CS 165A Machine Problem 2 Report

Nikita Buzov

06 / 07 / 2020

**Architecture.**

The gobang.py program defines all the functions that were used to play Gomoku game. The main() function starts executing once the program starts and takes in the arguments that it passes on to the play\_gomoku() function that contains the whole gameflow. Then the board is created, and the game starts with players taking turns. At the end of each turn the function checks if any of the players have won or there is a tie. When it’s the computer’s turn, getComputerMove() function is called, which itself calls for another function called generateMinimaxMoves() to evaluate the current situation and return the best move. However if it’s the first move, then just a random move somewhere in the middle region of the board is made. The way computer evaluates the board and different options for next move is described in the following section.

**Search.**

The algorithm that my best-performing AI agent uses is Minimax with search depth of 2 and alpha-beta pruning. I have tried to deepen the search to depth of 4, but that had slowed the program down by at least an order of magnitude, and things get worse as the game progresses since the search space grows (the code with this configuration is in gobang\_d4.py file). In order to reduce the search space and speed-up the search algorithm, I have only considered adjacent moves to stones that are already on the board. This makes sense because good Gomoku moves most of the time are in the region close to other stones, since the goal of the game is make five-in-a-row. All these potential moves are stored in a set. Then I start building a nested dictionary: for each move (dictionary key) that MAX can make (from the aforementioned list) make another dictionary with MIN moves as keys and values from my evaluation function. For each combination of MAX and MIN moves I compute a score based on my heuristic function (explained in the following paragraph). While computing each leaf of this tree, it is being compared to alpha/beta, according to alpha-beta pruning algorithm, to avoid unnecessary computation. Once all the leaf nodes are computed, the maximum and minimum values are being passed up the search tree to find the best move MAX (AI agent) can make at this state of the board, as stated by the Minimax algorithm.

The way my evaluation function setup is straight forward. The function getScore() checks the variation of the board where MAX and MIN have made their possible moves, counts weighted useful combinations for both players, and then subtracts player’s score from AI agent’s score. Therefore, we can see which possible scenario is better for the AI agent. The useful combinations and the weights I used are the following (format: ‘combination name’: [ weight, [combination strings]] , where ‘o’ indicates open, ‘n’ – opponents stone or end of the board, ‘x’ – players stones):

{'five': [200000000, ['xxxxx']],

'four2open': [2000000, ['oxxxxo']],

'four1open': [1000000, ['nxxxxo', 'oxxxxn']],

'three2open': [40000, ['oxxxo']],

'three1open': [15000, ['nxxxoo', 'ooxxxn']],

'voidFour2open': [7000, ['oxoxxo', 'oxxoxo']],

'voidFour1open': [3000, ['nxoxxo', 'nxxoxo', 'oxxoxn', 'oxoxxn']],

'two2open': [1000, ['ooxxo', 'oxxoo']],

'two1open': [400, ['nxxooo', 'oooxxn']],

'voidThree2opens': [100, ['oxoxo']],

'voidThree1open': [40, ['nxoxoo', 'ooxoxn']]}

Basically, each potential future board configuration was converted to a list of strings that represent rows, columns and diagonals, and then strings were search for matches to the patterns shown above. Ultimately, the more such patterns are on the board the greater the score for that particular player is. Another trick I employed was the short-cut plays, the moves that are certainly good. There are four situations like this:

1. a winning move is available, so the player must go for it;
2. opponent can win in one move, then the AI must block that;
3. opponent might form an open-four combination of stones (which is a sure loss in one move), so the AI agent must prevent such scenario;
4. AI agent can form an open-four itself, so it should make that move.

The main optimization solutions were the extensive use of sets and dictionaries, since processing them is much quicker than other information storage options. Also reducing the search space by only considering next moves that I adjacent to the current stones on the board has dramatically increased the processing speed.

**Challenges.**

There were two major challenges: how to search fast and what heuristic evaluation function to use. While Minimax with alpha-beta pruning is fairly fast, still with all possible ways to speed up the program, going through the search tree and computing the score for leaf nodes, I could not search deeper than depth = 3 within reasonable time. Ultimately, I chose depth of 2 to have enough spare time and compensated with a more explicit heuristic function. Regarding the heuristic, I have found the most important patterns in Gomoku and stored them in the program so my AI agent can search for them and try to make moves that form such patterns that yield wins. I have also introduced the short-cut moves as mentioned in the above section, so if the agent comes across a move that it definitely must take, it stops the search and make the move.

**Weaknesses.**

One of obvious weaknesses is the fact that my program can’t directly evaluate forks, which would be very useful, since usually that’s the way to win or prevent a loss. I attempted to do that, but such feature requires a more complicated way to represent the board and combinations.

Another weakness is in the way I search for pattern matches. My program only sees non-overlapping patterns on the board, this is definitely not ideal because it misleads the AI agent and misses some forks. One remedy to this problem is just to redefine my matching function so it can count overlapping patterns.

Another possible improvement could be made by computing the search tree while the opponent is making a move, since my AI agent is idle while it’s player’s turn. A solution to this is to expand the leaf nodes while its opponent’s turn and then when he made a move consider that particular branch.