Jake Stenger
CS 165A
MP2

Architecture

I chose to implement the gobang program 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 stored in 2 arrays, self_turn_values and opponent_turn_values. These arrays simply look up a value based on the number of consecutive stones and the number of adjacent empty tiles.

Search

I implemented minimax search with alpha-beta pruning, a simple move filter and move ranking to increase pruning.

For the heuristic, 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.

Each node of the game tree filters its branches by throwing out all moves that are not adjacent to existing tiles. This may limit some good moves, but it was a great help in reducing the runtime. Each move is then ranked based on the heuristic (ascending for minimizer, descending for maximizer). This means that the first moves evaluated are likely the best and therefore likely lead to early pruning. This does lead to more worst-case heuristic calls but in reality it cuts runtime

significantly, allowing for depth 3 on an 11x11 board. This early pruning allowed me to expand to depth 3 without any iterative deepening. For moves early in the game it prunes aggressively and runs quickly, affording me more time for later in the game. Near the end of the game, its moves can take longer than 10 seconds, but it almost always wins before time expires.

Challenges:

One 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.

The biggest challenge was optimizing after I had implemented standard alpha-beta pruning. I used a profiler to find that almost all my time was spent in the algorithm checking for consecutive stones, so I focused on decreasing call frequency and increasing speed per call. In the process, I converted my entire program to use Numpy, which ended up quadrupling the runtime. I think this is because all my arrays were fairly small and I did a lot of modification (padding, deleting, etc) rather than matrix math. Numpy made my code much nicer to read but I had to throw it out.

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 become less relevant because the algorithm would be able to see directly to the situations they lead to. However, this means even when the estimated value of a move is very high compared to the others, I still expanded all branches until alpha-beta pruning. If there was an unconnected row of four or five, I didn't want to miss it.

My heuristic is rather arbitrary and could likely be optimized. Instead of worrying too much about it, I spent more time optimizing for depth 3. Almost all the runtime is spent in the heuristic, and I suspect there are more efficient ways to traverse the board for consecutive stones.

I was able to beat the baseline with only depth 3 and alpha-beta pruning, but I only tested for an 11x11 board. As the game progressed, my turns took longer and longer, Without iterative deepening or further pruning based on heuristics and optimization, 11x11 is near the limit of effectiveness and my algorithm ties out on slightly higher sizes. To run at higher sizes, I would need to lower depth and just rely entirely on the heuristic.