Gomoku Report

Yifei Xu 11611209
Southern University of Science and Technology

1. Preliminaries

My project is to implement a simple program on gomoku game. Compared to other chessboard games, gomoku's rule is easy to learn and implement. In my project, the ai simply follows basic rules and finds a relatively good position to place a chessman. As following are the details of how to implement this simple ai.

1.1. Software

This project is based on Python and the IDE is Pycharm. As for he libraries, Numpy is included.

1.2. Algorithm

1. Min-Max Algorithm [1]
2. Alpha-Beta Pruning [2]
3. Depth-first Search [3]

2. Methodology

This part is to introduce the details of algorithm and the architecture of my program. The idea of implement such AI is that firstly, assume the black player places a chessman on some place ***(n, m)***, and then find all chess types. Secondly, withdrew that chessman. Thirdly, assume the white player place a chessman on the same place and then find all chess types. Finally, withdrew that chessman. Above these operations, we get all types for black player and whit player.

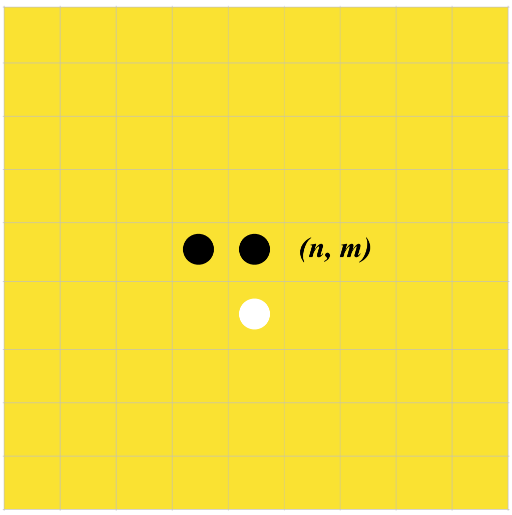


Fig 2.1

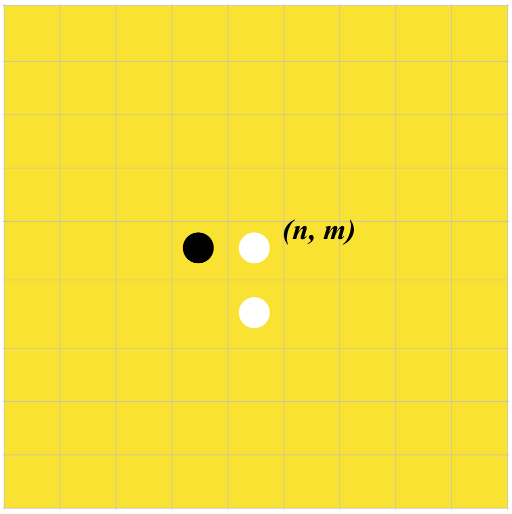
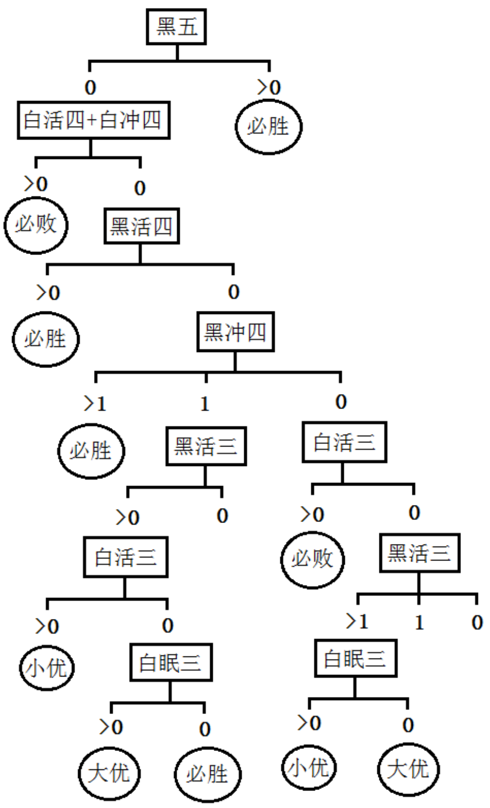


Fig 2.2

e.g. Fig 2.1 and Fig 2.2 show the process of how to do such operations, and in Fig 2.1, there is a live-2 for black player. In Fig 2.2, there is a live-2 for white player. So the total score is black live-2's score + white live-2's score.

Logic Tree (for black player)



2.1. Representation

There are several global constant:

Global configuration:

- COLOR_BLACK : black chessman
- COLOR_WHITE : white chessman
- random.seed(0) : to create a random num
- COLOR_NONE : no chessman
- max_deep : max depth for DFS (alpha_beta)

The scores for each chess type:

- SCORE : an array with one row and ten columns

Index of type in SCORE :
e.g. As_color-5 is "five in a row

- idx_5 : Index of as_color-5 in SCORE
- idx_live4 : Index of live-4 in SCORE
- idx_form4 : Index of as_color-4 in SCORE
- idx_live3 : Index of live-3 in SCORE
- idx_sleep3 : Index of sleep-3 in SCORE
- idx_live2 : Index of live-2 in SCORE
- idx_sleep2 : Index of sleep-2 in SCORE
- idx_live1 : Index of live-1 in SCORE
- idx_sleep1 : Index of sleep-1 in SCORE

To accelerate the process of scanning the board :

- forms : An array with two dictionaries. For each dictionary, it stores all types in a five-length or six-length line and the corresponding index of this type. e.g.: '011110' is a live-4 chess type in white and the corresponding index of type is "idx_live4"

2.2. Architecture

Generally, there are five functions in class "AI" :

①	***__init__*** :	Construct function
②	***go***	Find a new position to place
③	***alpha_beta***	Find the best solution
④	***evaluate***	Evaluate the status of the chessboard
⑤	***calculate***	Scan the chessboard and get all chess types in an "Union Jack"

Some private properties of class AI are as following.

2.3. Details of Algorithms

Here are the details of algorithms.

- ***go :*** This algorithm is to change all '-1'(*COLOR_BLACK*) in this chessboard to '2' in order to make it easier to scan chessboard and find an index where the next chessman should be placed.

Input: *chessboard*
Output: None

---

idx ← empty points on the chessboard
if *self.color = COLOR_BLACK* ∩ empty **then**
  new_pos ← the middle point of chessboard
else if one chessman **then**
  new_pos ← that chessman's neighbor
else
  new_pos ← a random element idx
  self.candidate_list ← new_pose
  new_pos ← alpha_beta(*chessboard, max_deep, idx, self.hum_color, infinite, -infinite,* **None**)[0]
self.candidate_list.append(new_pos)

- ***alpha_beta :*** This algorithm is to find the place which is as good as possible by using min-max algorithm and alpha-beta pruning.

Input: *chessboard, deep, idx, pre_color, alpha, beta, pre_step*
Output: *[value, new_pos]*, which contains a new position and its value

---

if deep = 0 **then**
  return *[evaluate(chessboard, pre_step, pre_color)]*
if *pre_color = hum_color* **then**
  for *i* **from** 0 **to** len*(idx)*
    n, m ← idx[i]
    *idx.*pop*(i)*
    chessboard[(n, m)] ← self_color
    temp ← alpha_beta*(chessboard, deep-1, idx, self_color, alpha, beta, (n, m))*
    chessboard[(n, m)] ← COLOR_NONE
    *idx.*insert*(i, (n,m))*
    if *temp[0] > alpha* **then**

```
        alpha ← temp[0]
        new_pos ← (n, m)
      if alpha >= beta then
        break
    return [alpha, new_pos]
else if pre_color = self_color:
    for i from 0 to len(idx)
      n, m ← idx[i]
      idx.pop(i)
      chessboard[(n, m)] ← self.hum_color
      temp ← self.alpha_beta(chessboard, deep - 1,
idx, hum_color, alpha, beta, (n, m))
      chessboard[(n,m)] ← COLOR_NONE
      idx.insert(i, v)
      if temp[0] < beta then
        beta ← temp[0]
        new_pos ← (n, m)
      if alpha >= beta then
        break
    return [beta, new_pos]
```

- ***evaluate :*** This algorithm is to evaluate the value of the specific position in a shape of "Union Jack" and return this value. And then sum values of two cases: 1. this chessman is in *self_color*. 2. this chessman is in *hum_color*. According to the **Logic Tree,** calculate the scores of this status.

Input: *chessboard, pre_step, pre_color*
Output: *score*

```
next_color ← the color of next chess player.
num ← empty array with 2 rows and 9 columns
num ← calculate(pre_color, pre_step, chessboard,
num)
chessboard[pre_step] ← next_color
num ← calculate(next_color, pre_step, chessboard,
num)
chessboard[pre_step] ← COLOR_NONE
// Calculate the scores of pre_color and next_color
pre_score ← sum(num[pre_color] * SCORE)
other_score ← sum(num[next_color] * SCORE)
score ← pre_score + other_score
if num[pre_color][idx_5] != 0 then
  score += float('inf')
else if num[next_color][idx_5] != 0 then
  score += 1000000000
else if num[pre_color][idx_live4] != 0 then
  score += 10000000
else if num[next_color][idx_live4] != 0 then
  score += 5000000
else if num[pre_color][idx_form4] > 1 then
  score += 4000000
else if num[next_color][idx_form4] > 1 then
  score += 3000000
else if num[pre_color][idx_live3] != 0 ∩
num[pre_color][idx_form4] != 0 then
  score += 2900000
else if num[next_color][idx_live3] != 0 ∩
num[next_color][idx_form4] != 0 then
  score += 2800000
else if num[pre_color][idx_live3] > 1 then
  score += 2000000
else if num[next_color][idx_live3] > 1 then
  score += 1500000
if pre_color = self_color then
  return score
else
  return -score
```

- ***calculate :*** This algorithm is to find all chess types of a specific point in a shape of 9-point long "Union Jack" and return these types' information

Input: *color, pre_step, chessboard, num*
Output: *num*

```
for every line crossing the center point pre_step
  priority ← infinite
  for count from 0 to len(idx - 5):
    line_bries ← 6-bit-long brie of line from count
    if forms[as_color] contains line_bries then
      t_priority ← forms[as_color][line_bries]
      if t_priority < priority then
        priority ← t_priority
  if priority != infinite:
    num[as_color][priority] += 1
return num
```

3. Empirical Verification

We can use the given file ***code_checker.py*** and compare the result to check the usability of this project.

3.1. Design

The experiment is to generate some special cases to test the quality of the project. As for the test file, it contains as many as possible cases that maybe happen in a real competition. Beside this, I write a simple GUI to represent the competition between each version of my project or my friend's project and mine.]

3.2. Data and Data Structure

The data I used to test is from sakai and my actual experience.

The data structures I used in my project include array, dictionary, tuple, list and so on.

3.3. Performance

The time complexity of the whole project is $O(n^{2d})$, where n is the size of chessboard and d is the max depth (e.g. if there is a15x15 chessboard, then n is 15). As for each method, the time complexity of ***go*** is $O(n^{2d})$. The time complexity of ***alpha_beta*** is $O(n^{2d})$. The time complexity of ***evaluate*** is O(1). The time complexity of ***calculate*** is O(1).

The performance in real competition is satisfied. If max depth is 1, it will cost about 0.02 sec to get a result. Actually, in the final version of this project, I define *max_deep* to 1.

3.4. Result

Although the *max_deep* is 1, its performance in real battle, which is about 8th and 17th in two rounds, is not too bad. The reason why I do not use deeper searching is that the actual quality is weaker.

3.5. Analysis

The silver bullet of my project is the process of scanning the chessboard. For each point, I just scan a shape of "Union Jack" and its length is 9. And for each direction, I use a 6 units long window(e.g. red rectangular in Fig 3.5.1) and get all bits ('000021')in this window. Use this combination of bits as a key to find the corresponding *idx_xxxx* in *form* (global dictionary). This dramatically increases the speed of scanning.

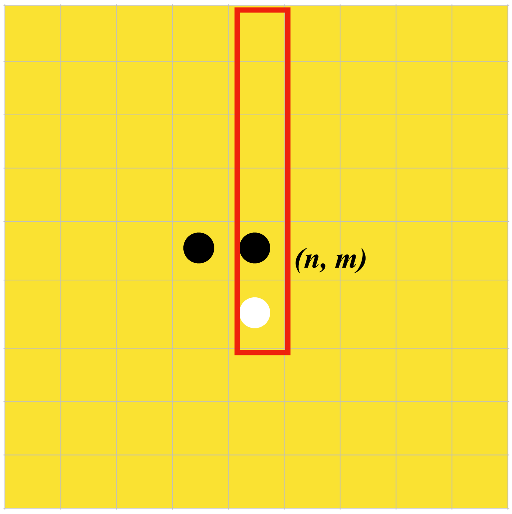


Fig 3.5.1

Acknowledgement

A big thank you to teaching assistance Yao Zhao who inspired and taught me and other student assistances of this course who spent their spare time developing and maintaining **AI vs Platform.**

References

[1] Wikipedia contributors, [Online]. Available: https://en.wikipedia.org/wiki/Minimax

[2] Wikipedia contributors, [Online]. Available: https://en.wikipedia.org/wiki/Alpha–beta_pruning

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.