp = 1
youxia_emp = 1
youshang_emp = 2

hang_unc = 4
lie_unc = 2
youxia_unc = 2
youshang_unc = 2
hang_emp_unc = 2
lie_emp_unc = 1
youxia_emp_unc = 1
youshang_emp_unc = 2
sum_score 6107
```



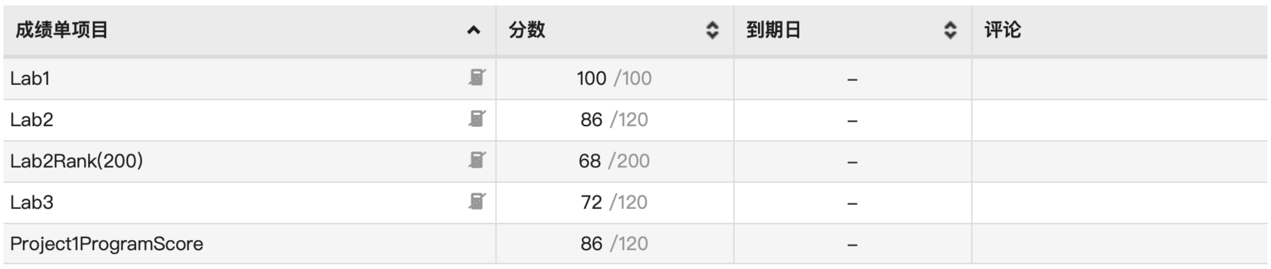
```
7 2
hang_c = 1
lie_c = 2
youxia_c = 1
youshang_c = 4
hang_emp = 2
lie_emp = 1
youxia_emp = 2
youshang_emp = 1

hang_unc = 1
lie_unc = 1
youxia_unc = 1
youshang_unc = 1
hang_emp_unc = 2
lie_emp_unc = 1
youxia_emp_unc = 2
youshang_emp_unc = 1
sum_score 3038
```

It will choose [3,4] to stop the Black pieces from immediate winning. And if it chooses [7,2], we still can be discouraged and we have to place [3,4] at last, so it's not the best choice.

3.2 Conclusion

From the empirical verification, I found that this program can correctly identify the "importance" of each available position and choose the best move. This program works fine when game reaches "game ball", for example, making the winning move or stop the opponent from winning. However, on other occasions, this might not work as well as the "MaxMinimal Algorithm" since it includes game theory analysis and has a bigger chance of winning. However, the adjusted parameters can somewhat "predict" the future game.

3.3 Final Results



成绩单项目	分数	到期日	评论
Lab1	100 /100	–	
Lab2	86 /120	–	
Lab2Rank(200)	68 /200	–	
Lab3	72 /120	–	
Project1ProgramScore	86 /120	–	

4. References

[1] Luo Pan. (2016, June 09). Five in a row [Online]. Available: https://blog.csdn.net/ChinaJane163/article/details/52599787