Andrew Kraft

andrewlkraft@ucsb.edu

3410933

MP2 Report

Architecture:

My implementation includes a gomoku class which stores the size of the board, n, which player goes first, l, and the minimax depth, m. It represents the board as a 1D array of chars, accessing coordinates (x, y) as board[n*y+x]. I represented moves as std::pair<int, int> for simplicity. I added public functions to process a player move and a computer move, as well as to check the state of the game (ie. ongoing, win, loss, tie).

Search:

My algorithm uses minimax with alpha-beta pruning to search for the best move considering the opponent's move. To lessen the branching factor at each minimax step, I created a vector of values called successorMoves, and everytime I made a move on the board, inserted only the nodes adjacent to that move as possible successors. Though this may not be a perfect approach, for the most part relevant moves are adjacent to already existing ones, so the value from cutting out so many moves outweighs the efficiency drop. For the evaluation function, I used a heuristic which scans through all rows, columns, and applicable diagonals on the board and scores each cluster of white or black stones based on 1) how many stones are in the cluster, 2) how many open ends each stone has and 3) who plays next. To increase the efficiency of the heuristic function, I created an escape so that if the program has already found a valuable cluster after any loop, it returns immediately to prevent the other loops from running. Finally, I used -O3 when compiling the program, as suggested in piazza as a way to squeeze some extra efficiency out of the program at compile time.

Challenges:

The biggest challenge I faced with this project was creating an efficient and good evaluation function. My minimax function would take 15-20 seconds each turn on anything greater than depth 2. I spent a lot of time diagnosing this issue, and decided the eval function was at fault. It was inefficient because it looped through the board 4 times, checking each row, diagonal, and column. To improve on this, I tried an eval function which took the last move played, as well as the next player, and only evaluated locally. This, however, had problems when there was a winning move outside the scope of the computer's "vision". It also made it difficult to evaluate both performance scores, because the function only had the last move for one player, so it wouldn't know where to search to get the score of the other player. I toyed with having a running eval score for each row, column, and diagonal for each player, and updating it after each move, but to me this seemed overcomplicated, so I ended up going with the approach which loops through the board 4 times. I added an escape though, incase I had already found a high-value group of stones.

Weaknesses:

Of the weaknesses in this project, the eval function is certainly the weakest part, and the part which could be improved most. Over the course of the project, finding an efficient and good eval function was the most challenging aspect, because I had a hard time visualizing how to evaluate a function while not being able to look at the entire board at once. For the sake of simplicity, my function evaluates each group of stones in a row, column or diagonal as independent, so if there is a group with a gap, it will treat it as 2 separate groups. This can be harmful, especially when the groups are a 2 and a 1, because this is considered a threat in gomoku but my function doesn't recognize it as such. If I were to try to overcome this, I might make a gap variable, and consider a cluster as connected if the gap is less than 2. Another shortcoming of my eval is that it doesn't take into account how much space there is for a group when calculating the score. If, for example, it saw 3 black stones with one white stone on one end and none on the other, it would call it a 1 open ended 3 cluster, and rate it highly, even though there might not be enough room for that cluster to grow into a 5 in a row, essentially making it useless. I could overcome this by keeping track of the space that group has and checking if it is less than five before scoring the group. The solutions I propose to these problems have their own flaws, however, which is why I chose not to implement them.