Gomoku Report

Xinhuai Deng 11610320
School of Computer Science and Engineering
Southern University of Science and Technology
Email:11610320@mail.sustc.edu.cn

1. Preliminaries

This project is a artificial intelligent algorithm designed for Gomoku. Gomoku is a very famous and traditional game where two players put different chess in turn to get a five connection for the win. What is more, it is perfect information and zero-sum game. That is to say, every state in such a game is deterministic and the environment is fully observable. [1] So the state of Gomoku is easy to represent and the goal of an agent is conflict. These become the reason why I choose classical adversarial search algorithm - The Mini-Max algorithm, which is stable and competitive.

The software used in the project is Python 3.6 with IDE called Pycharm. And the packeges used in Python is numpy and random.

2. Methodology

In this section, the general representation and structure of the project will be presented first, and then I will discuss how I designed my code and accelerate it.

2.1. Representation

The chessboard has already defined in the given project interface as a numpy 15×15 array, and value 1 represents the White chess while -1 represents Black chess. The default value 0 means there is no chess in the current position.

With regard to the representation of Gomoku game state, I use a simple python tuple to represent it. The five elements of the tuple are shown in table1.

TABLE 1. STATE TUPLE

state[0]	chessboard
state[1]	player
state[2]	last move
state[3]	utility
state[4]	board hashing value

2.2. Architecture

As required, the project should contain a class named AI, and it should contain a method called *go*. And *go* take a parameter of the chessboard and returns a candidate list where the last item will be the next move.

The project contains two classes and the general structure of them are listed as follows:

- **AI:**
  - ***go(chessboard).*** This method takes current chessboard and returns the candidate list where the last item will be the next move.
  - ***first_chess().*** Because the minimax algorithm is very slow for the first and second move and it is not necessary to search the adversarial tree, I set the first move to be the center position (tianyuan).
  - ***second_chess().*** When the color is white, the first move of this player is set as the neighbor of the first move.
- **Game:**
  - ***actions(state).*** This method takes current chessboard and returns all the empty point where chess can be put.
  - ***result(gamestate,move).*** This method takes the current move and the previous state to generate a new state which is consisted of current chessboard and player and the utility of the state.
  - ***terminal_test(state).*** This method tests whether the state is a terminal state. And in Gomoku game, the terminal state should contain a chessboard where there is a five connection in an arbitrary direction.
  - ***alphabeta_cutoff_search(gamestate,search_depth).*** This method is the key method of my adversarial search, using alpha-beta pruning and cutoff terminal test.
  - ***eval_fn(state).*** The method is employed when the cutoff terminal test is true. It will return a value to represent the current situation of input state. And the higher the grade is, the more advantageous to the player it can be.

2.3. Mini-max tree search Algorithm

Generally speaking, the project is actually a state-space search algorithm which could roughly be separated into four

parts: states, actions, transition model an terminal test. The structure of state and the description of the action and the terminal test has been shown in the previous subsection. So in this subsection, I will introduce the detail of my alpha-beta cutoff search which is the transition model of my state-space search.

Algorithm 1 Mini-Max cutoff algorithm

Input: the current game state
Output: the best actions in actions($state$)

function ALPHABETA_CUTOFF_SEARCH($state$)
    $value \leftarrow$MAX-VALUE($-\infty, +\infty, 0$)
    return the action with max value $value$
end function

function MAX-VALUE($state, \alpha, \beta$)
    if TERMINAL-TEST($state$) **then**
        return $+\infty$ if win, $-\infty$ if lose
    end if
    if CUTOFF-TEST($state$) **then**
        return EVAL-FN($state$)
    end if
    for each a **in** ACTIONS($state$) **do**
        $v \leftarrow$MAX(v, MIN-VALUE(RESULT(state,a), α, β))
        if $v \geq \beta$ **then**
            return v
        end if
        $\alpha \leftarrow$ MAX(α, v)
    end for
end function

function MIN-VALUE($state, \alpha, \beta$)
    if TERMINAL-TEST($state$) **then**
        return $+\infty$ if win, $-\infty$ if lose
    end if
    if CUTOFF-TEST($state$) **then**
        return EVAL-FN($state$)
    end if
    for each a **in** ACTIONS($state$) **do**
        $v \leftarrow$MIN(v, MAX-VALUE(RESULT(state,a), α, β))
        if $v \leq \alpha$ **then**
            return v
        end if
        $\beta \leftarrow$ MIN(β, v)
    end for
end function

2.4. State Evaluation

When the return value of the cutoff test is true, the evaluation function will be executed to return a value which represents the beneficial value for the win. And in the evaluation function, a position evaluation function called $eval\_posi()$ is employed. It will return the score of a position by counting the connection and transferring connection to score using the score setting table2. [2] After getting the grades of each **empty** point, sum the all the grades up and then name it as val. Initially, val become the only value to evaluate a state. But I found this make the Agent be confused with several special patterns such as two-three or three-four. So I add two parameters to detect the "must win" or "must lose" situation. Finally, the evaluation equation can be expressed as

$$EVAL(state) = val + defense + attatck.$$

And for more details, see algorithm2

TABLE 2. THE BASIC SCORE SETTINGS

Pattern	Score	Pattern	Score
Five	10000000	next_win	99900
Four	100000	BlockedFour	10000
Three	1000	BlockedThree	100
Two	100	BlockedTwo	10
One	10	BlockedOne	1

Algorithm 2 Evaluation function

Input: the current game state
Output: the evaluation value representing the possibility to win

Initialize two empty set called my_must_win, op_must_win
for all empty point i **do**
    v $\leftarrow$ EVAL-POS(i)
    # do not count small value which is negligible
    if $v > 100$ **then**
        val $\leftarrow$ val + v
    end if
    if $v > score$['FIVE'] **then**
        add $i \Rightarrow my\_must\_win$ if in my turn else op_must_win
    end if
end for
if len(my_must_win)≥ 2 **then**
    att $\leftarrow \infty$
end if
if len(op_must_win)≥ 2 **then**
    def $\leftarrow -\infty$
end if
return val + def + att

2.5. Algorithm acceleration

Even though I used α-β pruning and cutoff evaluation, it also takes a lot of time to search the adversarial tree. So it is necessary to take actions. I implement an approach to reduce the search space and use Zobrist Hashing [3] to avoid the duplicated search.

2.5.1. Informed Children Node Expansion. Instead of expanding all the children of a parent node, I use a sorted list to represent the children. And for each node, the

ACTIONS($state$) will return the first k items. And the sorting value for each child is calculated by EVAL_POS($state$) method. After empirical verification, I found this approach can significantly reduce the running time and will not influence the result until the EVAL_POS($state$) method calculated incorrectly or k is too small.

2.5.2. Zobrist Hashing. Zobrist Hashing is a popular and easy way to avoid repeated calculations. In this way, each chessboard has a hashing value that identifies a unique situation.

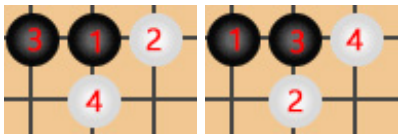

Figure 1. An example for Zobrist Hashing.

For example in Figure1, two different move sequences may result in the same board. So when program evaluates a chessboard, the order of the past move sequences does not matter. Zobrist Hashing is designed for such case. And the steps for using Zobrist Hashing in Gomoku are as follows:

- ***Initialize.*** Initialize two random matrixes (large integer) of size 15×15 for both white and black player, denoted as M_W and M_B. And use number zero to represent the origin chessboard denoted as $board\_hash$. Finally, initialize a Python dictionary named $eval\_dict$ for later use.
- ***Take move.*** When white player takes move (i, j), $board\_hash = board\_hash \wedge M_W[i][j]$. Similarly, When black player takes move (i, j), $board\_hash = board\_hash \wedge M_B[i][j]$.
- ***Evaluate with dictionary.*** When doing evaluation, check the $eval\_dict$ use key $board\_hash$. If key exists in $board\_hash$, fetch the value and compare the depth to decide whether use it. Otherwise, do the evaluation for the current state, and store the current depth and the current $board\_hash$ paired with evaluation value into the $eval\_dict$.

Here $\wedge$ is the logical operation XOR.

3. Empirical Verification

The very first verification I employed was the given Python file name $code\_check.py$ that can check whether my algorithm is applicable. After my program can run normally, the most important part becomes how to impove the power of my Gomoku algorithm.

3.1. How to get debug data

Appreciating this semester's Gomoku website, I can easily download the chessboard data where I find my program do not take a correct move. What is more, I can play against my algorithm, so that it becomes practical that I try to set some special case and think why the program will get this result.

3.2. How to debug

After getting the chessboard data, I found that the data is represented in many rows where each row indicates a move. So I write a method called log_to_chessboard($filename$) that can take a file name and return the corresponding chessboard. And then I can judge my program similar to the $code\_check.py$. But it is still a struggle for me to debug step by step because the minimax search algorithm is written recursively. So I wrote a simple print method that can correctly print the tree view results through different indent length. With the tree view, I can determine which step make mistake and then modify it.

3.3. How to judge performance

There are generally two perspectives to judge the performance. One is running time for each step and the other is the accuracy of the value returned by my evaluation function. And I use time method to easily judge the running time and use step-by-step visualization provided by project website.

3.4. Result and Analysis

After acceleration, my algorithm can search at most six depth with the length of children list being 4. But in this case, sometimes the running time of one move may exceed five seconds, especially when there are many empty points. So I set the search depth to be five and the length of sorted children list to be five. In this case, the average step time is 2 seconds which is much stable. And when I play against my algorithm, I tend to not win until I pay much attention.

Because of the time limitation, my algorithm can not search both deeply and widely. So the evaluation function takes an important role in the game. Finally, my algorithm can beat over 90% algorithm uploaded in the website.

References

[1] S. J. Russell and P. Norvig, *Artificial intelligence: A Modern Approach*, 3rd ed. Pearson Education, Inc., 2010, p. 161.

[2] S. Zhang, "zhangshun97/ai_gomocup," 2018. [Online]. Available:https://github.com/zhangshun97/AI_Gomocup/blob/master/final/score.py

[3] B. Bouzy and T. Cazenave, "Computer go: an ai oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.