Readme File – C Coursework

Commands required to make code function:

- gcc -o maze mazesave.c graphics.c
- ./maze | java -jar drawapp-2.0.jar

Code description:

The primary purpose of the background of the Drawapp in this project is to showcase a maze, which has dimensions of 9x9 with a boundary around it. The maze is represented by a 2D array (maze[11][11]) which holds each coordinate in the maze as a square with dimensions of 30 by 30 pixels, and is signified by a number – 0 holds the white boxes that can be traversed through, 1 holds the walls and 2 holds the finishing point.

The triangular robot which is travelling through the maze has a height of 20 pixels, and is represented by a struct (struct robot) which has the following parameters – robotX[3] which holds the x-coordinates of the 3 vertices of the robot, robotY[3] which holds the y-coordinates, and Directionrb which is a char holding the cardinal direction in which the robot is facing. There are some functions that control the basic movement of the struct: moveForward() in which the robot moves one box forward by changing the individual coordinates without affecting the direction, and turnLeft() and turnRight() which turn the robot in the respective direction by mathematically transforming the coordinates and changing the direction. These functions, and most others, are accessed by a pointer to the robot (struct robot *r1).

Other functions are used to determine the conditions wherein the robot can move and stop moving. canMoveForward() analyzes the block in front by finding the value of the robots current maze position mathematically and its direction, then allows it to move if the block in front is 0 or 2, but not if it is 1. atMarker() iterates through the 2D array to locate the grid coordinates of the point which is gray, and compares it to the current grid coordinates of the robot to see if they are numerically equivalent.

There are numerous search algorithm functions used, the first of which is hardcode() which only works for one preset maze – this is solved manually by placing all the necessary move commands in order until the finish is reached. The second one is basicsearch(), in which the robot keeps following the left wall until it either reaches a dead end, where it turns around and keeps following the left wall, or turns right and continues searching – this satisfies Task 3

However, to increase the complexity and thus the efficiency, and to make this work with any generated maze (Task 5), a more complex machine learning-based search algorithm had to be used and Depth First Search was selected in the depthfirst() function. This function requires a stack, which was implemented using an array – dfs_stack[83]. Stack related functions were used, like dfspop() which removes the last entered element into the stack (Last in First out), dfspush() which adds an element onto the stack, and isInStack() which checks if a particular element is in the stack by iterating through it. Depth first search works by iterating through each node of the maze (i.e., box in the 2D array), and pushing that node into the stack, until it reaches a dead end. This was done using a recursive function that calls itself and checks whether the robot canMoveForward() and is not atMarker(). Then, when it cannot move forward, it turns in either direction and checks if it canMoveForward, and moves if it can. Additionally, at every point the isInStack() function is used to check if the point in front is not in the stack, because if it is then the robot is moving in a loop. When the robot cannot move in any direction (i.e. dead end), it starts to backtrack – it pops its current cell from the stack, moves forward, checks if it canMoveForward() and the point is front is not in the stack, then turns in both directions and performs the same check. If there is no such direction, then it pops, moves in the direction of the next cell in the stack, and continues backtracking. This process continues until the marker is reached – and the final stack shows all the cells that make up the final path

Data conversion needs to be undergone to convert the (x,y) coordinates of the cell in the maze to the single int to be stored in the stack. Hence, a pairing algorithm was undergone in the encode_coords() function and the 2 integers were combined by multiplying each one with a relatively large prime number – thus having no chance of an overlap. The function was (47y + 89x) – and the number is decoded to its original coordinates in decode_coords() such that the path can be displayed after execution if necessary.

After exploring and entire section and backtracking out of it, the robot then goes on to explore the rest of another section, after which it starts backtracking. During this process, when it encounters the previous section it explores, it notices that it is not in the stack and starts to explore it instead of backtracking further. This causes it to enter an infinite loop. The solution here is to create a stack that will house the coordinates of the entry of every visited section of the maze, and while backtracking from the next section, if there is a point where it can turn and it isn't in dfs_stack, it will be checked if it is in the visited stack and if it is, backtracking will continue The entry of a section is found by seeing if it can turn in both directions (left and right) when backtracking. If yes, then make it turn back, add that point to the visited stack, and make it turn back and move forward again, then continue. This solution has not been flawlessly implemented yet and does give infinite loops and faults in some scenarios