Project 4

CS 4200 - Artificial Intelligence

Instructor: Dominick A. Atanasio

Computer Science Department

California State Polytechnic University, Pomona

Report Submitted On: 03-MAY-2020

Report Submitted By:

Damian Ugalde

Evaluation Function Strategy

The evaluation function I chose has 3 variables. It takes into account the amount of moves that the computer has available, the amount of moves the opponent has available, and where in the board is the computer positioned.

It is more beneficial for the player to be closer towards the center of the board. From this matrix, a higher score is given to the player if it is closer to the center.

$$\begin{bmatrix} -10 & 1 & 2 & 3 & 3 & 2 & 1 & -5 \\ 0 & 1 & 2 & 4 & 4 & 2 & 1 & 0 \\ 1 & 2 & 4 & 5 & 5 & 4 & 2 & 1 \\ 2 & 4 & 5 & 6 & 6 & 5 & 4 & 2 \\ 2 & 4 & 5 & 6 & 6 & 5 & 4 & 2 \\ 1 & 2 & 4 & 5 & 5 & 4 & 2 & 1 \\ 0 & 1 & 2 & 4 & 4 & 2 & 1 & 0 \\ -10 & 1 & 2 & 3 & 3 & 2 & 1 & -5 \end{bmatrix}$$

The formula takes how many moves the computer has and multiplies it by four. Then it subtracts the amount of moves that the opponent has multiplied by two, and adds the weight of the position of the computer to the board. The formula is:

$$value = (4(computer\ moves) - 2(opponent\ moves)) + position\ weight$$

This strategy leads to a more defensive agent. It has a greater priority to maximize the amount of moves it has available. As a second priority, it tries to minimize the amount of moves that the opponent has. As a final priority, it tries to move closer to the center of the board and avoid the edges. This gives the computer more options for movement later on, and it also helps the computer not be trapped by the edges of the board.

Another technique that could be used is an aggressive one. The computer prioritizes lowering the amount of moves that the other player can make. This can be done by multiplying the amount of moves that the opponent can make by a greater negative weight. Also, the computer can prioritize moves that the opposing player can make as well. It can also determine a linear distance between its current position and the opposing player, always trying to get close to the opponent and boxing them in.

Problems encountered and steps to resolve

The biggest problem I encountered was trying to make the alpha-beta pruning algorithm to work. I couldn't figure out how to distinguish when I had to return a utility or evaluation value and when to return a move. The way I solved this was by creating a private auxiliary class that contained the move and the value as a tuple. That way, I can return both.

Another problem I encountered was trying to find the available moves in an efficient manner. The way I solved it was by checking all the directions (up, down, left, right, up-left, down-left, up-right, and down-right), starting from the spaces adjacent to the player's position. I would check if the space was used or empty. If the space is used, I would set a flag to stop checking that flag. Otherwise, add the move to the list. I would keep going until there were no more paths to check or you reach the end of the board.

I also needed a quick way to determine if a move was valid. I solved it by creating multipliers for the row and columns. If the rows were equal, the multiplier for the row was set to 0. If the row we are trying to move was greater than the position, it was set to 1. Otherwise, it was set to -1. The same thing was done for the columns. Then a while loop will keep searching, starting from the position of the player, until you find an obstacle on the way or reach the end of the board (both indicating the move was invalid). If you reached the position you are trying to move, then it is a valid move. This turned out to be a very efficient solution.