University of British Columbia Okanagan

COSC 322

2020 Winter Term 2

Professor Gao Yong

Project Topic Number: 17

Game of the Amazons

Group Members:

Ruochan Wang 17512740

Siying Wu 60081734

Calvin Qu 60525995

Zhuoxuan Jiang 41637208

Implementation

COSC322Test.java

COSC322Test.java is the main class in the program. Firstly, a chess board is built to save the update from the client and give the AI's own move to the client. 0 is defined as an empty spot; 1 is black; 2 is white; and -1 represents the arrows for both sides. The board that we built in our program is upside down compared to the board in the client and also we count the first point in x-axis as 0, but the client board counts the first point as 1. For example, the point (1,1) in our board is equal to the point (9,2) Then the main method sends the player name to the server and checks the game environment if both players join the game room.

The method COSC322Test and onLogin are used to make a GUI-based player, create an instance of BaseGameGUI, and implement the method getGameGUI() accordingly.

The method handleGameMessage receives a message from GameClient when it receives a game-related message from the server which includes the opponent's move. The first step AmazonsGameMessage.GAME_STATE_BOARD will print out the room information in the console and update the GUI in the game client. Afterward, AmazonsGameMessage.GAME_ACTION_START will print out three informations which includes the state of the game and the ID's for black and white players. Therefore, we are able to receive the message to determine whether we are white or black. At the end, AmazonsGameMessage.GAME_ACTION_MOVE will print out the opponent's position which include the current position, next position and arrow's position. As a result, we define three ArrayList to store those positions and update our GUI by these messages. by calling

updateBoard, we update the opponent’s move onto our local board. Beside, we call the method which will send our move message to the client.

The method updateBoard receives three arrayLists of opponent’s position or our position which include the current position, next position and arrow’s position. In addition, the parameter receives an integer variable to determine whether it is the opponent's or our coordinates. If the method receives the position of the opponent, Since our board is upside down to the client board. To update the opponent's move on the local board, the method will convert the coordinates from the client board to the corresponding coordinates on the local board.

The method move receives the local board and an integer variable which presents our queen’s type. By calling the findTheBestMove method, the ArrayList<ArrayList<Integer>> stores three coordinates which are the best move. After the conversion of the coordinates, the method sendMoveMessage sends the coordinates to the game client and the method updateGameState updates the GUI. At the end, the method updateBoard is called to update the local board.

Move.java

In move class, nine methods are constructed which is used to provide the next coordinates of the queens or arrows after moving towards one of the nine different directions(goUp, goDown, goLeft, goRight, goUpLeft, goUpRight, goDownLeft and goDownRight).

For instance, the goUp() method needs the input of integers x and y, which represent the coordinates of the target object, such as queen or an arrow. In the method, an array named goUp is created, and the array holds two values which are the x and y coordinates. Once goUp() is called, the x-coordinate would plus one and y-coordinate keeps the same, and then it would return the new array as the new coordinates of the object. Similar operations for other eight directions movements.

MinMax.java

The findBestMove method in this class. The parameters of findBestMove receive the current board and the queen's type which is an integer. If type equals 1, current queens are white, else the queens are black. The currentPosition and opponentPosition methods are called in the findBestMove method. The parameters of findavailableMoves and findavailableArrows require the coordinates of the target. They return an arrayList which stores all available coordinates for moving. In addition, three for-loops iterate those coordinates to find the best move.

The first for-loop finds the available moves for each queen by calling the method findavailableMoves. The second for loop which is in the first for-loop finds the available arrows' coordinates for each available move by calling method findavailableArrows. The third for-loop finds the utility of each arrow.

We define the utility as the total number of blocks on the graph that are available to move for four queens. There are two situations, the first is to calculate the utility of our queens, the utility is higher when the number of available moves are higher. The second is to calculate

the utility of opponent queens. On the other hand, the utility is higher when the number of available moves are lower.

Based on this situation, we calculate the utility of opponent queens at the beginning of the game. After 10 rounds of the game(number of rounds can be changed), we calculate the utility of opponent queens.

The findTotal method returns the utility based on the current map and the queen's type. In the third for-loop, we mark (-1) each arrow on the map each time. The findTotal method is called to get the utility based on the marked map. By comparing each utility of the arrow, we are able to find the best positions which have the highest/lowest utility of the next round.

findavailableMoves will return the best move based on the above algorithm. It returns an ArrayList<ArrayList<Integer>> stores the original queen's position which will be moved, the new position which will be moved to, and the new arrow position. Each of the positions is stored in an ArrayList<Integer>.

Utility.java

In Utility class, there are five main methods. findTotal, findavailableArrows, findavailableMoves, currentPosition, and opponentPosition.

FindTotal method calculates the total number of blocks on the graph that are available to move for four queens. The color of queens is determined by the input type. If type equals 1, queens are white, else queens are black. For each direction, there is a nested loop of two

levels. The outer loop iterates the four friend queens on the map and the inner loop iterates through the nine blocks in the corresponding direction (except the one where the queen is positioned). Once there is an arrowed block on the path, the iteration breaks and records the number of blocks it passes into a global variable. Eventually, the available blocks in eight directions add up to the same global integer variable.

With inputting coordinates of a queen and the whole map, there are eight loops for each direction. The loop iterates through nine blocks on the corresponding path and stores the coordinates of the blocks it passes into an ArrayList of integers. Then findavailableMoves outputs the coordinates of all the available blocks where the current queen is able to move by returning the Arraylist of Integers. In order to easily store 2d coordinates into ArrayLists, we convert (x,y) into an integer by x*10+y.

Since we know that the positions where queens are able to move are also the positions where queens are able to fire an arrow. Thus, findavailableArrows method explicitly invokes findavailableMoves method.

CurrentPosition method returns the coordinates of the four friend queens on the graph by iterating through the whole map to find the friend queens according to the input type.

CurrentOpponent method returns the coordinates of the four opponent queens on the graph by iterating through the whole map to find the opponent queens reversely according to the input type.

Techniques

The game-tree search used in our project is based on MinMax. It’s an algorithm that assumes the opponent’s each step will minimize our queens’ chance of winning, then we should calculate a way to act that will maximize our chance of winning. In other words, MinMax always finds an optimized way to react in the worst condition.

In our implementation, due to some limitations, our algorithm went through a tree with only depth of 2. When it is our queen's turn to move, the algorithm goes through each available place on the board and returns the number of that. After that, we compare the numbers to get which queen gets maximized steps to move, and how many following steps after each of those steps. According to that, one optimized queen moving is generated. After the queen moves, the algorithm goes through the whole board to provide a list of coordinates that an arrow can be set. Then the same way is used to get the number of steps opponents can move after each possible arrow setting, the one getting minimized number would be where the arrow goes. At the same time, if there are a bunch of coordinates that get the same minimized number, we choose the one that is closest to the opponent's queens to set the arrow.