Gustavo Cornejo
CS165A

Architecture

For this project, my solution was written in python 2.7.11 as it's the default python version on CSIL. The architecture of my code is fairly simple as it only features one main calss, Gobang. This class provides all the functions and variables needed to keep track of the game. For example, the self.gameboard variable is what I use to represent the gameboard. It is a 2 dimensional python list that is instantiated based on the size given. Besides the size of the board, I also have self.player and self.AI to keep track of what colors each player is playing as.

The Gobang class features a large set of functions, most of which are simply helper functions. I'll explain all of their functionality by focusing on each of the main functions, current_board, print_board, get_perimeter, find_max, place_move, and evaluate. The function current_board is a simple function that just returns the current state of the board. To print out the ASCII representation of the board, I used the referee.py print_board function. The function get_perimeter takes in a move and a board size, and returns horizontal and vertical minimums and maximums for a given point in the board so that it provides an 11x11 sub-board. For example, if you give the point 2,3 on the board which is 26x26, it will return a bounding perimeter of horizontal min = 0 and max = 10. While the vertical min is 0 and vertical max is 10. The main purpose of this is that in some parts of my algorithm I only consider a smaller subset of the board to improve efficiency. To achieve this it uses helper functions hor_bounding_box, ver_bounding_box, and it_fits.

The other main function is find_max which serves as the entry point for my minmax algorithm. I only observe up to depth 2 so there is no recursion needed, only a call to the function find_min. In find_max we take a certain set of available moves using the helper function get_avail_moves to provide the search space for the algorithm. Then, for each one we call find_min. Within these functions we also have the alpha beta functionality built in to limit search space. Inside of find_min we evaluate the result of each move.

The other main function is my evaluate function which is composed of 3 rather large helper functions, diag_eval, col_eval, and row_eval. The functionality here is for each of these functions, they loop through the board and count the number of consecutive pieces for both light and dark. At the end we multiply the values to specific weights (explained later in report) and subtract the score minus the opponents score to determine the next move. These functions handle the heuristic and search of my algorithm.

Finally, to complete the functionality, the function place_move provides an interface for both AI and human to alter the board.

Search

My implementation is built on a depth = 2, minimax algorithm. The entry point is the function find_max , which when given a gameboard, will first retrieve a set of available moves. I implemented an optimization that would trigger when the board became larger than size 11. When triggered, instead of retrieving the entire board as possible moves, it calls the get_perimeter function I mentioned earlier, to provide an 11x11 bounding box around my last move. Then the find_max function will only consider moves that are found within this bounding box instead of the entire board. Either way we retrieve a set of moves that are given to the function find_min that provides the other half of the minimax algorithm. Before I explain the evaluation heuristic, I will provide a high level view of my algorithm. Find min then uses the evaluate function to find the move that returns the smallest possible value to return to find_max. Find_max then sorts through the min's to find he highest one. That is how my minimax of depth 2 works. Inside of my minimax I also provided alpha-beta pruning to eliminate branches that I would have to search. I simply pass in a 'best' variable into the find_min function and say that if the value from the current one is less than the best already seen by the find_max function, then just exit since there is no need to further continue in the search of that branch.

My evaluation heuristic is called on each move inside of my find_min function and is the essential piece of my algorithm. As explained earlier, my evaluation function is composed of 3 large helper functions diag_eval, col_eval, and row_eval. Each one returns two python lists, whiteCount and blackCount. These are of size 5 and are used to count the number of 'n in a row' pieces for light and dark values. Here index 0 is for 2 in a row, index 1 is for 3 in a row, index 3 is for 4 in a row, and index 4 is for 5 in a row. Essentially, these 3 helper functions traverse every row, column, and diagonal that spans at least 5, and update their respective blackCount and whiteCount arrays. After calling each one, I add their values for blackCount into a total blackCount and a separate total whiteCount for their whiteCount values. I then take whichever array (blackCount or whiteCount) that corresponds to the AI and multiply the index 1 by 1, index 2 by 2, index 3 by 6 and index 4 by 100, then add all of the values together. Then the array that is left I multiply its index 1 by 1, 2 by 2, 3 by 60, and 4 by 100, and also add them together. Finally, I subtract these values and return the score to the find_min function. I gave heavier weights to the opponent as I felt that my algorithm was not playing defensively enough as a result of a bad heuristic. I discuss this and other potential weaknesses later in the report.

Challenges

One of the main challenges that I faced was run time. Given the way I was searching through my board for 'n pieces in a row' I would search numerous rows on every call to my evaluate function. Although it was fine for 5x5 it would take a long time to pick a move in a 26x26 board. To optimize, I restricted the possible move set by only allowing moves to be picked from an 11x11 perimeter on my last move made. This increased the performance of my solution greatly, allowing me to continue to evaluate the entire board for outcomes, but only consider reasonable moves. Another optimization I made to speed up my code was implementing alpha-beta pruning. By giving the current best score on find_max to my find_min

function it was able to determine if the current branch of moves was worth looking into. By reducing the number of branches to look into I was able to have even better performance.

The other main challenge was creating an adequate heuristic function. I started with the one provided on piazza but quickly realized it was not good enough as I was easily defeating my AI in 5x5 games. I then set my own weights based on how many n in a row pieces my search function had seen. 10 for 5 in a row, 6 for 4 in a row, 2 for 3 in a row, and 1 for 2 in a row. I did the same for the n in a row pieces of the opposing color and subtracted these values to determine a score for each possible move. I then noticed that my algorithm would sometimes block a 3 in a row move instead of a 5 in a row. So I switched 10 to 100 since the 5 in a row is a more important move. I still noticed my AI not performing optimally which may have been a fault of just a bad heuristic, so to offset for this I gave a heavier weight to some of my opponent's n in a row which made it played defensively. Ultimately, I ended up with an adequate AI for any board size.

Weakness

One of the main weaknesses of my code is my heuristic. I believe there is more to consider then just # of pieces in a row. For example, my algorithm cannot tell that there is no need to pursue getting more pieces in a row, in a row that does not have 5 spaces left. In addition, my code cannot distinguish from a move that has 3 in a row alone, or 3 in a row blocked on either side. Obviously we want to have moves that are not blocked as there is no purpose to 4 in a row if there is no room to grow to 5. The other main weakness is that my depth only goes to 2, this was because any greater gave a huge performance decrease. Perhaps changing my search to a method that does not need to scan the entire board would allow me to increase depth and not lose performance.

Note:

I wanted to note that I was not able to correctly interface with the referee.py file. I was having trouble running it past 1 move for some reason, I tried to flush std but it still would not work. I just wanted to say that my algorithm works adequately, just not with the script. Thank you.