Angello Polo Espinoza and Brian Russell

HW2 Report

Point

The Point class represents one of the 64 possible positions on the board. The most important properties are:

state- an int which is equal to:

- 0 if the space is blank
- -1 if the space is occupied by an O (player 2)
- 1 if the space is occupied by an X (player 1)

num_wins - a counter of the number of wins in a given point.

num_games - a counter of the number of games in which a given point on the board is played.

i_pos, j_pos, k_pos - these are ints which represent the x, y, and z position of the given point on the board

getUtilValue() is a method that returns a double representing the current utility value of the given point. This utility value is determined by num_wins/(num_games), with some minor extra logic added to avoid dividing by one in the initial calls of the method

Board3D

The board class contains a 4x4x4 array of 64 points. The only property of this class is:

board - a 4x4x4 array containing 64 points.

When a Board3D object is constructed, **board** is initialized with 64 points, each with value 0.

The most important methods for this class are:

updateBoard() - called every time a player makes a move on the board.

Takes in a boolean isP1 representing the player making the move and three ints representing the x, y, and z values of the point in which the move is being made. Updates the corresponding point's state, and then calls **checkIfWon()** and if no one has won, **checkIfOverNoWin()**. Returns an int **wonValue** representing the current state of the game. If **wonValue** is:

- -1: p2(O's) has won the game
- 0: the game is still ongoing
- 1: p1(X's) has won the game
- 2: the game has ended in a tie

checkIfWon() - called each time the board is changed by **updateBoard()**. Checks to see if the game has been won. Evaluates the 3 "straight" lines of 4 (constant y and z, constant x and z, constant x and y) in which the game can be won for that specific point, as well as the 6

diagonals in which that specific point can win. Then it evaluates the 4 “universal” diagonals on which the game can be won, each of which runs from one corner to its “opposite” corner (opposite meaning running through the cube itself with varying x, y, and z). Checking these 4 universal diagonals as individual cases rather than on a point by point basis meant that we only needed a single for loop to test for wins. If one or more of these tests evaluates to true after all 4 iterations of the for loop, the winner’s id (-1 for p2, 1 for p1) is returned, otherwise 0 is returned.

> **checkIfOverNoWin()** - returns false if any point on the board is blank (has a state value of 0).

The other methods of the Board3D class are all fairly self-explanatory.

Solver

The main method of our **Solver** class reads in the 3 integer numbers of trials using a scanner and then initializes a new Board3D object named **board3D.** Then, a for loop is used to play n games, where n is the number of trials provided in the first value. Moves are chosen using the **nextMoveExploration()** method of the Solver class.

nextMoveExploration() - takes in the Board3D object representing the current position of points and two ints, totalNumTrials and currentNumTrials. totalNumTrials is equal to n as discussed in the previous paragraph. currentNumTrials is equal to the current number of trials completed. These two numbers are used to influence the choice between exploration and exploitation, as the double **exploitationLikelihood** is equal to the **currentNumTrials**/**totalNumTrials**, or the proportion of the overall results that have been run. A random number is then generated and compared to **exploitationLikelihood** so that over time, exploitation is preferred to exploration.

Exploitation comes from the **nextMoveExploitation()** class, which is called when the randomly generated double from 0.0 to 1.0 is less than **exploitationLikelihood.** **nextMoveExploitation()** returns the blank point with the highest utility value, and in the case of ties, randomly selects a point from a list of all points with the highest utility value.

Exploration comes in the form of a randomly selected blank point being chosen by the player if the randomly generated double is greater than **exploitationLikelihood**. When this else statement is reached, a random point is chosen from a list of all blank points on the board and then this point is returned.

When each game finishes, utility values are updated. For each point played by the winning player in a given game, the point’s **num_wins** property and the **num_games** property

are both incremented by 1. For each point played by the losing player in a given game, only the point's **num_games** property is incremented by 1. In the case of draws, only the **num_games** property is increased. Points not played by either player do not have their **num_games** or **num_wins** properties incremented.

After the requested number of trials are completed, the utility values of each point are printed to the console. The above process is repeated for the other two trial values provided.

Contributions:

While we both worked on all parts of the assignment together, Brian was largely responsible for the Board3D class while Angello was largely responsible for the Point class. Work was more evenly distributed on the Solver class, although Angello did contribute more to this class than Brian did. The report and readme were both collaborative efforts in which we contributed roughly equal amounts of work.