Gomoku

Project 1 Report of

CS303 Artificial intelligence

Department of Computer Science and

Engineering BY

Hongxin Peng

11710716

1. Preliminaries

Problem Description

Gomoku is a relatively simple board game. Its instrument is universal with Go, and it originated from one of the ancient Chinese black and white chess. Usually, the two sides use black and white chess pieces respectively, and at the intersection of the straight line and the horizontal line of the board, the next step is to form a five-member line to win.

In this assignment, we use the default board of size 15*15 board (administrators can modify the settings as needed). Students need to implement the AI algorithm of Gomoku according to the interface requirements and submit it to the system as required for usability testing, points race, and round robin.

Problem Applications

The AI algorism can apply in APP in mobile phones, people can race against with AI if can't find other people. Kids can improve their intelligence, Older can prevent senile dementia. Everybody can use it to relax and comfort lonely.

The AI algorism is to find a optimal solution, not just in present, also prepare to next step. It can enhance big-picture thinking, foresight.

2. Methodology

Notations

Calc_score() to get the score on one direction

ScoreModel() to store all the chess model and score.

Evaluate() to separate four directions to get scores and add together.

Has_neighbor() to find whether there are the same color chesses around center chess.

Run() to decide the next position. from all empty position to get attack score and defend score. Compare score to get next position.

Rearrange() to arrange the blank list, make the nodes which around the last position we decided on the front of blank list.

Data Structure

ai_list = [] to store all the position which the color the same as.

people_list = [] to store he position which another color the same as.

blank_list = [] to store the position which is empty.

aipeo_list = [] to store all the position which is not empty.

new_pos = [] to store the decision position

shape6 = tuple() to get chess model with length Is 6.

max_attack = [0, 0] to get max attack score and its defend score

max_attack_po = () to get the position which has max attack score

max_defend = [0, 0] to get max defend score and its attack score

max_defend_po = () o get the position which has max defend score

max_score = 0 the final max score.

Model design

In order to finish this project, I used the score table to record the scores of the models of chess. And every time to search all the empty nodes on the chessboard then get two scores: attack score and defend score. Attack score: make my own color chess instead of empty chess, then make this point be center to search my color chess position in four directions and add together to get the attack score. Defend score: put another color chess on the same position then get the defend score in the table.

When we calculate the score on one direction, because there will be different scores of different models, we need to choose max score. Then add all the scores of four directions as the attack score or defend score.

Because of the advantage of black chess so when my chess color is black, I need to enhance the offensive, that is multiply 1.5 to the attack score. If my color is white, multiply 1.5 to the defend score. Then compare the max score between all the attack scores and defend scores to get best position. If there are two positions have the same attack score then compare the defend scores to choose the position with larger defend score.

Detail of Algorism

Scoremodel to store scores and chess models. Different model has different scores.

```
ScoreModel = [(1261, (0, 1, 1, 0, 0)),
                  (260, (1, 1, 0, 0, 0)),
                  (260, (0, 1, 0, 0, 1,0)),
                  (260, (0, 1, 0, 0, 1,0)),
                  (260, (0, 0, 0, 1, 1)),
                  (1261, (0, 0, 1, 1, 0)),
                  (260, (1, 0, 1, 0, 0)),
                  (260, (0, 0, 1, 0, 1)),
                  (1115, (1, 0, 0, 1, 1)),
                  (1150, (0, 1, 0, 1, 0)),
                  (1161, (0, 0, 1, 1, 0)),
                  (1300, (1, 1, 0, 1, 0)),
                  (1300, (0, 1, 0, 1, 1)),
                  (1310, (0, 0, 1, 1, 1)),
                  (1310, (1, 1, 1, 0, 0)),
                  (1320, (1, 0, 1, 1, 0)),
                  (1320, (0, 1, 1, 0, 1)),
                  (5170, (0, 1, 1, 1, 0)),
                  (5002, (0, 1, 0, 1, 1, 0)),
                  (5002, (0, 1, 1, 0, 1, 0)),
                  (5000, (1, 1, 1, 0, 1)),
                  (5000, (1, 1, 0, 1, 1)),
                  (5000, (1, 0, 1, 1, 1)),
                  (6000, (1, 1, 1, 1, 0)),
                  (6000, (0, 1, 1, 1, 1)),
                  (50000, (0, 1, 1, 1, 1, 0)),
                  (99999999, (1, 1, 1, 1, 1))]
```

```
calc_score(self, x, y, x_direction, y_direction, list_1, list_2,
score_all,is_defend):
    max_score =0
    for I in range(-5,1):
        get shape6 from start chess #the length of shape6 is 6
        if it is defending then
            if shape6== special model not in ScoreModel:
                max score= score
        for model,score in ScoreModel:
            If shape6==model and score >= max_score
                max_score=score
    return max score
```

```
rearrange(self) :
    for last position in aipeo_list
        for position around the last position
            if position.color==0 :
                put it on front of blank list
```

```
has_neighbor(x,y):
    for chess(i,j) around chess(x,y):
        if  chess(i,j).color == chess(x,y).color:
                return true
    return false
```

```
run():
       rearrange()
```

```
        max_attack = [0, 0]
        max_attack_po = ()
        max_defend = [0, 0]
        max_defend_po = ()
        max_score = 0
         if self.color is black:
             r1=1.5
             r2=1
         else:
             r1=1
             r2=1.5
         for nextstep in blank_list:
             if not has_neighbor:
                 continue
             ai_list.append(nextstep)
             attack_score = evaluate ()
             ai_list.remove(nextstep)
             people_list.append(nextstep)
             defend_score = evaluate ()
             people_list.remove(nextstep)
             if attack_score >max attack_score
                 max_attack_score =attack_score
                 max_attack_po = nextstep
             if defend_score > max defend_score
                 max_defend_score = defend_score
                 max_defend_po = nextstep
         if max_attac_score >max_defend_score:
             new_position =max_attack_po
         else
             new_position = max_defend_po
```

```
evaluate (self, x, y, list_1, list_2, is_defend) to separate four directions
    total_score += score in direction (0,1)
    total_score += score in direction (1,0)
    total_score += score in direction (1,1)
    total_score += score in direction (-1,1)
    return total_score
```

```
go(self, chessboard):
        # Clear candidate_list
        self.candidate_list.clear()
        # ==================================================================
        # Write your algorithm here
        # Here is the simplest sample:Random decision
        self.ai_list = all chess.color is self.color
        self.people_list = all chess.color is -self.color
        self.blank_list = all chess.color is 0
        self.aipeo_list = all chess.color is not 0
        new_pos = []
        if len(self.aipeo_list) == 0: # first step in empty chessboard
            self.candidate_list.append(chessboard[7,7])
            return
        new_pos = self.run()  # get the best position
        # ==============Find new pos========================================
        # Make sure that the position of your decision in chess board is empty.
        # If not, return error.
        if len(new_pos) == 0: # no way to make 5 chesses connected
            new_pos = random position in blank_list
        assert chessboard[new_pos[0], new_pos[1]] == COLOR_NONE
        # Add your decision into candidate_list, Records the chess board
        self.candidate_list.append(new_pos)
```

3. Empirical Verification

Dataset

I used the check file in sakai and get some chessboards when I fight with other AI to find better decision and some chessmodel in internet:

```
        chessboard = np.zeros((self.chessboard_size, self.chessboard_size),
dtype=np.int)
        chessboard[7, 9] = -1
        chessboard[8,8] = -1
        chessboard[7,7] = 1
        chessboard[7,9] = 1
        if not self.__check_result(chessboard, [[9, 10]]):
            self.errorcase = 2
            return False

        chessboard = np.zeros((self.chessboard_size, self.chessboard_size),
dtype=np.int)
        chessboard[1,0]=1
        chessboard[12,3]=-1
        chessboard[1,1]=1
        chessboard[10,13]=-1
        chessboard[1,2]=1
        chessboard[10,1]=-1
        chessboard[1,5]=1
        chessboard[9,12]=-1
        chessboard[1,7]=1
        chessboard[9,6]=-1
        chessboard[1,13]=1
        chessboard[7,13]=-1
        chessboard[1,14]=1
        chessboard[7,12]=-1
        chessboard[3,1]=1
        chessboard[7,1]=-1
        if not self.__check_result(chessboard, [[7, 10]]):
            self.errorcase = 2
            return False
        chessboard = np.zeros((self.chessboard_size, self.chessboard_size),
dtype=np.int)
        chessboard[2, 2] = 1
        chessboard[2, 4] = 1
        chessboard[3, 2:4] = 1
        chessboard[5, 2] = 1
        chessboard[1, 10:12] = -1
        chessboard[2, 10] = -1
        chessboard[4, 12:14] = -1
        if not self.__check_result(chessboard, [[4, 2]]):
            self.errorcase = 3
            return False
```

```
chessboard = np.zeros((self.chessboard_size, self.chessboard_size),
dtype=np.int)
        chessboard[12,7]=1
        chessboard[2,11]=-1
        chessboard[12,9]=1
        chessboard[5,1]=-1
        chessboard[4,2]=1
        chessboard[2,3]=-1
        chessboard[1,4]=1
        chessboard[2,5]=-1
        chessboard[3,5]=1
        chessboard[2,4]=-1
        chessboard[2,6]=1
        chessboard[1,6]=-1
        chessboard[1,9]=1
        chessboard[3,7]=-1
        chessboard[7,11]=1
        chessboard[1,3]=-1
        chessboard[10,2]=1
        chessboard[12,2]=-1
        chessboard[12,1]=-1
        chessboard[13,11]=1
        chessboard[2,7]=-1
        if not self.__check_result(chessboard, [[2, 2]]):
            self.errorcase = 3
            return False

        chessboard = np.zeros((self.chessboard_size, self.chessboard_size),
dtype=np.int)
        chessboard[7,6]=-1
        chessboard[6,6]=1
        chessboard[8,4]=-1
        chessboard[8,5]=1
        chessboard[6,5]=-1
        chessboard[5,4]=1
        chessboard[11,7]=-1
        chessboard[9,9]=1
        chessboard[11,10]=-1
        chessboard[11,8]=1
        chessboard[10,7]=-1
        chessboard[13,7]=1
        chessboard[11,11]=-1
        chessboard[11,12]=1
        chessboard[9,10]=-1
        chessboard[12,11]=-1
        chessboard[10,11]=1
        chessboard[13,12]=-1
        chessboard[14,13]=1
        chessboard[12,6]=-1
        chessboard[13,5]=1
        chessboard[13,6]=-1
        chessboard[7,9]=1
        chessboard[7,2]=1
        chessboard[4,5]=-1
        chessboard[4,3]=1
        chessboard[10,12]=-1
        chessboard[9,11]=1
        chessboard[6,11]=-1
        chessboard[3,4]=1
        chessboard[5,2]=-1
        if not self.__check_result(chessboard, [[8, 1]]):
            self.errorcase = 3
            return Fals
```

Performance Measure

Time complexity :

Firstly, traverse every node to classify every node to ai_list, people_list, blank_list. -- 15×15

Secondly, traverse blank_list to calculate scores. In this part, x = blank_list.size, every p in blank_list to calculate two scores, but I discard nodes that no chesses around(not has_neighbor), four directions, six patterns, matching patterns in each direction. --2×4×6×27×*p (The worse situation is every blank node has_neighbor. This time, p=15*×15-3×3)

The worse complexity 2×4×6×27×(15×15-3×3)

Hyperparameters

In different color I will multiply 1.5 to attack score or defend score black chess should enhance the offset and white chess should enhance defend. To change different scores of different chess models, such as

(5170, (0, 1, 1, 1, 0)), ----[1]

(5002, (0, 1, 0, 1, 1, 0)), -----[2]

It is about priority problem in chessboard. [1].score > [2].score, because after another defend, [1] will be (0, 1, 1, 1, -1) we can make (0,1, 1, 1, 1, -1) to continue attack. But [2] will be (0, 1, -1, 1, 1, 0), and then (1, 1, -1, 1, 1, 0) and (0, 1, -1, 1, 1, 1) are no threat. So there are:

(1261, (0, 1, 1, 0, 0))	(5002, (0, 1, 1, 0, 1, 0))
(260, (0, 1, 0, 0, 1,0))	(5170, (0, 1, 1, 1, 0))
(1150, (0, 1, 0, 1, 0))	

The closer the chess pieces, the higher score. And some special situations in defense out of ScoreModel, so I listed them in the calc_score. Detail in Conclusion.

I used different ScoreModels and parameters in Points Race and Round robin. Original edition:

```
    scoreModel = [(160, (0, 1, 1, 0, 0)),
                  (160, (1, 0, 1, 0, 0)),
                  (110, (1, 0, 0, 1, 1)),
                  (110, (0, 1, 0, 1, 0)),
                  (160, (0, 0, 1, 1, 0)),
                  (200, (1, 1, 0, 1, 0)),
                  (300, (0, 0, 1, 1, 1)),
                  (300, (1, 1, 1, 0, 0)),
                  (300, (1, 0, 1, 1, 0)),
                  (300, (0, 1, 1, 0, 1)),
                  (5170, (0, 1, 1, 1, 0)),
                  (5002, (0, 1, 0, 1, 1, 0)),
                  (5002, (0, 1, 1, 0, 1, 0)),
                  (5000, (1, 1, 1, 0, 1)),
                  (5000, (1, 1, 0, 1, 1)),
                  (5000, (1, 0, 1, 1, 1)),
                  (5171, (1, 1, 1, 1, 0)),
                  (5171, (0, 1, 1, 1, 1)),
                  (50000, (0, 1, 1, 1, 1, 0)),
                  (99999999, (1, 1, 1, 1, 1))]
```

It think less about detail. When I race with other students, observe every step to modify the ScaoreModel.

Experimental results

Usability test cases: pass all.

Usability text cases score: 100

Points Race rank: 68

Points Race score: 86

Round robin score: 82

Project1ProgramScore: 90

Conclusion

Disadvantage:

- I just search depth ==0. That is narrow.
- My score model is not best score model.
- I don’t use chess manual

Advantage:

- use less time
- My score model separate many situations and with different scores
- I will enhance attack or defense in different situations

- Some position will make different scores between attack and defense, I list them out. Such as

  When it is defense (-*1* is new position)

  (0, 1, 1, 1, -*1*, -1)and (0,-*1*, 1, 1, 1, 0, -1) should has almost scores to avoid (0, 1, 1, 1, 1, 0)

  (-*1*, 1, 1, 0, 1, 0) and (0, 1, 1, -*1*, 1, 0) should has almost scores to avoid (0, 1, 1, 1, 1, 0)

  (0, 1, 0, 1, 1, -*1*) and (0, 1, 1, -*1*, 1, 0) should has almost scores to avoid (0, 1, 1, 1, 1, 0)

  (0, 1, 1, 0, 1, -*1*) and (0, 1, 1, -*1*, 1, 0) should has almost scores to avoid (0, 1, 1, 1, 1, 0)

  (0, -*1*, 1, 0, 1, 1) and (0, 1, 1, -*1*, 1, 0) should has almost scores to avoid (0, 1, 1, 1, 1, 0)

  (-*1*, 1, 0, 1, 0) and (0, 1, -*1*, 1, 0) should has almost scores to avoid (0, 1, 1, 1, 0)

  (0, 1, 0, 1, -*1*) and (0, 1, -*1*, 1, 0) should has almost scores to avoid (0, 1, 1, 1, 0)

```
            if is_defend:
                if shape6 == (-1, 1, 1, 1, 1, 0) and i == -1 and (
                        x + 5 * x_direction, y + 5 * y_direction) in
self.blank_list:
                    if 50050 > max_score:
                        max_score = 50050
                elif shape6 == (0, 1, 1, 1, 1, -1) and i == -4 and (
                        x - 5 * x_direction, y - 5 * y_direction) in
self.blank_list:
                    if 50050 > max_score:
                        max_score = 50050
                elif shape6 == (-1, 1, 1, 1, 0, 0) and i == -1:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (0, 0, 1, 1, 1, -1) and i == -4:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (0, 1, 0, 1, 1, 0) and i == -4:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (0, 1, 1, 0, 1, 0) and i == -1:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (0, 1, 1, 0, 1, 1) and i == -1 and (
                        x + 5 * x_direction, y + 5 * y_direction) in
self.blank_list:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (1, 1, 0, 1, 1, 0) and i == -4 and (
                        x - 5 * x_direction, y - 5 * y_direction) in
self.blank_list:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (1, 0, 1, 1, 1, 0) and i == -4 and (
                        x - 5 * x_direction, y - 5 * y_direction) in
self.blank_list:
                    if 5170 > max_score:
                        max_score = 5170
                elif shape6 == (0, 1, 1, 1, 0, 1) and i == -1 and (
                        x + 5 * x_direction, y + 5 * y_direction) in
self.blank_list:
                    if 5170 > max_score:
                     max_score =5170
```

- Not just compare one score, if two positions have same attack score (defend score) them compare defend score (attack score)

4. Reference

[1] H.J. van den Herik, “Gomoku” in Searching for Solutions in Games and Artificial Intelligence. Maastricht, Netherlands, 1994, ch.5, pp 121-152.

[2] W.S.laoqiu(2016) 五子棋AI的思路[online]. Available: https://www.cnblogs.com/songdechiu/p/5768999.html