Jun SeungJun Cho, 5774377

Architecture:

I stuffed all of my parts into a single “game” class, which takes the variables “n” (board dimensions) and “l” (AI color) in the constructor. There are 4 public functions: receive move, find move, send move, and print board. Receive move, send move and print board are fairly self-explanatory, and they do not do much else outside of what the function name states. The find move function is the main recursive function that forms the minimax tree, which takes the player number, the current depth, and the number of new spots needed to be considered in the next depth. The reason why I send this information is because the number of points needed to be considered changes depending on the depth, so I thought it would be a good idea to store the new, temporary points to consider in a stack then pop them as needed depending on the number passed through the function. Due to the method I evaluated the board, as well as the nature of my heuristics function, I needed to keep two separate representations of the current game board state, one in which all of the program’s moves are stored as 1’s and the opponent’s are stored as 2’s, as well as the inverse, where the program’s moves are stored as 2’s and the opponent’s are 1’s.

Search:

I think I implemented my search function slightly differently from most of my classmates. While trying to make the code run as fast as possible even before pruning, I had the idea of rather than evaluating the entire board at the final depth, at each level I would calculate the change in score if the stone was placed at each available space (spaces adjacent to already existing stones). This speeds up the board analysis significantly, as now rather than needing to look at the entire board for each possible outcome, I only need to look at a single space per board, cutting down the exponential runtime significantly.

To do this, I made my heuristics function look at the current location (x, y), as well as the 4 spaces to the left and right of the spot in frames of 6 spots at a time. To clarify, if the board had a row that looked like this: _ _ O _ $\boxed{\textbf{X}}$ X _ _ _, where the boxed space is the current location, my heuristics function would first look at “_ _ O _ X X”, then “_ O _ X X _”, then so on, finding which frame had the highest scoring potential. This process was repeated across the four axes (row, column, left and right diagonals), and the four results were summed together to determine the temporary score of that space before future predictions. I stored my heuristic as an unordered map, using the 6-spot frames as keys after a conversion to integers One of my comments in the code next to the heuristic map goes over how I did this in more detail, as well as a link to the google sheet I created to make the process easier, which I used to write down all of the possible and important frames of 6 I needed for my heuristic. (https://docs.google.com/spreadsheets/d/1wyPO3BosZo8A6r7s1XJ5ZhKLDqvBuoCzwra-iT-oZaw/edit?usp=sharing)

Another “optimization” I did with my heuristics function was by using the two separate boards I mentioned earlier. Due to how I had to manually think of combinations for my heuristic, not only did I not want to redo them except just with swapped numbers, I also did not want to make the list of numbers at the top of my code any longer than it already was. So instead, I stored the current state of the game board in two separate vectors as I mentioned earlier. This allowed me to do two things: first, I was able to use the same heuristic numbers for both score calculations for the program as well as the opponent by using the matching board. Secondly, I was able to incorporate a score for blocking the enemy by also analyzing how much score that same spot would give if the opponent were to place their piece there. This can be seen in the heuristic search function.

As for the actual minimax tree, I did something a bit weird with that too. Rather than using the min function every other depth, I instead opted for multiplying the scores from parent and children depths by -1 and using the max function each time. It is still a minimax tree, but it might look a bit strange due to the lack of mins. In fact, this was one of the main reasons why I struggled to implement the alpha-beta pruning for my trees, as I kept confusing myself as well. This part of my code is my least favorite, due to being just confusing without providing any meaningful benefit, while the other parts were important, even if they seem odd.

One interesting part about the minimax tree, however, was my combined use of the temporarily available stack and unordered set. At first, I was not sure how to incorporate adding more spots to check as the depth increased, since adding and removing from the already existing “available” set could remove spots that should still be there after one check. This led me to think about creating a separate unordered set just for the temporarily available coordinates. However, with just an unordered set, it was impossible to remove just the last x number of spots, which would have been helpful so that I would not have to keep track of which coordinates were added at a specific depth. This led me to consider using a stack of temporary coordinates that I could simply pop the last x items. While I could have also used the normal set, I wanted to use the unordered set due to the quicker average case insertion, access, and removal of nodes, so I ended up using the combination of both the unordered set and the stack for my purposes.

Challenges:

The biggest challenge to me was the restriction on the time. In order to not time myself out and lose by default, I tried my absolute hardest to come up with an implementation that would be as time efficient as I could possibly think of, which was how I ended up with the unordered map (with manual, pre-entered data) format for my actual heuristic, and was why I changed up my board evaluation function from the typical full board evaluation to the single point based, change in score type evaluation. In addition to this, I tried to use data structures with O(1) average case for many operations, hence my extensive use of unordered sets, maps, and the stack. All of my efforts were not in vain, however, as my program ended up becoming fast enough to evaluate fairly quickly at a depth of 3 and 4, even without alpha-beta pruning.

Weaknesses:

There are definite clear weaknesses of my system. Aside from clear miscalculations due to my heuristic weights being not ideal, there is also the inability to check for double open 3’s (board with two lines of three unblocked stones), which is the easiest way to win Gomoku. These can be fixed by tweaking the heuristic model more, as well as including flags to indicate win or loss conditions more effectively.