Derek Lee Artificial Intelligence Fall 2020

Professor Sable Project #1

Settings

A custom board can be specified from a text file by accessing the Settings menu after running the program. The text file format for loading in a custom board is as follows:

1. Each piece is separated by any default delimiter for cin (e.g. space, newline, etc.)
2. Pieces are represented as:
    a. 1 – White King
    b. 2 – White Man
    c. 3 – Red King
    d. 4 – Red Man

It is also possible to manually add and remove pieces using the program, one at a time.

The settings for each player can also be changed through the Settings menu. This can change whether the computer plays itself and which color has the first move.

Heuristic

My heuristic favors men closer to the final row, where they would be crowned. When one player has more pieces and the other player has a king in a double corner, my heuristic favors specific squares. These squares encourage the program to narrow its search, allowing it to eventually force the king out of the double corner. My heuristic also encourages a king to get closer to an enemy piece by using a factorial-like function, which adds numbers instead of multiplying them. In theory, this should reward advancing pieces that are further away. My heuristic also favors reducing the number of pieces on the board for the player in the lead. The

goal is to implicitly encourage trading pieces when a player has an advantage. My heuristic uses a random number generator to add some randomness to game states that would otherwise be equivalent. This is in addition to my minimax search, which randomly chooses between equivalent states.

Board Class

My implementation of my board class relies on pointers to my piece class. I create instances of my piece class at the start of the game and store pointers to those pieces in my board class. I use a single instance of an "empty" piece and a single instance of a "filler" piece to represent multiple locations in the 2d board array. Unlike regular pieces, empty and filler pieces do not need to store their location, meaning the same piece can be reused for every location that requires an empty or a filler piece.

Minimax Search

I did not use my Terminal Test function in my minimax search. My heuristic returns a very large number (much larger than a normal score) when a terminal state is reached, so that should be sufficient for the search. My minimax search calculates the time elapsed every time it is called, so once the remaining time reaches a threshold, I abort the search and use the results from the previous depth. Once a definite win is detected, my program stops searching. However, my program continues searching even once a definite loss is detected. This is useful for playing against humans. If the program stops searching early, it can reveal that there is a definite win, even if the human player did not realize a definite win was possible. In addition, when a terminal state is encountered, my minimax search adds a depth factor to the terminal score, which

penalizes the score based on depth. This results in the winning player wanting to win faster and the losing player wanting to lose slower.

In my minimax function, I use recursion, passing by reference my board object. In order to evaluate a set of moves, I would have to "play" each move and calculate the score of the resulting board using my heuristic. "Playing" each move involves my moveResult function, which directly alters the board object. Because I passed my board object by reference, changing the board object at depth D will also change the previous board objects from depth 0 to D-1. To deal with this, I make a copy of the board class at the beginning of my minimax function, which is why I decided to use pointers to my piece class, rather than using instances of my piece class; copying pointers is faster than copying the class. This ensures that the data structures (unordered_set and list) I use in my board object will not be affected. However, the data structures contain pointers to pieces, meaning a change to the pieces at depth D will also affect the pieces from depth 0 to D-1. To handle this, I created the isolateBoard function, which will make copies of all pieces that are potentially affected by a move. This is significantly less than copying the entire board (12 for a move and 16 for a jump, compared to 64 pieces in the entire board). Then, pointers to the original pieces in the object are replaced with pointers to the copied pieces. Thus, any change made to the board object will not affect previous board objects.