Gomoku Report

Yilin Zhang 11510280

School of Computer Science and Engineering
Southern University of Science and Technology
11510280@mail.sustc.edu.cn

October 25, 2018

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 Go board, usually using 15*15 grid board.Players alternate turns placing a stone of their color on an empty intersection. The winner is the first player to form an unbroken chain of five stones horizontally, vertically, or diagonally.

This project is to implement a AI algorithm of Gomoku according to the interface requirement, and upload codes to the web battle platform, match with other Gomoku AIs.

1.1. Software

This project is written in Python using IDE *PyCharm*. The libraries being used includes *Numpy* and *Random*.

2. Methodology

I mainly used value function to determine the next step in this project. I also tried hard to use minmax search tree and alpha & beta prunning, but the performance was not satisfactory that it easily to be time out.

2.1. Representation

There are some given constants and variables such as **chessboard_size**, **color**, **chessboard**, **candidate_list** and, **chess_type**, a self define dictionary, that are essential to use in the program.

- **Constant:**
  - **COLOR_BLACK:** black chess
  - **COLOR_WHITE:** white chess
  - **COLOR_NONE:** none chess
  - **chessboard_size:** edge length of chess board
  - **color:** the role you played in this game
  - **chess_type:** chess types and their corresponding values, details refers to *table 1*
- **Variables:**
  - **chessboard:** two-dimensional matrix to present chess board state
  - **candidate_list:** a list of empty candidate position, system will get the end of the list as final decision.

2.2. Architecture

Search algorithm, pattern matching are two main algorithm being used in this project. I defined other two functions *candidate* and *check* to modularize codes. With two given functions *__init__* and *go*, there are four functions in the Python file *11510280_Gomoku.py*, all of them are defined in class *AI*.

- **Given:**
  - **__init__:** init function of object *AI*
  - **go:** Be called when your turn

Table 1: *chess_type*

Pattern	Value
AAAAA	50000
?AAAA?	4320
AAAA?	730
?AAA??	730
?A?AA?	710
AAA?A	710
AA?AA	710
??AA??	120
A?A?A	120
AAA??	120
??A???	20

- **Self define:**
  - **candidate:** find all none chess positions which near chesses within two cells, return a set
  - **check:** check and calculate when a certain color put on the position (i, j), return a int

2.3. Algorithm

First, find all none chess positions which near chesses within two cells. Then, calculate the benefit (value) by **formula 1** when we choose these positions as next step by using search algorithm to match patterns, where k_1 and k_2 is the coefficient of values. When $k_1 \geq k_2$, the chess style of AI is more aggressive and otherwise, is more conservative. Finally, find out the position with the max value, append it in *candidate_list*. Here are details of these algorithms.

$$Value = k_1 * Value_{self} + k_2 * Value_{enemy} \quad (1)$$

Algorithm 1 candidate

Input: chessboard
Output: A set of all none chess positions which near chesses within two cells.

1: **for all** *emp* such that *emp* on *chessboard* and *emp* == *COLOR_NONE* **do**
2: If there is a chess near to *emp* and dis(*chess*, *emp*) <= 2;
3: Add *emp* to *candidate_set*;
4: **end for**;
5: **return** *candidate_set*;

Algorithm 2 check

Input: chessboard; i, j; color
Output: Value of position (i, j)

1: Search from (i, j), record four *patterns* in all four directions until meet opponent's chess.;
2: Matching *patterns* with *chess_type*, get corresponding values v.;
3: $value \leftarrow v_1 + v_2 + ... + v_n$;
4: **return** *value*;

3. Empirical Verification

3.1. Experiments

Experiments have been given in file *code_check.py*, and we can use it by modify the file path in file *code_check_test.py*. There are some simple tests in *code_check.py*, including init *AI* object, testing if the ob-

Algorithm 3 go

Input: chessboard

1: **if** chessboard is empty **then**
2: add $(size2, size2)$ to *candidate_list*;
3: **else**
4: Find out all *candidates*;
5: Check all *candidates*;
6: Add the *candidate* with largest *value* to *candidate_list*;
7: **end if**

ject is usability when the game start or in progress, and if the code is strong enough to handle base tests. My code has passed this test.

3.2. Data and data structure

Data being used in this project is chessbord and the color of the chess on our side. Data structure used in this project includes dictionary, array, tuple, and set.

3.3. Performance

Since this is a project with interaction process, so the performance cannot be simply measured by actual running time. So, theoretic running time, computed by time complexity of functions is about O(n^2), which is good enough to play in limit time when tested on oringinal chess board with 15*15 size.

3.4. Analysis

Here the code I stated is just based on the evaluation function, which is a simple linear calculation.

Besides, I did implement another algorithm based on minmax search tree and alpha beta prunning, but it is too slow to pass the time limit due to the large search space when I set the search depth to 3. Whereas, if I set the depth to 1, there is no difference with using evaluation function directly. So, I didn't represent this method's details here. But for the further work, maybe this problem can be solved by optimazing calculation of the chessboard value and sorting choices before every recurse.

References

[1] H.A. Dong, *Research and implementation of Gomoku computer game system*, Shandong Normal University, 2005.