Project 1 Reversi

Name:Yi Xiang SID:11912013
Department: Department of Computer Science and Engineering
Institution: Southern University of Science and Technology
Email: 11912013@mail.sustech.edu.cn

Contents

1. Preliminaries

1.1. Hardware & Software

This project is written in *Python* with editor *Pycharm*. The main testing platform is *Windows 10 Home Edition* (version 20H2) with Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz of 4 cores and 8 threads, the memory is 16GB. And the develop platform is *Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-151-generic x86_64)* with Intel(R) Xeon(R) Gold 6278C CPU @ 2.60GHz with 8 cores and 16 threads, the memory is 16GB.

Both of the *python* version are 3.8.8 and the *numpy* module's version is 1.20.1 .

1.2. Problem Description

Reversi is a **two-person zero-sum information-symmetry** board game. Two players alternate on an 8-by-8 board until there are no more pieces left to play. The rules are very simple. At first, two pieces from each side are placed alternately in the center of the board, shown as 1. A player can then only place a piece in a valid position, which is defined as a piece in a valid position where there are and only opposing pieces on the line with at least one of their own pieces in eight directions. At this time, will be two own pieces sandwiched by the other side of the piece flipped (into their own pieces). Neither side can choose not to do anything (in other words, each player must do something if they can do something). Unlike traditional reversi-chess, the condition for winning the game is to end the game with fewer pieces on your side than on the other side.

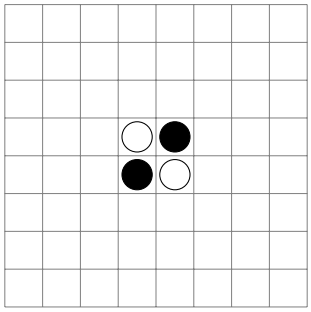

Figure 1. The initial state

1.3. Problem Applications

Chess AI research focuses on game theory, or the game between two players. Such research has certain applications in urban planning and finance. In the existing studies, Yang Yuhong and Chen Zhong used evolutionary game model to describe the competition and cooperation relationship between enterprises[1], and Wang Yan and Dewen used evolutionary game theory to study the game process of government and enterprises in the continuous case of pollution level[2].

2. Methodology

2.1. Notation

As the figure shown as above 2. The main structure and the corresponding comment is shown as the table below.

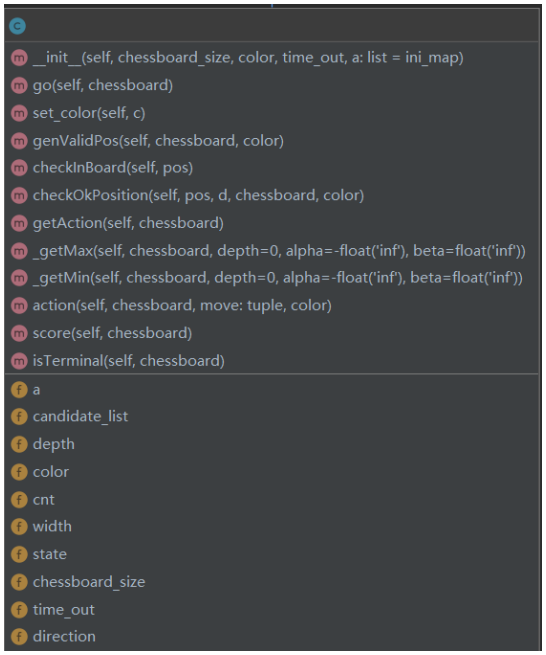

Figure 2. The main structure of the project

The notation	meaning
go(chessboard)	Generate the valid moving of the AI, the last position of the list is the move that the AI chooses. Return nothing.
genValidPos (chessboard, color)	Get all the valid moving of one player, the argument color determine which player we choose. Return the list of moves.
checkInBoard(pos)	Check the whether the position pos is in the chessboard. Return a boolean.
checkOkPosition (pos, d, chessboard, color)	Check whether the postion is ok to put down a chess. The argument d is a vector where the direction that is being judged. Return a boolean.
getAction(chessboard)	Get the best move of the AI, always come up with _getMax() and _getMin(). Return a tuple, as the move.
_getMax(chessboard, depth, alpha, beta)	Get the max value and the corresponding move of the AI, return a float value and a tuple.
_getMin(chessboard, depth, alpha, beta)	Get the max value and the corresponding move of the opponent, return a float value and a tuple.
action (chessboard, move, color)	Generate a possible move and act it. Return a temp chessboard with the origin board act the move move with color.
score(chessboard)	Get the score of the chessboard. Return a float value.
isTerminal(chessboard)	Check if the game is end. Return a boolean.

TABLE 1. THE NOTATION OF THE PROJECT

2.2. Date Structure

There are not many complex data structures are applied. The *candidate_list* is a list of the positions which have found. The *MyMap* is a 2-dim matrix which specfic the value of each block of the chessboard. The *chessboard* is a 2-dim matrix which specfic the game state of that time. The *direction* is a list of tuple of the valid direction (the 8 directions of the connecting sides), which is used to action one move in the chessboard and obtaion the valid possible move.

2.3. Model Design

Firstly, I use **Brute Force** to find out all valid position for the agent to go, and successly pass the first state of the battle. I append them all to the candidate_list.

Algorithm 1 The algorithm to find the valid position to go.

Input: The current game-state: *chessboard*
Output: The valid position for the agent to go

$idx \leftarrow$the places that the chessboard is empty
$result \leftarrow []$ ▷ Initial an empty list
for each pos in idx **do**
  for each direction in directions **do**
    if The player can flip the oppsite chess in such position and such position **then**
      append the pos in the result list
      break
    end if
  end for
end for
return result

Mainly, I use the alpha-beta pruning algorithm to obtain the best possible move of the agenet. For the method of getting the value of each game-states, I use the standard **multipy** in the package numpy to construct a checkerboard situation and evaluate the dot product of the matrix.

Algorithm 2 The algorithm to get best move of the agent.

Input: The current game-state: *chessboard*
Output: The best move of the agent

function GETACTION
  $gamestate \leftarrow generateState(loopCnt)$ ▷ Get the game state
  bestValue, bestAction = getMax(chessboard, depth=0)
  return bestAction
  function GETMAX(chessboard, depth, α, β)
    if depth==end or chessboard is terminal state **then**
      return score(chessboard) ▷ Return the score of the leaves node of the searching tree.
    end if
    $value \leftarrow -\inf$
    for each act in action_list **do**
      new_value $\leftarrow getMin(action(chessboard, act)), depth + 1, \alpha, \beta)$

```
            value ← max(value, new_value)
            if α ≥ β then
                break                                   ▷ β cut-off
            end if
            return value, act
        end for
    end function
    function GETMIN(chessboard, depth, α, β)
        if depth==end or chessboard is terminal state then
            return score(chessboard)        ▷ Return the score of the leaves node of the searching tree.
        end if
        value ← inf
        for each act in action_list do
            new_value ← getMax(action(chessboard, act)), depth + 1, α, β)
            value ← min(value, new_value)
            if α ≤ α then
                break                                   ▷ α cut-off
            end if
            return value, act
        end for
    end function
end function
```

2.4. Detail of Algorithm

For more detail of the searching algorithm, I use a **heuristic algorithm** to simply the problem. This following code will be append to the 2 at line 9 and line 24. The algorithm is to pretreatment the action_list and find the first **width** action to make the time usage less.

Input: The candidate_list
Output: The intensified candidate_list

```
1: enhance each action of the scoring function
2: sort the list with the key score in decreasing order
3: return The shortlisted candidate_list ▷ The shortlisted candidate_list's length is width
```

Following, I use genetic algorithm to obtain evaluation matrix. The origin of the initial population is a combination of my subjective will and the chess score in the course reference materials. It took me quite a bit of time because we won in the opposite way to the traditional Othello.

From the first generation of fathers and mothers, we get the first generation of children. These children will have some random variation (since the original parents are identical, crossover makes no sense). We then loaded the children's scores into the AI's evaluation matrix and polled them with the same algorithm. Due to machine limitations, a large number of polls in a single round can consume a lot of time. Considering that each move takes up to five seconds, each side uses about 120 moves in total, which means that the maximum length of a single game is about 10 minutes. In the case of 100 offspring, a single round of genetic polling will consume about $C_{100}^2 \times 10/24/60 \approx 34.375$ days. Counting the time spent iterating and maintaining the checkerboard, such inheritance doesn't make sense. Therefore, I adopted the method of reducing the population, and compressed the search layer number and width, and tried to get a suitable evaluation matrix in an effective time.

Algorithm 4 The Genetic Algorithm-From generation to generation

Input: The Parents
Output: The Childrens

```
1: children ← []                                        ▷ Initial the children
2: for each child in children do
3:     for each value in the child score map do
4:         value ← random.choose(parents)               ▷ Choose randomly from mother or father.
5:         flag ← randomInt(0, 50)
6:         if flag % 17 == 0 then
7:             value ← value + 50 - randomInt(0, 100)
8:         end if                                       ▷ Certain probability of variation.
9:     end for
10: end for
11: return children
```

In the algorithm above 4, I can obtain the generation from one to one randomly. Then is to get the best children in one generation. I define a module **Util.py** to create a battle system for two children to battle each other, both sides take turns at blackening. Then I statistical the winning counts of each children. Hint: there are many function I can continuing use as I have implementationed in **ai.py**.

Algorithm 5 The Genetic Algorithm-Battle System

Input: child A, B
Output: Result

```
1: A ← white
2: B ← black
3: cnt ← 0
4: chessboard ← initial_board
5: while The game is not end do
6:     cnt ← cnt + 1
7:     if cnt % 2 == 0 then
8:         call A.go(chessboard)
9:         move ← A.candidate_list[−1]
10:    else
11:        call B.go(chessboard)
12:        move ← B.candidate_list[−1]
13:    end if
14:    if move exist then
15:        action(chessboard, move)
16:    end if
17: end while
18: result ← count of white and black in chessboard
19: exchange the white and black then do the same thing
20: return result
```

After ending the wheel wars, we can get the win status of all the offspring. Then we take the first two offspring and mate with them. Get a new generation of sons for iteration. Note that I would keep the parents of this generation *running* with the next generation in order to ensure that an

unexpected mutation would cause the offspring to adapt to a precipitous descent. I end up with the evaluation matrix that I want.

3. Empirical Verification

In the process of this project, I did not submit many codes at the beginning of the second phase after the end of the first phase. In my opinion, it has the following advantages:

- Avoid the influence of ranking fluctuations on their own strategies and avoid utilitarian referrals.
- The online evaluation system sometimes leads to abnormal code test results due to network fluctuations or server abnormalities (the student assistant once mentioned problems with the server in the course group), thus causing meaningless debugging, which is very uneconomical.
- Be responsible for own studies, artificial intelligence is just one of the major required courses, I want to be able to last the code in the form of homework at once submitted (and project grading policy is and my idea match, namely the initial stage, the score does not affect the score, only affect the score in polling).

However, there are also many disadvantages, which I have some reflections after the end of the second stage:

- There is no other measure of strength other than playing chess.
- Easy to fall into the embarrassing situation of overfitting.
- Online battle platform can provide log files of match information (which I learned at the end), which is a very valuable learning resource and I wasted it.

I make up for this is that adding the previous years' code into the battle system. It join against list but bot to join the breeding. Just as with-runners to strengthen my own code.

3.1. Data Set

The project is divided into two phases. In the first phase, I used the test samples provided by the platform and my own test samples to conduct preliminary tests. And after I passed the platform test, I didn't update my test sample (thinking the platform test sample was relatively complete). Shown as figure 3.

In the second phase, I use the self battle system to test the strength of the agent, the evaluation criteria is the winning rate of the agents. The following section will show the winning rate with the agent and random agent.

3.2. Performance measure

Base on my experience of chess games, we can divide chess into early, middle and late state. The first thing we should do is to measure the number of valid positions. At this time, I use two random agents. The **random** means that they choose the position randomly, but the position they choose are always valid. Otherwise, the game cannot go on. I do such things for 100 times and get the average number of valid positions in each game, shown as the figure below.

```python
class MyTestCase(unittest.TestCase):
    def test1(self):
        ai = AI(8, 1, 5)
        ai.go(ini_board)
        self.assertEqual(ai.candidate_list, test1_ans)

    def test2(self):
        ai = AI(8, 1, 5)
        ai.go(board1)
        self.assertEqual(ai.candidate_list, test2_ans)

    def test3(self):
        t = npy.array([[0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 1, 0, 0, 0, 0, 0],
                       [0, 0, -1, -1, -1, 0, 0, 0],
                       [0, 0, 0, -1, -1, 0, 0, 0],
                       [0, 0, 0, -1, -1, -1, -1, 0],
                       [0, 0, 0, -1, -1, -1, -1, 0],
                       [-1, -1, -1, -1, -1, -1, -1, 1]])
        ai = AI(8, -1, 5)
        ai.go(t)
        print(ai.candidate_list)


if __name__ == '__main__':
    unittest.main()
```

Figure 3. Test cases

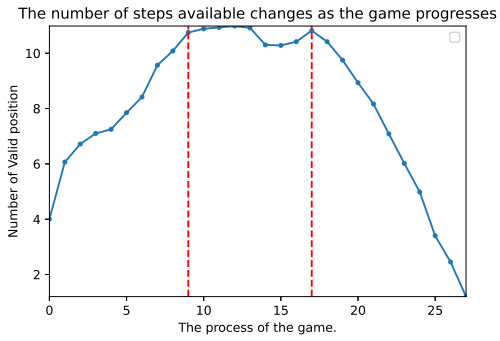

Figure 4. The number of steps available changes as the game progresses

We can obtaion the complexity of the agent search changes with the situation of game progresses. Then I define three game state for my agent. The Start game, the Middle game and the End game. The definition is shown as the following table (**cnt** means the count of steps did the game take).

For the three different game state, I use different ways to search the best action of the agent.

Start	In this state, since the number of valid position is not so much, I enlarge the searching width and depth, in order to maximize the value of best action.
Middle	In this state, unlike the opening, I shrink the width and depth. The reason was that I could pass the local

State	Define
Start game	0<cnt<10
Middle game	10≤ cnt <20
End game	Other situations

TABLE 2. THE DEFINITION OF THE **GAME STATE**

tests to limit the time, but never passed the tests after submitting the platform (it turned out to be platform load). So in order to make the submission work, I took a very conservative approach, drastically reducing the width and depth of the search in the middle of the panel to pass the basic tests. But the result is a much lower win rate.

End In this state, since the game is nearly end, and the complexity become smaller and smaller, I enlarge the searching width and depth again. More importantly, I added the judgment to check the end of the game terminal, returning a very large weight when searching for the end of a victory for your side and a very small weight when searching for the end of a victory for your opponent. This setting make sure the agent can kill the game decidedly.

From the time measuring, by doing the test of the AI self battle. I obtain the figure which show the steps for the agent cost. You may ask why I choose **28** as the totally steps. It from experience. We know that the maximum count of steps is 30. So I just choose a number smaller than it.

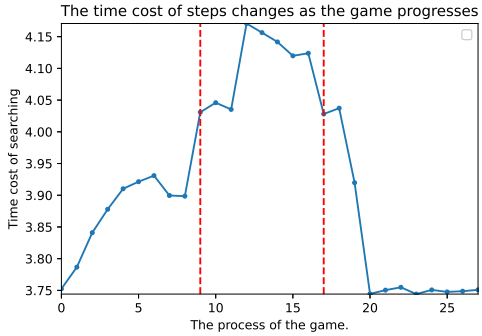

Figure 5. The time cost of steps changes as the game progresses

After dividing the situation, I am able to get a relatively reasonable time use interval locally. To make sure I passed the feasibility test, I keet it to just 4.2 seconds, which sacrificed some search accuracy, but I think it is worth it.

3.3. Hyperparameters

The main **Hyperparameters** of this project is the *Evaluation matrix*. In this section, I will explain how I obtaion the evaluation matrix.

Firstly, I need to initial the parent for the Genetic algorithm. Totally randomly generate it is not feasible. Because the evaluation matrix will be very ugly and useless. I got the initial evaluation matrix based on my own experience and the game with random agents.

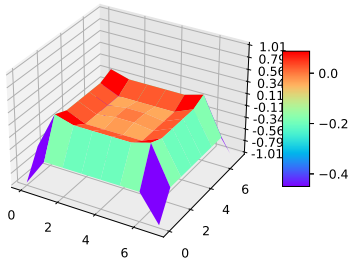

Figure 6. The initial evaluation matrix

After that I call algorithm 4 and 5 to evolve the generation. Due to the limitation of computation power and time, only 5 iterations were carried out. The Genetic algorithm does not converge, so the result is not very ideal. This leaves my final evaluation function almost unchanged. However, I have benefited from the experiment.

3.4. Experimental Result

Some of the experimenttal result is shown above like Fig.4 and Fig.5. I won't go into details of them in this section.

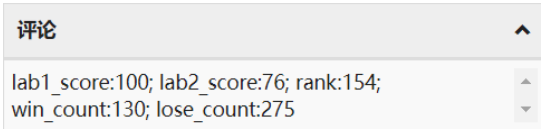

Figure 7. The final standing: screen-shoot from sakai

From the comment I can get the following result. I passed all the feasibility tests in phase one, and I won 130 and failed 275 in phase two. **The winning percentage is about 32 percent** and rankings for 154.

3.5. Conclusion

In general, I used alpha-beta pruning algorithm, combined with genetic algorithm to obtain a set of evaluation functions. And adopted different strategies in different stages of the game. Unfortunately, the final score was not ideal. Consider areas for improvement:

1) Change the evaluation function dynamically according to the changing situation.
2) Consider introducing the concept of stabilizers.
3) The concept of odd and even bits is introduced, which is similar to the above point, in order to reduce the occurrence of stabilizers.

Acknowledgments

Thanks to the student-assistants for their wonderful questions and reliable online battle system. Thanks to Yuan Bo and Zhao Yao for their wonderful courses, thanks to Zhao Yao for his help in the LAB course.

References

[1] YANG Yuhong, CHEN Zhong. Evolutionary Equilibrium Analysis on Coopetition in Intermediary Industry[J].System Engineering-Theory Methodology Applications,2006(01):26-31+38.

[2] WANG Yan, DING Dewen. Game analysis of public participation in environmental protection[J].Journal of Dalian Maritime University, 2006(04):19-22+35.