CS 657 Intelligent Systems

Assignment 1 – Expert System

Jacob Byerline
Red ID: 821301215

For questions, please reach out to jbyerline@gmail.com

Problem Description:

The problem at hand is to develop an intelligent system that can drive a simulated car which can detect obstacles in its path. Obstacles are represented as 1's in a 2D array. The car can detect obstacles in front of itself as well as in + or - 45deg increments of its current view. The goal is to get the car from a given starting position through a randomly generated map of obstacles without crashing to the winning position in as short of time as possible. The car is not aware of the map, it learns the obstacles' location as it approaches new obstacles.

My Approach:

I decided to create this program in Java. At a high level, I broke this program down into 4 distinct classes. They are as follows:

1. Main
2. Coordinate
3. Grid
4. Car

Starting from the top of the class list, the Main class serves as an entry point of this program. It takes in user input via the Scanner class to build the map with the specified parameters. The class then parses this entered data, catching any errors. We instantiate a game grid object and randomly generate the obstacles based on the percentage the user specified. Finally, the class prints out the answer key for comparison later and then triggers the Car class to begin the expert intelligent search.

The next class is the Coordinate class. This class serves as a template for an (x,y) coordinate pair. Its sole job is to make the passing of coordinates between classes and methods simpler.

Next, we have the Grid class. This class defines a template for the game grid. It is used for both the master key grid as well as the car's memory grid. It contains several methods for related tasks such as randomly populating the grid and printing out the grid in a usable fashion. The randomize grid method accepts an integer that represents the percentage of the grid we wish to block out with 1's. This is accomplished by finding the percentage given of the number of spots on the grid. From there we fill an ArrayList the size of the grid with the calculated number of 1's needed. We fill the rest of the list with 0's. Then we shuffle the list with the Collections class. Finally, we store each of the values in the list into the 2D grid, thus producing a dynamic and randomized grid.

Finally, we have the Car class which includes all of the relevant actions pertaining to the car. These include runCourse(), move(), rotateClockwise(), checkCurrentView(), checkPossibleMovies(), and determineStepsToGetToWinningPosition(). These methods are used to break our expert logic into smaller sections for easier understanding. Starting from the simplest method and moving onward.

The move() method accepts a Coordinate and simply updates the cars memory with its new position, as well as appending the new position to a list of all the past moves. It will also update the cars memory, replacing its previous position with the number 3 to signify that it has already been to this location.

The rotateClockwise() method simply rotates the direction of the car by 90deg in a clockwise motion. This is used when the car cannot find any viable paths within its view so it must rotate to try to find an alternative.

The determineStepsToGetToWinningPosition() method does exactly what the title says. When the car finds the winning position, identified by a 9, it may not be immediately next to the car's current position. Because of this, we need to calculate the moves needed for the car to get to the winning position.

The checkCurrentView() method scans in front of the car and at + or − 45deg of the cars front to look for 0's or 1's. It then updates the car's memory with the new values it has discovered. If it encounters a 9, it will set the foundWinningPosition flag to true and store the winning position.

The checkPossibleMovies() method runs after the checkCurrentView() method has updated the car's memory. This determines the best place to go given the car's current position. By default, it will go to the first 0 it finds unless it cannot find one. At which point it will begin to backtrack its steps looking for other 0's it has not yet visited. If the car cannot find anymore 0's and has not found the 9, it will exit the program with a message stating the winning position cannot be found.

Finally, the runCourse() method puts it all together. This method loops and arbitrarily high number of times. We use the grid size cubed here but it doesn't matter as long as it is a large number. For each iteration we call checkCurrentView() to check what is in front of us. We then calculate the possible moves around us with checkPossibleMovies(). If there are no possible moves, we rotate and check again got possible moves. After 4 rotations with no possible moves, we can exit and determine the course unsolvable. Otherwise, if we did find some potential moves, we first check if we found the winning location. If so we call determineStepsToGetToWinningPosition() and then move to the ending location. Otherwise, we move to the first position in the possible move list.

Notable Additions:

The checkPossibleMoves() list uses some logic to determine the next best move. It does this by avoiding repeated moves and backtracking to find the next unvisited position. This provides a decent algorithm for randomly visiting nodes without too much overhead.

Some minor calculations I performed:

Running Instructions:

This program was written with Java 14.0.1. The executable jar file can be run using the command `java –jar "Program 1.jar"` on MacOS. Please keep in mind, to execute a jar file, you need to have the same version of Java installed on your system. Otherwise, you can compile the source code using your local java by opening the code in and IDE and executing the main method.

Once the program has started you will be prompted to ender a set of parameters about the car and the grid. They will come as follows:

1. Enter the dimensions of the grid. This is an integer like 10. This would produce a 10x10 grid.
2. You will enter the car's starting location, an example being 5,2. Be sure to use this format, including the comma and ensure the coordinate given is within the bounds of the specified grid.
3. You will next enter the ending location in a similar fashion to the start.
4. Next you will enter an integer which will be the percentage of the grid filled with obstacles.
5. Finally, you will enter an integer to represent the starting direction that the car will be facing. 1 – North, 2 – East, 3 – South, and 4 – West

The program will then print to the console the generated key map. It will then print out the car's memory map upon completion of the course.

As a reminder, the key to the grid numbers is as follows:

1 - Blockade
2 - Car's current position
3 - Block's car has visited
7 - Blocks unknown to car's memory
9 - Winning block