Gomoku Project

Zhouying 11610701
School of Computer Science and Engineering
Southern University of Science and Technology
11610701@mail.sustc.com

1. Preliminaries

Gomoku is a game played on a go board with players alternating and attempting to be first to place five counters in a row [2]. This project need us to implement the AI algorithms of Gomoku according to the interface requirements and submit it to the system as required. The chessboard is initially in size of 15 by 15. Because computer does not have the natural thinking ability like humans do, it has to do some extra searching with evaluation to make up this shortcoming. Our goal is getting the victory under 100MB for memory, 5 seconds for one step and 180 seconds in total.

1.1. Software

The code is written in Python, compiled using Python3.6. Only NumPy package is extra imported.

1.2. Algorithm

The main algorithms are implemented including Aho-Corasick algorithm and Negamax algorithm. A heuristic strategy is used for reducing computational cost.

2. Methodology

2.1. Representation

- **chessboard**: a two-dimensional NumPy array
- **chess status**:
  - COLOR_BLACK: black chess
  - COLOR_WHITE: white chess
  - COLOR_NONE: empty chess
- **chess pattern**
  - WIN5
  - ALIVE4
  - RUSH4
  - ALIVE3
  - SLEEP3
  - ALIVE2
  - SLEEP2

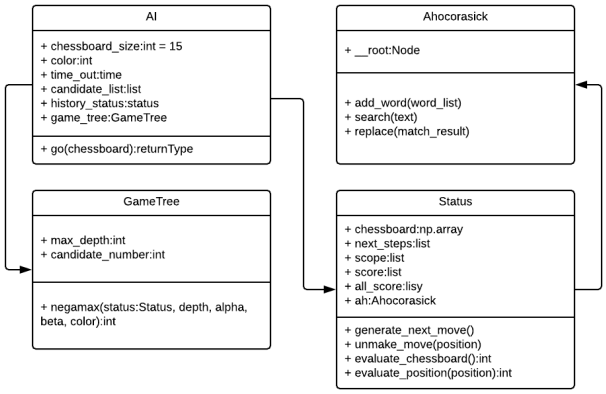

Figure 1. UML Class Diagram

2.2. Architecture

The UML Class Diagram of this project is shown in Figure 1.

2.3. Detail of Algorithm

2.3.1. Aho–Corasick Algorithm. The Aho–Corasick algorithm is a string-searching invented by Alfred V. Aho and Margaret J. Corasick [1]. It matches a finite set of strings within an input text simultaneously. You can go to the reference to find the details of the implementation. I wrapped this algorithm for convenience. The input is a set of patterns and a text string, and it will output a set of patterns which are matched in the text.

2.3.2. Negamax Algorithm. Negamax algorithm is a variant form of minimax algorithm. It relies on the fact that $max(a, b) = -min(-a, -b)$ to simplify the implementation of the minimax algorithm. Due to the limitation of time, I set the max_depth to 5.

2.3.3. Heuristic Search. On the one hand, Negamax algorithm has a heavy dependence on the order of scanning for each layer of nodes, because the alpha generated by the previous nodes directly determines how much the subsquent are pruned. On the other hand, the number of nodes in

each layer also has a great influence on computational cost. Therefore, I only select the top five nodes with high score in remaining empty positions on the chessboard.

Algorithm 1 Negamax Algorithm

```
1: function NEGAMAX(depth, alpha, beta, col)
2:     if depth = 0 then
3:         return Evaluate()
4:     end if
5:     GenerateNextMove()
6:     while MovesLeft() do
7:         MakeMove(pos, col)
8:         val = −negamax(depth − 1, −beta, −alpha, −col)
9:         UnmakeMove(pos)
10:        if val > alpha then
11:            alpha = val
12:            if depth = max_depth then
13:                AppendCandidates()
14:            end if
15:        end if
16:        if alpha >= beta then
17:            return beta
18:        end if
19:    end while
20:    return alpha
21: end function
```

Algorithm 2 GenerateNextMove

```
1: function GENERATENEXTMOVE
2:     AddRemainningEmptyPosition()
3:     for pos in NextSteps do
4:         score ← EvaluatePos(pos)
5:         candidates ← score
6:     end for
7:     heapq.nlargest(5, candidates)
8: end function
```

Algorithm 3 Move

```
1: function MOVE(pos, col)
2:     chessboard[pos] = col
3:     UpdateScore(pos)   ▷ score[2][72] stores the chessboard score (2 characters 72 lines, including horizontal, vertical, slash and backslash) allScore[2] stores total score of chessboard (2 characters)
4:     UpdateScope(pos)   ▷ scope[4] stores up, down, left and right boundary value
5: end function
```

Evauate Position Score. First, it supposes this position is black or white chess, and scans the relevant range of vertical, horizontal, slash, and backslash direction of this position as Figure 2 shown. Next it matchs four direction lines with chess patterns and add score corresponding to the chess pattern. Finally, getting position score by adding white chess score and black chess score. When this position score stands out, it represents that it is important for black chess or white chess or both.

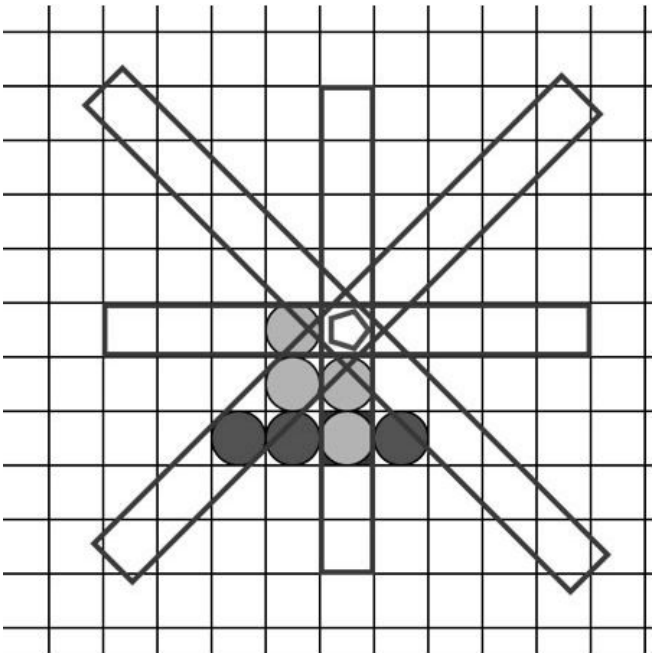

Figure 2. Evaluate Position

3. Empirical Verification

The code has passed the tests in sakai and it has been reviesed many times by playing against others on the platform.

3.1. Data and data structure

The data structure used in this project including heap, list, dictionary and etc.

3.2. Performance

Fifth place in the round robin.

3.3. Analysis

It still has many places to improve. For example, the evaluation function is not accurate enough.

Acknowledgments

In the first artificial intelligence project, the code is above five hundred lines including comments. Although the code is not very long, I did spend a lot of time on the debug. All races finish in the end. Thanks to those friends who have helped me a lot.

References

[1] Aho, Alfred V.; Corasick, Margaret J. (June 1975). ”Efficient string matching: An aid to bibliographic search”. Communications of the ACM.

[2] ”Gomoku - Japanese Board Game”. Japan 101. Archived from the original on 2014-03-26. Retrieved 2013-06-25.