CS303 Artifical Intelligence

Gomoku Project

Hu yubin, 11712121[1]

[1]Department of Computer Science and Engineering, Southern University of Science and Technology

October 9, 2019

Abstract

This project is implementing a Gomuku AI that can play on a specified platform with other AI.

Keywords
Gomoku, AI, Search tree

1 Preliminaries

Gomoku, also called Five in a Row, is an abstract strategy board game. It is traditionally played with Go pieces (black and white stones) on a chessboard. The chessboard is initially in size of 15×15[1]. This project need us 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. The use of memory cannot go beyond 100M, the time to find a place to drop cannot be longer than 5s, the whole battle cannot take longer than 180s.

1.1 Software

The code of Gomoku is written in Python. Version is Python 3.7.4. And only *numpy* package is extra imported.

For analytics section, I use *math*, *time* and *random* packages as tools to test the strength of AI.

1.2 Algorithm

The main algorithm is minimax seach tree, depth-first-search, α-β pruning algorithms, heuristic strategy for calculate the scores and Aho-Corasick automata[2] in some version.

2 Methodology

The basic implementation is minimax tree, all the optimization is based on it. For the correction, I design the evaluation function and some special judge to find the better way to go the pieces. For time and space limit, I do α-β pruning algorithms, heuristic strategy, global and local search[3] and some other optimizations.

2.1 Representation

- *Chess*
  - COLOR_BLACK = -1
    This is a constant. "-1" represents blank chess
  - COLOR_White = 1
    This is a constant. "1" represents white chess
  - COLOR_NONE = 0
    This is a constant. "0" represents that there is no chess
- *shape*

$$\begin{aligned}
=\{&'h5' : 20000,\ Win\\
&'h4' : 2000,\ \#Alive4\\
&'t4' : 450,\ \#Jump4\\
&'h3' : 450,\ \#Alive3\\
&'c4' : 300,\ \#Rush4\\
&'t3' : 300,\ \#Jump3\\
&'h2' : 100,\ \#Alive2\\
&'c3' : 50,\ \#Rush3\\
&'c2' : 20\}\ \#Rush2
\end{aligned}$$

  This is a dictionary. It represents different chess pattern and their scores. The higher, the better.

- *totleTime* This is a double parameter. It records the time that the AI used.

- *candidate_ list* This is a list. It records the steps that AI will go. We choose the last step as AI's best step.
- *Depth* This is a integer parameter. It is the depth of the minimax search tree. It may be changed because of the time limit.
- *sortedNum* This is a integer parameter. It is the number of the steps in each depth of the minimax search tree. I will evaluate every the single point from both players and sort them by the calculated score.
- *empty* This is a list. It is the empty chessboard.
- *myScore* This is a integer parameter. It records the current score of AI.
- *enemyScore* This is a integer parameter. It records the current score of AI's enemyScore.
- *totalScore* This is a integer parameter. It records the current score of both players.
- *level* This is a 15 × 15 list. I think the pieces in the middle is better in the same case. So that it is added to the score with small impact.

2.2 Architecture

There are several different versions of AI, some of them use different implementations. So that I will show them separately.

- *AI*
  - *init* Initialize some parameters, such as timeout, chessboard status, minimax search tree's depth, candidate-list, two players' status and scores and score table. Especially, I designed AI for black and AI for white, They has the same modes of thought. But for timeout, and search ability, I give them different parameters to achieve better performance.
  - *go* judge the first step for blank and white. If AI is blank, it will choose (7, 7) directly. If AI is white, it will choose on of (6, 6), (6, 8), (8, 8), (8, 6) randomly. Otherwith, *go* will put all the candidates steps searched by *minimax* to the candidate list for the given chessboard. Especially, for test case, I judeg whether the given chessboard is a test case or not by count the number of chess. If the different between the history and the given chessboard is over one, I guess it is a test case and do the optimization on it.
  - *minimax* The main function in this project, it is a minimax search tre to find the steps that AI will go. It divides into black part and white part.
  - *count_ chess_ number* Count the number of chesses in the current chessboard.
  - *evaluation_point* Evaluate the single point's score in the current chessboard.
  - *evaluation_ global* Evaluate the global score in the current chessboard. It depends on all the single point's score.
  - *evaluation_ line* Evaluate the score of a specified length line.
  - *next_ step* Generate the top 5 steps for AI to run the minimax search tree.
  - *update_ score* Update *myScore*, *enemyScore* and *totalScore* after a step is choosed in the minimax tree.

In another version, I use Aho-Corasick automata to find the pattern in the chessboard.

- *Node*
  - *init* Initialize next, fail, isword and word.
- *automata*
  - *add* Add the pattern into the automata.
  - *make_fail* Build the automata by the patterns that I added before.
  - *search* Search which pattern appear the in a specified line.

2.3 Mode design

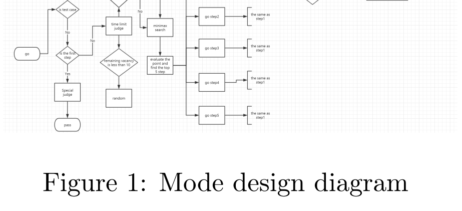


Figure 1: Mode design diagram

The flow chart of Gomoku project is shown in *Figure 1*.

2.4 Detail of Algorithm

2.4.1 minimax search tree

When dealing with gains, it is referred to as "maximin"—to maximize the minimum gain. Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision-making in the presence of uncertainty.

For Gomoku, we set the depth of minimax search tree and the number of node in each depth. Due to the time limitation, I set the depth to 5 and the number of node to 5 when AI is blank. And I set the depth to 4 and the number of node to 7 when AI is White.

Algorithm 1 minimax seqarch tree

```
1:  function MINIMAX(node, depth, maxAI)
2:      if depth == 0 or point is a terminal point then return evaluation of point
3:      end if
4:      if maxAI then
5:          maxEva = -inf
6:          for each candidate do
7:              eva = minimax(candidate, depth-1, false)
8:          end for
9:          maxEva = max(maxEva, eva)
10:         return maxEva
11:     else
12:         minEva = inf
13:         for each candidate do
14:             eva = minimax(candidate, depth-1, true)
15:         end for
16:         minEva = min(minEva, eva)
17:         return minEva
18:     end if
19: end function
```

2.4.2 α-β pruning algorithms

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move.

Algorithm 2 *alpha-beta* pruning

```
1:  if maxAI then
2:      if s thencore > alpha
3:          alpha = score
4:      end if
5:      if a thenlpha >= beta:
6:          return alpha
7:      end if
8:      return alpha
9:  else
10:     if s thencore < beta:
11:         beta = score
12:     end if
13:     if a thenlpha >= beta:
14:         return beta
15:     end if
16:     return beta
17: end if
```

2.4.3 evaluation_line

I find that a chess only affect at most 32 chesses in the chessboard.

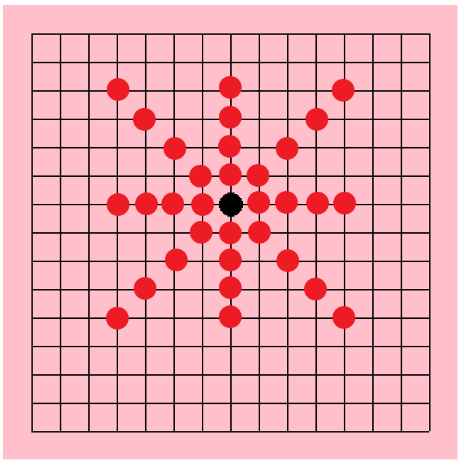


Figure 2: 32 affected chesses

2.4.4 Heuristic Search

I evaluate the point in two ways. One is the score that the AI color chess is in the point, the other is the opponent color chess in the point. The sum of them is the final score of the point. I consider both of them because the point that benefits opponent also should be taken into account.

Due to the time limitation, I only choose 5 candidates in each depth.

```
for i in range(self.chessboard_size):
    for j in range(self.chessboard_size)
    :
        if self.chessboard[i][j] ==
            COLOR_NONE:
                sortedlist.append((self.
                    totalScore[i][j] + self.
                    level[i][j], i, j))
sortedlist.sort(reverse=True)
```

2.4.5 Aho-Corasick automata

It is easy for us to find all the pattern in the string. Although it will count more pattern, but this mistake affect a little that I can accept it.

The key to automata is to build a Trie(dirctionary tree). Trie's build process is like this, when to insert a lot of words, we need to traverse the entire string back once upon a time, when we found out that the current has been built to insert the character of the node to previously, we directly to consider next character can, when we found that the current is no longer the character to be inserted before a character does not have its own node under the tree, which is formed by the we're going to create a new node to represent the characters, took down the traversal of other characters.Then repeat.

For the example, I have "she", "he", "say", "her", "shr" this five strings. The Trie is:

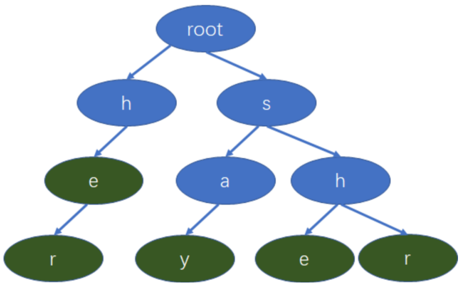


Figure 3: Trie

And the function *make_fail* do this thing:

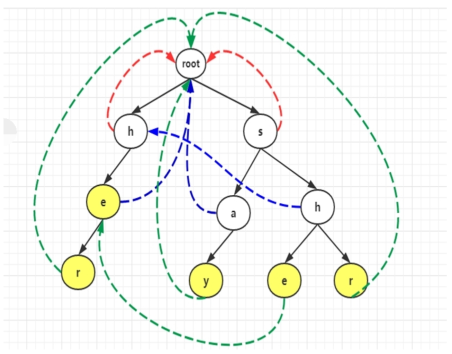


Figure 4: make$_fail$

3 Performance Analytics

3.1 Dataset

In the first stage, I cannot pass the pretest for my first submission. So that I creat some given chessboard for my AI and step by step debugging to find bugs. For convenience, I wrote a visual program that could display a chess board and I could play chess on it to simulate the AI against me, and at the same time, I could also watch the AI data change. Here are some dataset and the chessboard I built:

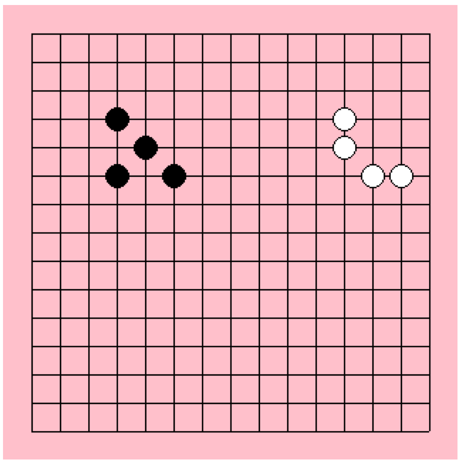


Figure 5: blank first, blank win(Alive3)

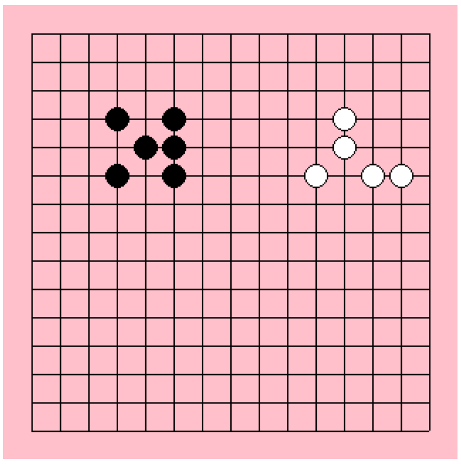
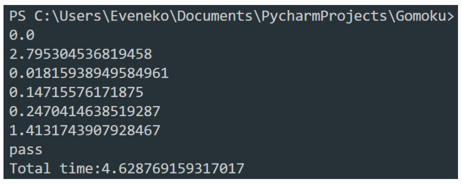


Figure 6: white first, white win(Alive3 + Jump3)

From this data, I pass all the pretest and find how adjust the parameter of pattern's score.

3.2 Performance measure

To keep track of time, I added some time keeping code to my AI code. After each run of the dataset I built myself, I let the program to output the total test time, the time spent on each test sample:



```
PS C:\Users\Eveneko\Documents\PycharmProjects\Gomoku>
0.0
2.795304536819458
0.01815938949584961
0.14715576171875
0.2470414638519287
1.4131743907928467
pass
Total time:4.628769159317017
```

Figure 7: Running time

Due to this way, I get the average time for the dataset, but it is not the worth case. The worth case is the AI should run the while minimax search tree with no α-β pruning. But I think my AI could not meet this case because it select top 5 steps in each depth and sort them. It is benefit for the α-β pruning so that the running time is much less than the worth case.

3.3 Hyperparameters

- For the first stage, pass 25 pretest is my goal. So I set a basic score table for each pattern. It just follows Five $>$ Alive4 $>$ Alive3 $>$ Ruch4 $>$ Jump4 $>$ jump3 $>$ Alive2 $>$ Rush3 $>$ Rush2
- For the second stage, That score table is too week to fight with other AI. So that I refined each pattern's score and their combination. For example, Alive3 + Jump3 = ALive4. And I find some special case:

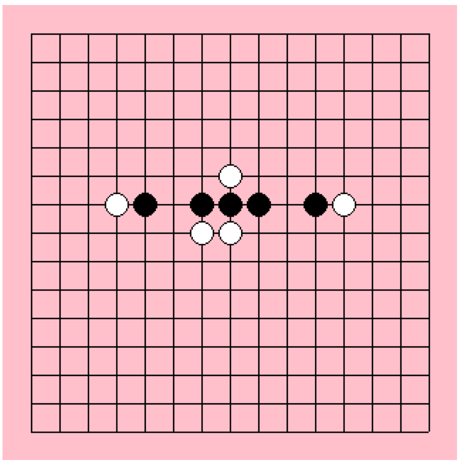


Figure 8: Running time

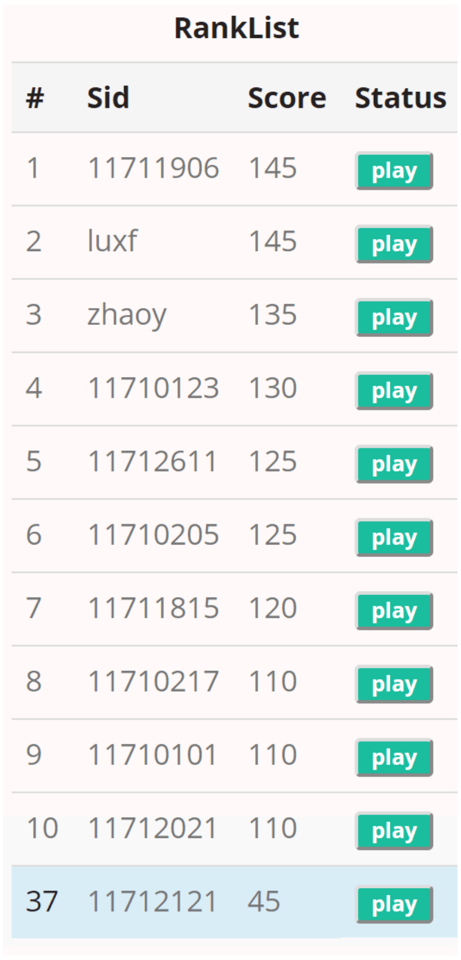
For my evaluation function, it just thinks black only have a jump4 and it is not a winning game. In fact, it is a winning game for black.

When I fix those bugs, my rank is up. When the rank is top 20, it is not easy to fight against all the AI, so I play will the AI by hand or "playto". Then play chess on their weaknesses.

- For the third stage, I think the server runs the AI of many classmates at the same time.The computing power of the server will decrease significantly.Therefore, I adjusted the time limit of my code to make it run a little faster, and when the time was running out, I reduced the accuracy and greatly improved the speed, so as to avoid the failure caused by overtime.

3.4 Experimental results

Before I did that:

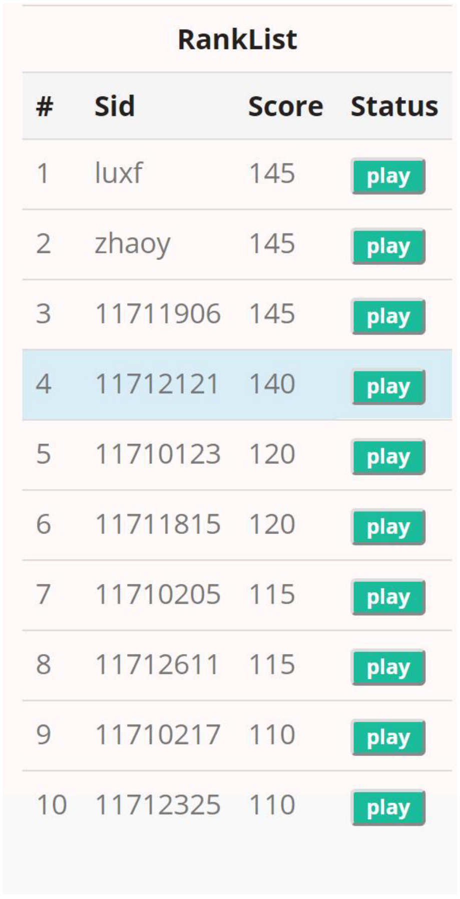


RankList

#	Sid	Score	Status
1	11711906	145	play
2	luxf	145	play
3	zhaoy	135	play
4	11710123	130	play
5	11712611	125	play
6	11710205	125	play
7	11711815	120	play
8	11710217	110	play
9	11710101	110	play
10	11712021	110	play
37	11712121	45	play

Figure 9: Rank37

After I did that:

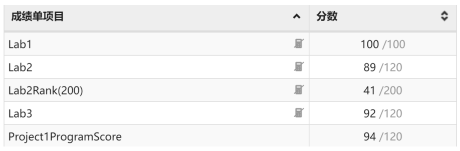


RankList

#	Sid	Score	Status
1	luxf	145	play
2	zhaoy	145	play
3	11711906	145	play
4	11712121	140	play
5	11710123	120	play
6	11711815	120	play
7	11710205	115	play
8	11712611	115	play
9	11710217	110	play
10	11712325	110	play

Figure 10: Rank4

The final score:



成绩单项目	分数
Lab1	100 /100
Lab2	89 /120
Lab2Rank(200)	41 /200
Lab3	92 /120
Project1ProgramScore	94 /120

Figure 11: Rank4

3.5 Conclusion

The biggest disadvantage is that I use DFS when using minimax search tree.In fact, BFS is probably better, because if something goes over time, the worst case for DFS is that the program doesn't add any points to the board, or it finds a point where the score isn't that high.The BFS outputs the current optimal result.

Second, because of the time and the difficulty of code implementation, I did not write the hash substitution table. With BFS, the depth of minimax search tree can be greatly increased by recording the cases that have been run in the hash table.

Third, I think my strategy to deal with the time limit is worth learning. Thanks to the parameter adjustment in the third stage, my score in the third stage is higher than that in the second stage.

Finally, my kill chess so bad that it cannot pass the pretests. Kill chess is a very smart method. As long as you can find a continuous Alive3 or Rush4, go them continuously and the enemy cannot defense it.

Acknowledgements

Thanks for the platform SA built. Although it has so many bugs, it spent two weeks of great nights with us.

References

[1] R. H. Case, "Gomoku.," *Computer Programs*, p. 68, 1971.

[2] Aho, A. V, Corasick, and M. J, "Efficient string matching: an aid to bibliographic search," *Comm Acm*, vol. 18, no. 6, pp. 333–340, 1975.

[3] M. Müller, "Global and local game tree search," *Information Sciences*, vol. 135, no. 3, pp. 187–206, 2001.