Report for Project Gomoku

Name ID: Yutong Wang 11611808
Department: Computer Science and Technology
Institution: Southern University of Science and Technology
Email: 11611808@mail.sustc.edu.cn

***Abstract*—Gomoku is an abstract strategy board game which is traditionally played with Go pieces (black and white stones) on a Go board, using 15×15 of the 19×19 grid intersections[1]. In the first project of SUSTech CS303 Artificial Intelligence, we are asked to implement a simple AI agent which is able to make decisions on the next step during the online game. Using brute force method to compute every possible step on the chessboard is a waste of time and memory. So optimization is important.**

I. Preliminaries

In this part, I will briefly describe the software and algorithm I used for the project. For the Algorithm part, I introduce the general idea of my implementation, the detail of which will be discussed soon in Section Methodology.

A. Software

This project is written in Python(version 3.6.6) using the IDE PyCharm. The library being used includes only numpy.

B. Algorithm

The algorithm uses one-level basic search. To find the best step for the greatest probability of victory, I made efforts on the following two questions:
i. Which positions on the chessboard should I take into consideration?
ii. When I look at one position, how to evaluate its probability of victory?

For the first question, it is obviously time-consuming to take every empty position into consideration. So my method is to only check all the empty positions which *have neighbor*. The definition of *having neighbor* will be illustrated soon. And for the second question, one method is to find out all the chess shapes on the chessboard after filling this position with one color, and evaluate the probability of victory for the whole chessboard. Of course this is also not efficient and my solution is to evaluate only part of the chessboard. Details will be shown in the next section.

II. Methodology

This part includes the representation, the architecture of codes and more details of my algorithm.

A. Representation

Representation of some variables need explanation.

- **chess status** :
  - *COLOR_BLACK* : black chess, value is -1.
  - *COLOR_WHITE* : white chess, value is 1.
  - *COLOR_NONE* : empty position, value is 0.
- **scores for different chess shapes** : definition of different score levels.
  - *level_one* : 100000
  - *level_higher_two* : 35000
  - *level_higher_two1* : 30000
  - *level_higher_two2* : 29000
  - *level_higher_two3* : 28000
  - *level_higher_two4* : 27000
  - *level_higher_two5* : 26000
  - *level_higher_two6* : 25000
  - *level_higher_two7* : 24000
  - *level_higher_two8* : 23000
  - *level_two1* : 10000
  - *level_two2* : 9000
  - *level_three1* : 5000
  - *level_three2* : 4900
  - *level_four1* : 1000
  - *level_four2* : 900
  - *level_higher_five1* : 600
  - *level_higher_five2* : 590
  - *level_five* : 500
  - *level_six* : 400
  - *level_seven* : 100
  - *level_eight* : 90
  - *level_nine* : 50
  - *level_ten* : 10
  - *level_eleven* : 9
  - *level_twelve* : 5
  - *level_thirteen* : 2
  - *level_fourteen* : 1
- **chessboard** : A numpy 2d array (15×15)
- **chessboard_size** : Value is 15.
- **candidate_list** : Add my decision into the candidate_list.
- **new_pos** : My final decision of the next position.
- **is_me** : If *is_me* is **True**, the decision will be based on my side. If *is_me* is **False**, the decision will be based on opponent side.

B. Architecture

Here list all functions in the codes with their usage.

- Given :

    - ***__init__*** : Initialize the value of variables *chessboard_size*, *color*, *time_out*, *candidate_list*, and *new_pos*.
    - ***go*** : Pass me the current state of the chessboard and let me make dicision. Then put my decision into *candidate_list*.
- Self_define :
    - ***is_blank*** : Check if the chessboard is blank at all positions.
    - ***ai*** : Clone an editable chessboard and pass it to the function *decide_new_pos* to make decision.
    - ***generate_kids*** : Passing the chessboard as parameter and this function will return a list including all the empty positions which need taken into consideration.
    - ***has_neighbor*** : Passing the position's index (i, j) as parameter and justify if the position *has neighbors*.
    - ***decide_new_pos*** : Decide the best position for next step.
    - ***evaluate*** : Return a evaluation score for a certain position.
    - ***one_direction*** : Determine the chess shape of a list. *Ex. [0, -1, -1, -1, -1, 1] is a chess shape of die four for black chess.*

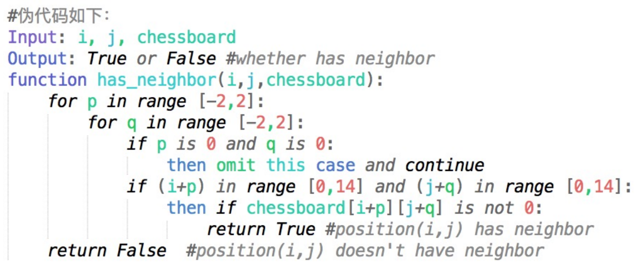
C. Detail of Algorithm

In this part, more details for the two questions listed in *Section 1 Algorithm* will be discussed.

Definition 1 (Having neighbor). For any given position on the chessboard, if there exists a non-empty position with the distance no larger than two, then the given position is called *having neighbor*.

Example 1. For a given postion *(i, j)*, if there is a non-empty position of the following: *(i-2, j-2), (i-2, j-1), (i-2, j), (i-2, j+1), (i-2, j+2), (i-1, j-2), (i-1, j-1), (i-1, j), (i-1, j+1), (i-1, j+2), (i, j-2), (i, j-1), (i, j+1), (i, j+2), (i+1, j-2), (i+1, j-1), (i+1, j), (i+1, j+1), (i+1, j+2), (i+2, j-2), (i+2, j-1), (i+2, j), (i+2, j+1), and (i+2, j+2)*, then we can say that position *(i, j)* has neighbor.

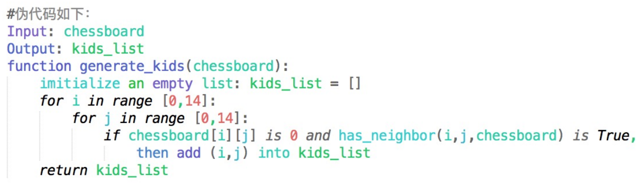
Algorithm 1 has_neighbor


```
#伪代码如下:
Input: i, j, chessboard
Output: True or False #whether has neighbor
function has_neighbor(i,j,chessboard):
    for p in range [-2,2]:
        for q in range [-2,2]:
            if p is 0 and q is 0:
                then omit this case and continue
            if (i+p) in range [0,14] and (j+q) in range [0,14]:
                then if chessboard[i+p][j+q] is not 0:
                    return True #position(i,j) has neighbor
    return False  #position(i,j) doesn't have neighbor
```

Then let's discuss the first question released in *Section 1 Algorithm* (aka. Which positions on the chessboard should I take into consideration?).

To avoid exhaustly computing all the empty positions on the chessboard, I design the rule that only takes the positions which *have neighbors* into consideration. Since the function *generate_kids* will return the list which includes all the empty postions need consideration, all of the positions in the return list satisfy the requirement of *having neighbors*.

Algorithm 2 generate_kids

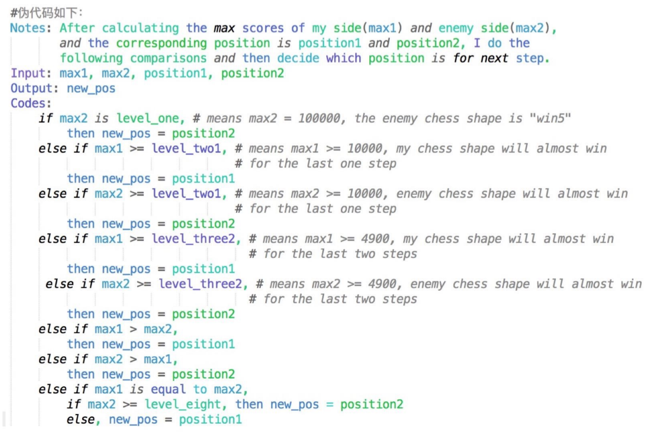

```
#伪代码如下:
Input: chessboard
Output: kids_list
function generate_kids(chessboard):
    imitialize an empty list: kids_list = []
    for i in range [0,14]:
        for j in range [0,14]:
            if chessboard[i][j] is 0 and has_neighbor(i,j,chessboard) is True,
                then add (i,j) into kids_list
    return kids_list
```

Then comes the solution of the second question in *Section 1 Algorithm* (aka. When I look at one position, how to evaluate its probability of victory?). To avoid finding out all the chess shapes on the chessboard after filling this position with one color, the following *scoring mechanism* is designed.

Scoring Mechanism.

i. *General idea for scoring mechanism*: First call the function *generate_kids* to generate a list that contains all the empty positions need consideration. For each position in the list, evaluate its score based on both my side and enemy side. Find out the maximum score with corresponding position for my side and the maximum score with corresponding score for enemy side. Then decide the final new position, *aka. new_pos*, according to the two maximum values. As is shown below:

Algorithm 3 *part of* decide_new_pos


```
#伪代码如下:
Notes: After calculating the max scores of my side(max1) and enemy side(max2),
       and the corresponding position is position1 and position2, I do the
       following comparisons and then decide which position is for next step.
Input: max1, max2, position1, position2
Output: new_pos
Codes:
    if max2 is level_one, # means max2 = 100000, the enemy chess shape is "win5"
        then new_pos = position2
    else if max1 >= level_two1, # means max1 >= 10000, my chess shape will almost win
                                # for the last one step
        then new_pos = position1
    else if max2 >= level_two1, # means max2 >= 10000, enemy chess shape will almost win
                                # for the last one step
        then new_pos = position2
    else if max1 >= level_three2, # means max1 >= 4900, my chess shape will almost win
                                  # for the last two steps
        then new_pos = position1
     else if max2 >= level_three2, # means max2 >= 4900, enemy chess shape will almost win
                                  # for the last two steps
        then new_pos = position2
    else if max1 > max2,
        then new_pos = position1
    else if max2 > max1,
        then new_pos = position2
    else if max1 is equal to max2,
        if max2 >= level_eight, then new_pos = position2
        else, new_pos = position1
```

ii. *Scoring method*:

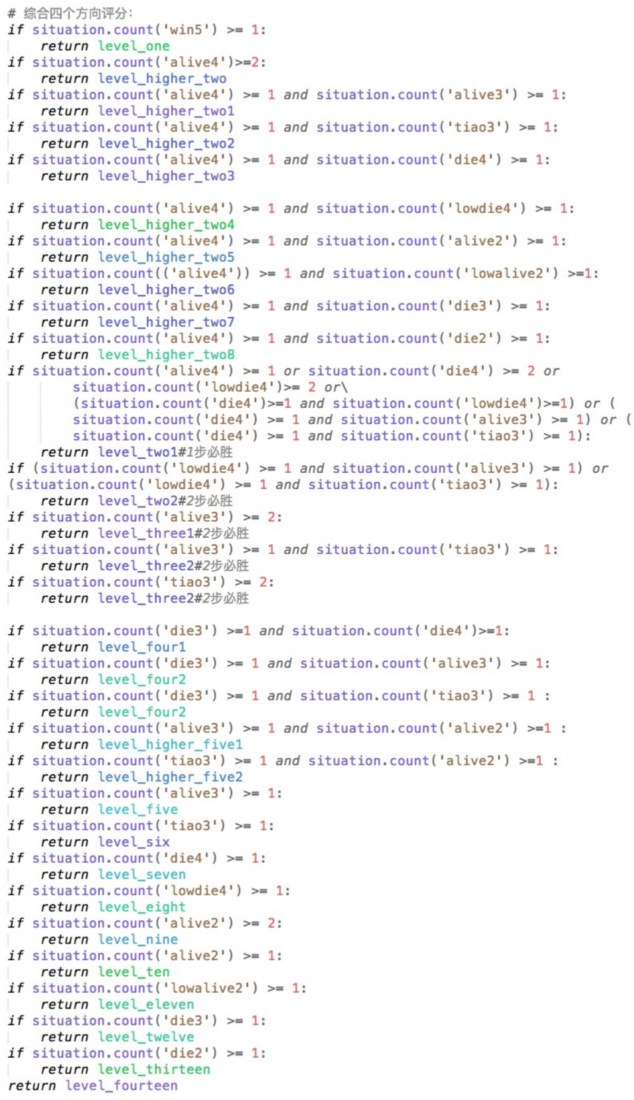
- Pre-position (i, j) with color *COLOR_BLACK* or *COLOR_WHITE*.
- Extract chess shapes in four directions centering at (i, j) within length 9: x-direction, y-direction, and two diagonal directions.
- Judge the chess shapes of four directions.
- Return a score of position (i, j) according to its four chess shapes.

iii. *Scoring rules*:

The scores for different levels can be found in *Section 1 Representation*.

Algorithm 4 *part of* evaluate

The following are chess shapes with corresponding scores:



```
# 综合四个方向评分:
if situation.count('win5') >= 1:
    return level_one
if situation.count('alive4')>=2:
    return level_higher_two
if situation.count('alive4') >= 1 and situation.count('alive3') >= 1:
    return level_higher_two1
if situation.count('alive4') >= 1 and situation.count('tiao3') >= 1:
    return level_higher_two2
if situation.count('alive4') >= 1 and situation.count('die4') >= 1:
    return level_higher_two3

if situation.count('alive4') >= 1 and situation.count('lowdie4') >= 1:
    return level_higher_two4
if situation.count('alive4') >= 1 and situation.count('alive2') >= 1:
    return level_higher_two5
if situation.count(('alive4')) >= 1 and situation.count('lowalive2') >=1:
    return level_higher_two6
if situation.count('alive4') >= 1 and situation.count('die3') >= 1:
    return level_higher_two7
if situation.count('alive4') >= 1 and situation.count('die2') >= 1:
    return level_higher_two8
if situation.count('alive4') >= 1 or situation.count('die4') >= 2 or
        situation.count('lowdie4')>= 2 or\
        (situation.count('die4')>=1 and situation.count('lowdie4')>=1) or (
        situation.count('die4') >= 1 and situation.count('alive3') >= 1) or (
        situation.count('die4') >= 1 and situation.count('tiao3') >= 1):
    return level_two1#1步必胜
if (situation.count('lowdie4') >= 1 and situation.count('alive3') >= 1) or
(situation.count('lowdie4') >= 1 and situation.count('tiao3') >= 1):
    return level_two2#2步必胜
if situation.count('alive3') >= 2:
    return level_three1#2步必胜
if situation.count('alive3') >= 1 and situation.count('tiao3') >= 1:
    return level_three2#2步必胜
if situation.count('tiao3') >= 2:
    return level_three2#2步必胜

if situation.count('die3') >=1 and situation.count('die4')>=1:
    return level_four1
if situation.count('die3') >= 1 and situation.count('alive3') >= 1:
    return level_four2
if situation.count('die3') >= 1 and situation.count('tiao3') >= 1 :
    return level_four2
if situation.count('alive3') >= 1 and situation.count('alive2') >=1 :
    return level_higher_five1
if situation.count('tiao3') >= 1 and situation.count('alive2') >=1 :
    return level_higher_five2
if situation.count('alive3') >= 1:
    return level_five
if situation.count('tiao3') >= 1:
    return level_six
if situation.count('die4') >= 1:
    return level_seven
if situation.count('lowdie4') >= 1:
    return level_eight
if situation.count('alive2') >= 2:
    return level_nine
if situation.count('alive2') >= 1:
    return level_ten
if situation.count('lowalive2') >= 1:
    return level_eleven
if situation.count('die3') >= 1:
    return level_twelve
if situation.count('die2') >= 1:
    return level_thirteen
return level_fourteen
```

To briefly illustrate, take alive-four and double-alive-three as an example. For one position (i_1, j_1), if its chess shapes include more than one alive-four and no better chess shapes, then its score will be at least *level_two1*, *aka. 10000*. For another position (i_2, j_2), if its chess shapes include two alive-three and no better chess shapes, then its score will be 5000, which is lower than position (i_1, j_1).

iv. *Method of judging chess shapes*:

Set position (i, j) as the center and count the number of successive chesses with the same color. And judge the chess shape of one direction based on this number.[2]

III. Empirical Verification

To varify the quality of my program, I conduct two kinds of experiments. And the detailed explanation is as follows.

A. Design

- **Design 1.** Using the provided code checker
  In the first phase of the project, there is a provided code checker including six test cases: *empty chessboard, only one empty position remain, win, defense 5 inline, two three, defense*. Every time I test one of the six cases and check under which case my program has bugs.
- **Design 2.** Download the chessboard data from the website during the online game
  When my AI agent fight against others' and fail, I download the chessboard data and analyse the reason why my program choose a bad position to go. And I will also compare my next step with others' next steps to improve my program.

B. Result

I implement the above two designs of empirical verification through printing the chessborad and every steps along with their scores. Through the analysis step by step, I can find many fatal flaws of my program and modify them. The result is that my program's performance when fighting with others becomes better. The results meet my expectation about the program.

Acknowledgment

I appreciate all the great efforts paid by our student assistants and teaching assistants. They have spared no pains to build up and maintain the excellent platform of Gomoku Online Game. Besides, they've made a pretty fair scoring rule for our programs. What's more, they are pacient enough to help us understand the algorithms and solve our problems. Thanks them for providing such a great learning opportunity of Artificial Intelligence and python programming.

References

[1] Gomoku —- Japanese Board Game http://www.japan-101.com/culture/gomoku_japanese_board_game.htm

[2] The Approach of Gobang AI https://www.cnblogs.com/songdechiu/p/5768999.html