MP2 Report

Architecture:

For the Architecture of my AI, I decided to make the minimax algorithm use a tree data structure. It made it very easy to traverse (recursively) through each of the possible board moves, calculate their heuristics if necessary, and move on if not (alpha-beta pruning). In terms of basic functionality, I created a main function that simulated the current state of the game for my AI (i.e. processed the opponent's moves and placed them onto the board, processed it's own moves and placed them onto the board, checked input errors/win conditions, etc.) and handled all of the initial startup flags (-l, -n). When it was the AI's turn, it would simply run the decide_move function with the current board state and the color that the AI was playing as and have its next move returned to it. It would then update main's game board and send the move it made to the other player. Within decide_move, my code would essentially create a new max tree root and run the build_tree function. This would create my search tree of all possible moves (will be talked about in the Search section). It then ran the generate_heuristics function, which would calculate the cost of all the moves in the search tree that it needed to (not all possible move costs were calculated because I used alpha-beta pruning). The best move's index within the root's children would be stored in the root as v_index and the last line of decide_move would return the move that was contained in the root's child at v_index. This move is then parsed by the main function and play continues normally. Some other functions I had were debug_print_options, calc_score, and print_board. Debug_print_options, when enabled, would allow me to see all of the moves within the tree that my decide_move function generated for debugging purposes. Calc_score contained my heuristic function and would calculate a heuristic score for the board and player color passed into it. Print_board was a function that I directly copied from referee.py to use for humans to play against my AI and for debugging purposes (all credit for print_board() goes to Karl Wang and all other 165A TAs that have worked on it in the past).

Search:

My search function went as follows. First, my decide_move function would build the tree, starting from an empty max root, using build_tree. My build_tree function would first generate the different parameters for the next layer based on current depth and original depth (i.e. the node type, player color at that node, and default value for all the children on that layer). It would then iterate through all empty spots on the game board and check to see if there was at least one stone adjacent to the current spot. If there was, it would count that spot as a potential move, create a child node and add it to the root's children, then move on. By the end, the root would have a children array full of potential moves that were adjacent to at least one other stone. I did this to speed up the later calculation of heuristic scores later, because for the most part any moves that were not adjacent to some other stone are going to be less useful than ones adjacent to a stone. While this may have caused some hit in play ability, the time saving aspect was worth it (you reduce your search space by a large amount and it allows you to scale up with bigger boards very easily). This would then repeat itself for all min/max nodes until you reached the lowest depth, effectively giving you a tree full of potential moves within a limited search space for every min/max node. After this tree was generated, the decide_move function would run the generate_heuristics function. This function would recursively move through the tree and keep track of alpha/beta scores for alpha beta pruning. It

iterates through each “current” root node’s children and checks each one to see if it is better than the previous best value for that node (higher value for max, lower value for min). If it is better, it will update the alpha or beta value as well (alpha for max nodes, beta for min nodes). At the end of every check of a child, it will see if current beta <= alpha. If it is, then you can prune the rest of the current root node’s branch and break out of the children iteration. The values that I have been mentioning as “better” or “worse” are the heuristic score values. When generate_heuristics encounters a leaf node, it will use my calc_score function to generate the heuristic score for that leaf node. This value is then propagated up through all the nodes if it is the best value. If a leaf node gets pruned, this heuristic score will never be calculated. Lastly, I must talk about my heuristic score calculator. My calc_score function was very simple, but effective. It would iterate through each spot on the board it was calculating the score for and check 5 spots in a row above, to the right, left, below, and diagonally of the spot. If it saw a five-in-a-row, four-in-a-row, three-in-a-row, or two-in-a-row for either dark or light stones, it would add 1 to the respective bins. These bins represented the frequency of each pattern occurring for both the dark and light stones. At the end, my calc_score function would calculate a weighted score based on the number of occurrences for each pattern. If the player color was dark, it would count dark patterns as positive and light patterns as negative and vise-versa for if the player color was dark. The weights were 50000, 1000, 6, and 2, respectively, for the bins. I wanted to make sure five-in-a-row was a very large number to make sure if the AI had a winning move it would always play it and if the other player had a potential winning move it would avoid/block it. These scores would be returned to the leaf nodes and the move that produced the largest heuristics score at the lowest depth would be chosen.

Challenges:

There were many challenges I faced. The largest was how to write my heuristics function and possible move building function in fast, efficient ways. At first, my searches would take forever and get worse and worse as I expanded the board size because I was using the whole search space for possible moves. Once I changed it to only include possible moves that were adjacent to stones already played on the board, my calculation time decreased significantly per-move and scaled very easily with larger board sizes. My heuristics function posed another challenge. Originally I calculated the heuristics score for all potential moves, then chose the best one. This would also severely affect move calculation times. To solve this, I implemented alpha-beta pruning, which improved my AI’s performance significantly. Originally, it would take 5-10 seconds minimum to decide on a move. Now, it only takes 0.05-1 second to make a move, regardless of board size.

Weaknesses:

One of the biggest weaknesses I think my program has is that it will potentially double-count patterns when it is running the heuristics function. This is because if there is a five-in-a-row that is pointing straight down, for example, the move will be counted once when the calc_score function is at the top stone looking down and twice when it is at the bottom stone looking up. This effectively doubles all scores and technically over-estimates the heuristic function. A couple ways to fix this are first to halve the relative weights you want to account for this and second to implement some sort of tracker that doesn’t count a pattern more than once if all stones have already been used in a previous pattern.