Jake Stenger
CS 165A
MP2

Architecture

I chose to implement gobang in native Python, The main method reads in the command line arguments, assigns a board size and light and dark functions accordingly. The board is represented as two n by n boolean arrays representing the positions of the light and dark stones. The loop alternates between the human and minimax functions, which take in the board and handle the printing and modification of the arrays. It then checks for the end of game conditions, and prints the results after breaking. The win function traverses the board to check for five in a row, and another function checks if the board is full.

The human function simply accepts and checks inputs from the command line, modifies the board and prints the move. The minimax function calls a recursive helper function, which returns the value and position of a move. This function recurses while lowering the depth, and calls the heuristic function for the base case. The heuristic function returns the estimated value of the board, with higher values indicating a better state for the maximizer, and lower values for the minimizer.

The single_player_value function manages the traversal of the board to find consecutive stones. It then adds up the values of these rows, which are calculated by one of two functions, self_turn_values and opponent_turn_values. These functions simply look up a value based on the number of consecutive stones and the number of adjacent empty tiles.

Search

I implemented standard minimax search with alpha-beta pruning. The recursive minimax function can take a variable depth, calls itself recursively until it calls the heuristic function, and prunes branches that are unreachable from a parent minimizer/maximizer. I run depth 2 against the baseline, so my edge against it must come from my heuristic function. I calculate all consecutive stones of length two or greater, and assign them a value depending on the number of adjacent empty tiles and whether it is the player's turn. I picked the values of these by hand, and all that matters is their relative magnitudes. I arranged them with very large differences in size so that a more immediate win or loss will always have a higher priority than a less immediate one. I valued rows with two adjacent empty tiles very highly compared to rows with only one, with the intent of building up smaller rows that eventually lead to multiple longer ones, rather than simply extending a row of three for it to be blocked immediately. I found it was useful to play the game against a strong AI online to get a feel for the values of different positions, and then translate that to the values of the heuristic.

One time optimization was in traversing the board diagonally, where I cut off the corners of the board to save some time. I also used a single data structure for the board, which means I only wrote (and reversed) a single boolean value twice per branch rather than copying memory.

Challenges:

The most challenging part of implementation was finding and fixing bugs in the heuristic algorithm, specifically the traversal of the board to find adjacent tiles. For diagonals, optimized runtime and reused code by reversing and flipping the board matrix, and then checking only certain parts of each. I had to make sure that I would not miss any diagonals while doing this.
Another challenge was figuring out how to calculate the heuristic depending on whose turn the leaves were on. I initially used the same function to analyze the stones of the maximizing player, regardless of the turn. This did work somewhat well for depth 1, so it was difficult to figure out what was going wrong for depth 2. I eventually realized that the values for minimizing and maximizing need to be symmetric, one is just the negative of the other. The only factor that should affect the magnitude of the values is whose turn it is.

Weaknesses

My heuristic considers only open tiles adjacent to consecutive stones, which means a five-in-a-row could be blocked by an edge or opponent stones for shorter rows, but the heuristic would still value this highly. It also does not highly value two rows of two with an empty tile between. At higher search depths these problems would become less relevant because the algorithm would be able to see directly to the situations they lead to, but at depth 2 they cause weaknesses.

I was able to beat the baseline with only depth 2 and alpha-beta pruning, but I could have improved the AI greatly with some additional speed improvements that would allow it to run at depth 3 for an 11x11 board. I generally used around 0.5 seconds per turn at this board size, but with depth 3 it took more than 30 seconds. Without further optimization, 11x11 is near the limit of effectiveness and my algorithm times out on slightly higher sizes. To run at higher sizes, I would need to revert to depth 1 and just rely entirely on the heuristic.

I initially was planning on optimizing my heuristic but I now think the best path would be to implement speed optimization techniques with iterative deepening search. Alpha-beta search can be sped up by first expanding branches that are likely to cause a pruning. This means a heuristic should be run on the moves in a certain subtree to rank them before expanding. My approach would be to do this with iterative deepening starting at depth 1. The values of each node calculated with the heuristic would then be stored and used to order the expansion of the next iteration with higher depth. This would lead to more pruning. For very high board sizes, it would also be useful to not consider moves to far from any existing tiles. It would be optimal to change the structure of my minimax code to better allow for sharing of a data structure between depths and iterations, as well as keeping track of the time to cut off the search at various depths.