Midterm Report for AI Final Project: Gomoku

Jialun Shen 16307110030 Jinming Liu 17307110101

I. Introduction

In this project, our goal is to create our own AI agent(Name: Eggplant) to play the Gomoku game. As a midterm progress, we have implemented an agent using Minimax with Alpha-Beta Pruning to search for the best step given the current situation. This report covers some details of the design of our AI agent.

II. Basic Algorithm

1. Representation of the Current Board and its Successors

We use the same board as given in ***example.py*** as our board, which is a 2-dimensional list of size **pp.height×pp.width** (in this case, 20×20). **board[x][y]** denotes the chess of position (x, y) on the board: 0 means the position is free, 1 means the position is occupied by me(our AI agent), 2 means the position is occupied by the opponent.

We have defined a **Node** class to represent the relationship between the current state and its possible successor states. **Node.successor** gives all the successors of the current Node, and **Node.value** stores the board.

2. Find Possible Successors

A successor of a state is given by the function **get_son_position(state)**, where **state** is a board. The function returns a list of positions we may put a chess given the state.

For an empty position (x, y) on board, we look at its 2*8=16 neighboring positions around it, and add it to the list if only there is an occupied (whether by AI or opponent) position. As a special case, when the board is empty, we simply choose the central position as the starting place.

Note that we do not need to choose all the empty positions as a possible position for the next step. An empirical explanation is that all positions we may put a chess is within two blocks from all the current occupied positions, in all 8 directions. In this way, we can decrease the size of our search space.

3. Minimax Search(Adversarial Search) with Alpha-Beta Pruning

We use minimax search (adversarial search) to search for a "best" move given the current state, implemented as **find_position_by_alpha_beta(node)**. Our depth of search is 2.

Gomoku is a zero-sum game, gain of one player leads to loss of the other. Therefore, it is reasonable for us to choose adversarial search as our search method.

The basic idea of adversarial search is simple: our AI agent maximizes its utility (in this case, the chance of winning), while our opponent minimizes it. Nodes are maximized if AI moves, otherwise minimized. Recursively

We cannot search till the terminal state (either player wins) for each step, which may have a large computational cost. Instead, we use a depth-limited search. For an arbitrary state, a evaluation function gives a value indicating how likely it is for our AI agent to win.

To accelerate the adversarial search, we implemented alpha-beta pruning.

4. Evaluation of a situation

The key part of adversarial search is the evaluation function. A good evaluation function must be correct, quick and consistent. Our evaluation function is implemented as **evaluate(board, position, player)**.

getline(position, distance) is a support function that returns a list of eight lists(stands for eight directions), each list is positions 1 to **distance** away from **position** that are on board.

The main idea of our evaluation function is as follows:

For a board, a new position, and index of player, we use **getline** to get lines with distance 5, 4, 3, 2, 1, and give values according to the pattern we observe, then take the maximum value of all patterns. Five in a row is a special case, indicating that the game is already over. Our AI agent is offensive. For example, suppose AI uses white(○): if we observe a consistent four white(○), we give the position a value of 10000; if we observe a consistent four black(●), we give the position a value of 9000<10000. A complete table of values for different patterns is given below:

Pattern	Strategy	Value	Pattern	Strategy	Value
○○○○○	/	0	●●●●●	/	0
○○○○	offensive	10000	●●●●	defensive	9000
○○○	offensive	1000	●●●	defensive	900
○○	offensive	100	●●	defensive	90
○	offensive	10	●	defensive	9

Table 1 Values for different patterns, suppose AI uses white(○)

In practice, we need to evaluate the values of a series of positions $\{p_1, ..., p_n\}$ according to their corresponding boards $\{b_1, ..., b_n\}$ (n stands for depth of search).

$$V(p_n, b_n) = \sum_{i=1}^{n} (\gamma^{n-i} V(p_i, b_i))$$

where γ=0.1 is the decreasing factor. In this way, we take into account the former possible steps, but with a lower priority.

III. Performance against Mushroom

Our AI agent beats the Mushroom agent in all three openings:

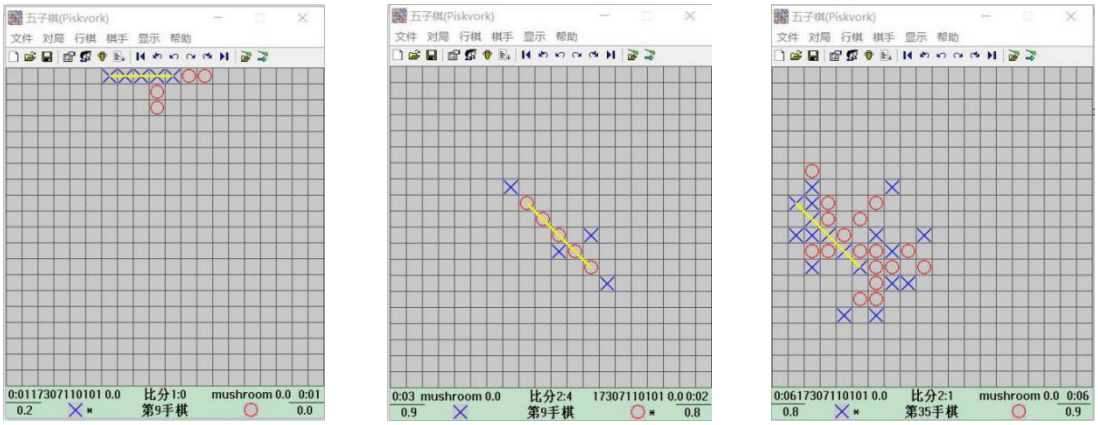

Figure 1 Performance of our AI against Mushroom

IV. Future Improvements

1. We found that a change in the evaluation function may have a dramatic influence on the performance of our AI. Our rules for evaluation is still simple, considering only basic patterns like continuous chess. We can add more complicated patterns, like ○×○×○ (× means empty). In fact, we have another (more complicated) version of evaluation in progress (in ***util.py***), which still requires further debugging.

2. We can increase the depth of adversarial search.

3. We can try other algorithms, including Monte Carlo Tree Search, Proof-Number Search, Genetic Algorithm, etc.

References:

[1] Stuart J. Russell, Peter Norvig (2009) Artificial Intelligence A Modern Approach, 3rd Edition.2009, Prentice Hall

[2] Go-moku and threat-space search(1993), Louis Victor Allis and Hj Van Den Herik.

[3] 五子棋 AI 教程第二版，言川, https://github.com/lihongxun945/myblog/issues/11