Plot and Navigate a Virtual Maze

I. Definition

Project Overview

This project takes inspiration from Micromouse competitions, wherein a robot mouse tries to plot a path from a corner of the maze to its center.
This project requires creating functions to control a virtual robot to navigate a virtual maze. A simplified model of the world is provided along with specifications for the maze and robot; the goal is to obtain the fastest times possible in a series of test mazes.

Problem Statement

On each maze, the robot must complete two runs. In the first run, the robot is allowed to freely roam the maze to build a map of the maze. It must enter the goal room at some point during its exploration, but is free to continue exploring the maze after finding the goal. After entering the goal room, the robot may choose to end its exploration at any time. The robot is then moved back to the starting position and orientation for its second run. Its objective now is to go from the start position to the goal room in the fastest time possible.
My strategy for solving the problem is first to start with understanding the provided code, then come up with an efficient algorithm to map out the maze, and finally use a searching algorithm to search for an optimal path of the mapped area.
An expected solution could include a mapping algorithm that utilises the dimension of maze and the distance between the current robot location and the goal to decide which cell for the robot to move into next; and a search algorithm like A* to search for the best path after mapping the maze.

Metrics

The robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. A maximum of one thousand time steps are allotted to complete both runs for a single maze. This metric works fine for this problem.The score for the first run is a discounted number of time steps because the robot needs to explore the maze thoroughly enough to map out the area required to search for an efficient path to the goal, and that takes a few hundred time steps to do so.If we take the whole time steps for mapping as the score, then the score for the second run will mean not much for the metric. The score for executing the second run should be the concrete number of time steps it takes because combined with the discounted first run score, we get a better measurement of the robot's performance than simply counting all the time steps the robot has taken.

II. Analysis

Data Exploration

The robot can be considered to rest in the center of the square it is currently located in, and points in one of the cardinal directions of the maze.

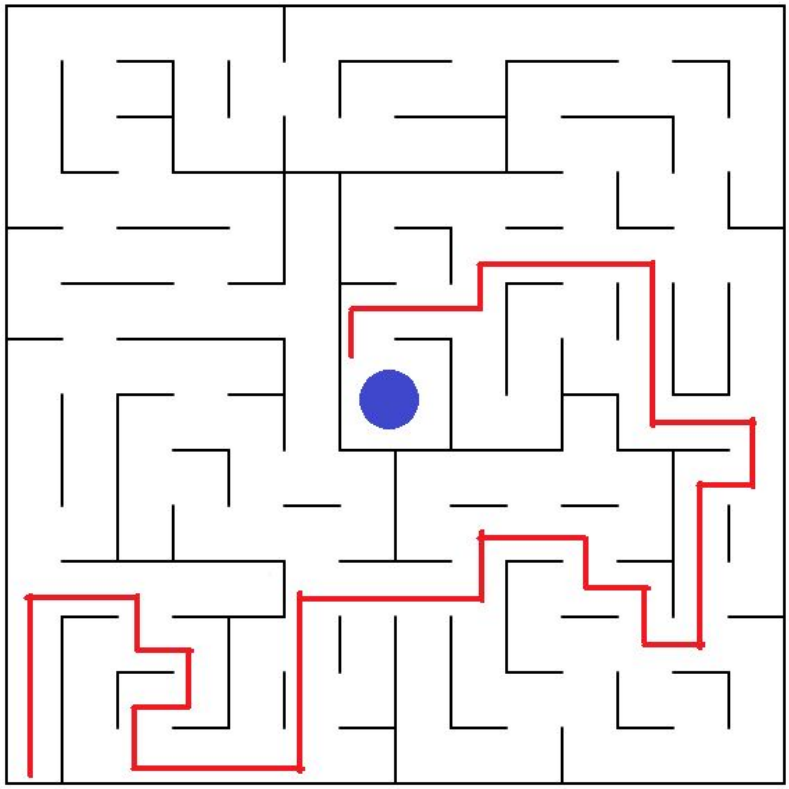
The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor; for example, in its starting position, the robot's left and right sensors will state that there are no open squares in those directions and at least one square towards its front.

On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units. It is assumed that the robot's turning and movement is perfect.

If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robot's new location and/or orientation to start the next time unit.
More technically, at the start of a time step the robot will receive sensor readings as a list of three numbers indicating the number of open squares in front of the left, center, and right sensors (in that order) to its "next_move" function. The "next_move" function must then return two values indicating the robot's rotation and movement on that timestep.

Rotation is expected to be an integer taking one of three values: -90, 90, or 0, indicating a counterclockwise, clockwise, or no rotation, respectively. Movement follows rotation, and is expected to be an integer in the range [-3, 3] inclusive. The robot will attempt to move that many squares forward (positive) or backwards (negative), stopping movement if it encounters a wall.

Exploratory Visualization



Maze 02 (14 by 14)

The red path in the above image represents a possible solution of the problem. It takes 43 single steps for the robot to reach the goal. But the robot can take 3 steps at a time so the total steps of this possible solution is 25.

Algorithms and Techniques

In the first run, we map the maze by using an algorithm that combines the number of times the robot has been in the current cell and a heuristic that calculates the manhattan distance between the current cell and the center of the goal area. This combination is used as an evaluation method to decide the next cell to be mapped.

In the second run, we use the A* algorithm to search the mapped area for the goal and return the moves that the algorithm finds.The A* algorithm works fine for this project because we can utilize the dimension of the maze and the robot's location to inform the robot which move it should take next. Using this information will make the robot perform better than simply choosing the shortest path the robot currently has traverse.
A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost still to go to the goal. Specifically, A* selects the path that minimizes:

f(n) = g(n) + h(n)

where n is the last cell on the path, g(n) is the cost of the path from the start cell to n, and h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal.

Here the heuristic is the manhattan distance between the current cell and the center of the goal area: h(n) = dis_x + dis_y (dis_x is the distance between the current cell and the center of the goal area along the x axis, dis_y is the one along the y axis).

Benchmark

the optimal moves for the maze_01, maze_02, maze_03 (when the robot can makes up to 3 steps each time) are 17, 23, and 25 respectively. This only happens if the mapped mazes contain the optimal paths. if we assume that in the first run, the robot maps out the whole mazes, it at least takes N*N (N is the dimension of the maze)steps to do so. because the robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run, the scores of the best-case scenarios for the 3 mazes are 21.8, 29.5 and 33.5. these scores will be the benchmark used to examine how well the algorithm performs.

III. Methodology

Data Preprocessing

There is no data preprocessing needed in this project , because the sensor specification and environment designs are provided.

Implementation

there are a few class variables that need to be initiated in the implementation, some notable ones are:
self.map: a matrix that contains the wall numbers describing which edges of the cell are open to movement.

self.count: a matrix that counts the number of times the robot has been in this particular cell when mapping.

self.dir_sensors: a dictionary which map the heading of the current cell to the directions of the sensors data.

self.dir_int: a dictionary which map the directions of the sensors data to the values of each side of the wall.

self.policy: after performing the A* search algorithm, this matrix stores the policy(the move) of every cell in the maze.

the functions that got implemented are:
map_maze(self, sensors) : this function map out the maze in the first run each step at a time.

```
def map_maze(self, sensors):
  #add one count to the current location
  x = self.location[0]
  y = self.location[1]
  self.count[x][y] += 1

  #compute the integer that represents the walls of the current cell
  if self.map[x][y] == -1:
     wall = 0
     for i in range(len(sensors)):
        if sensors[i] != 0:
           key = self.dir_sensors[self.heading][i]
           wall += self.dir_int[key]
     if self.location != [0 ,0]:
        wall += self.dir_int[self.dir_reverse[self.heading]]
     self.map[x][y] = wall
```

this function starts with getting the axises of the current location and add one count to the cell's counter.
then it compute the integer that represents the walls of the current cell.

```
#if we reach the goal, return 'Reset'
goal_bounds = [self.maze_dim/2 - 1, self.maze_dim/2]
if x in goal_bounds and y in goal_bounds:
   self.reset = True
   self.location = [0, 0]
   self.heading = 'u'
   return 'Reset', 'Reset'
```

here we check if we have reach the goal, if we have, the first run is done and return the a tuple of 'Reset', 'Reset'.

```
#choose next cell to move into based on a combination of the count and the heuristic
        choice_list = []
        for i in range(len(sensors)):
            if sensors[i] != 0:
                cell = [0, 0]
                direction = self.dir_sensors[self.heading][i]
                cell[0] = self.location[0] + self.dir_move[direction][0]
                cell[1] = self.location[1] + self.dir_move[direction][1]
                cost = self.count[cell[0]][cell[1]] + self.heuristic(cell)
                choice_list.append([cost, cell[0], cell[1], i])

        if len(choice_list) != 0:
            choice_list.sort()
            choice_list.reverse()
            choice = choise_list.pop()
            self.location[0] = choice[1]
            self.location[1] = choice[2]

            rotation = self.dir_angle[str(choice[3])]

            if rotation == -90:
                self.heading = self.dir_sensors[self.heading][0]
            elif rotation == 90:
                self.heading = self.dir_sensors[self.heading][2]
            elif rotation == 0:
                pass

            return rotation, 1
```

next we choose which nearby cell(if the nearby cell is not blocked by a wall) to move into based on a combination of the counter number and the heuristic we define earlier. The one with the least cost gets chosen, we also change the heading of the robot based on the rotation it needs to make. The robot only move one step at a time.

```
        # in a dead end, rotate 90(or -90) degree and don't move
    else:
        self.heading = self.dir_sensors[self.heading][2]
        return 90, 0
```

if the choice list is empty, that means the robot is in a dead end, let the robot rotate an angle(here 90 degree) and not move a single step. the next time it will rotate 90 degree again(it's the only option) and move out the dead end.

A_star_search(self) : this function implements the A* algorithm, which is used to find the optimal path within the mapped area of the maze at the start of the second run. It return a policy that contain the optimal path.

```
def A_star_search(self):
    init = [0, 0]
    cost = 1
    heading = 'u'
    goal_bounds = [self.maze_dim/2 - 1, self.maze_dim/2]

    closed = [[0 for row in range(self.maze_dim)] for col in range(self.maze_dim)]
    closed[init[0]][init[1]] = 1

    #the action taken that leads to the current cell
    action = [[' ' for row in range(self.maze_dim)] for col in range(self.maze_dim)]
    #the rotations taken that leads to the current cell
    rotations = [[-1 for row in range(self.maze_dim)] for col in range(self.maze_dim)]
    #the rotation and action that the cell's gonna take
    policy = [[(-1, ' ') for row in range(self.maze_dim)] for col in range(self.maze_dim)]

    x = init[0]
    y = init[1]
    g = 0
    f = g + self.heuristic(init)
    frontier = [[f, x, y, heading]]

    found = False  # flag that is set when search is complete
    resign = False # flag set if we can't find expand
```

The function starts at the [0,0] position, the robot facing ‘up’. The cost of moving towards the next cell is 1. goal_bounds is the boundaries of the goal area.
A few local variables defined to keep track of the informations that are needed to constructed an optimal path solution:
closed: a matrix that marks the correspoding cell closed. If the cell has been added to the frontier or has been visited, it is marked as closed(represented as 1) otherwise open(represented as 0)
action: a matrix that stores the action taken that leads the robot to its current position.The values can be ‘up’, ‘down’, ‘right’, or ‘left’.
rotations: a matrix that stores the rotation taken by the robot at the last cell so it could move to the current cell.
policy: a matrix that stores a tuple representing the rotation and action that the **current** cell’s going to take.

the f value is the evaluation that combines g, the cost to reach the current cell, and h(heuristic), the cost of the cheapest path from the current cell to the goal(here the manhattan distance between the current cell and the center of the goal area).

```
while not found and not resign:
    if not frontier:
        resign = True
        return 'fail'
```

start a loop searching for the optimal path until the goal is reached or fail.

```
else:
        frontier.sort()
        frontier.reverse()
        choice = frontier.pop()
        x = choice[1]
        y = choice[2]
        f = choice[0]
        heading = choice[3]
        g = f - self.heuristic([x, y])
```

choose a cell that has the least cost and make it the current cell.

```
#if goal found, add the policy from the goal back to the root
if x in goal_bounds and y in goal_bounds:
   while x != init[0] or y != init[1]:
      x_p = x - self.dir_move[action[x][y]][0]
      y_p = y - self.dir_move[action[x][y]][1]

      policy[x_p][y_p] = (rotations[x][y], action[x][y])

      x = x_p
      y = y_p

   found = True
```

if the goal is found, add the policy by tracing the action and rotation from the goal back to the root.

```
wall = self.map[x][y]
for i in [1, 2, 4, 8]:
   if wall & i != 0:
      step = self.dir_move[self.action[str(i)]]
      x2 = x + step[0]
      y2 = y + step[1]
      wall2 = self.map[x2][y2]
      if wall2 != -1 and closed[x2][y2] == 0:
         g2 = g + cost
         f2 = g2 + self.heuristic([x2, y2])

         action[x2][y2] = self.action[str(i)]

         rotation_directions = self.dir_sensors[heading]
         rotation = self.dir_angle[str(rotation_directions.index(self.action[str(i)]))]
         rotations[x2][y2] = rotation

         heading2 = heading
         if rotation == -90:
            heading2 = self.dir_sensors[heading][0]
         elif rotation == 90:
            heading2 = self.dir_sensors[heading][2]
         elif rotation == 0:
            pass

         frontier.append([f2, x2, y2, heading2])

         closed[x2][y2] = 1
```

Expand the current node, add the resulting nodes to the frontier if there are no walls in the directions and the nodes are not closed and the nodes have been mapped during the first run.

search_maze(self, sensors): this function uses the policy returned by the A_star_search function to compute the next move's rotation and movement steps. It makes up to 3 steps at most each time the function gets called.

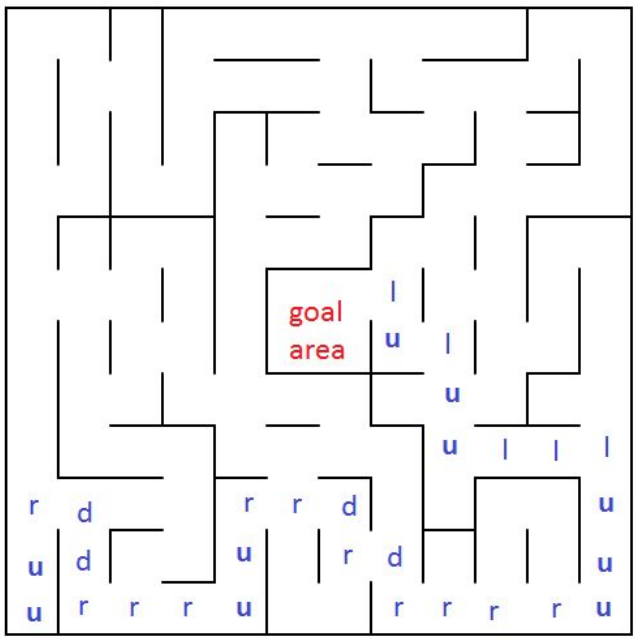
heuristic(self, cell): calculate and return the heuristic value of the cell, which is the manhattan distance from the cell to the center of the goal.

IV. Results

Model Evaluation and Validation

test_maze_01 score: 30.500, coverage:70.83%, first run moves:405, second run moves: 17

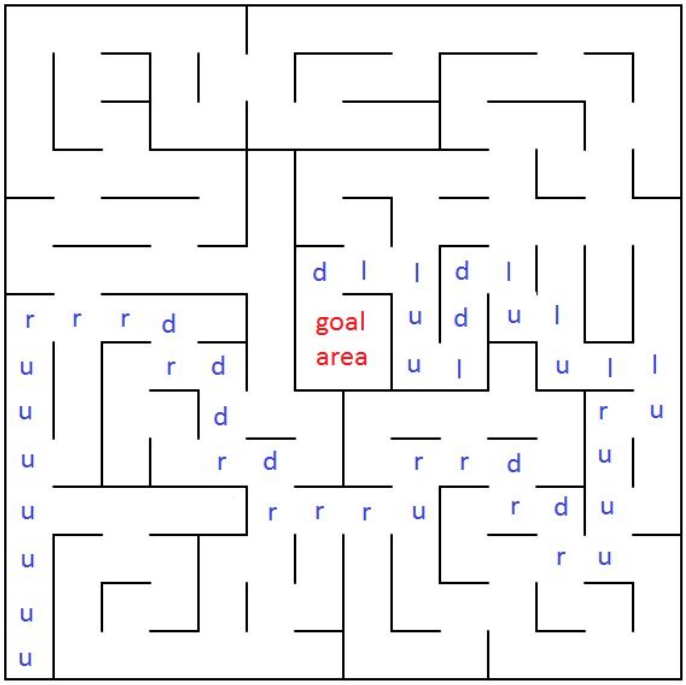
the path by the second run is:



which is the same as the optimal path of the maze.

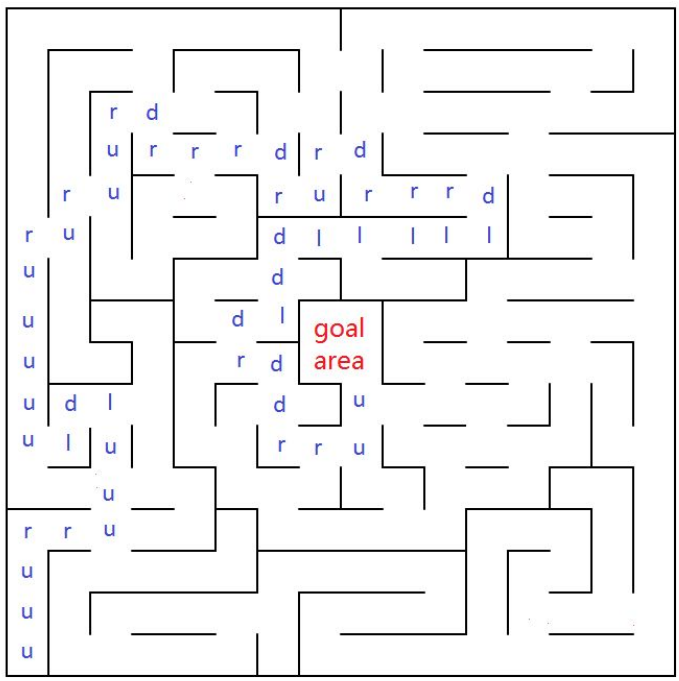
test_maze_02 score: 42.300, coverage: 62.76%, first run moves: 399, second run moves: 29

the path by the second run is:
which is more than the optimal path of 23 moves.



test_maze_03 score: 34.967, coverage: 33.98%, first run moves: 119, second run moves: 31

the path by the second run is:



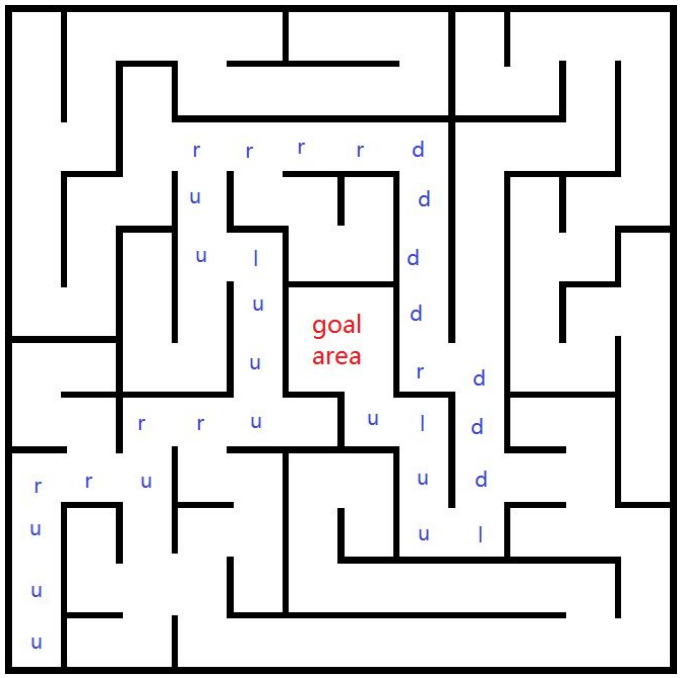
which is more than the optimal path of 25.

Justification

As defined in the benchmark section above, the scores of the best-case scenarios for the 3 mazes are 21.8, 29.5 and 33.5.
The scores for the implementation are 30.500, 42.300 and 34.967, which is a fine performance compared to the best benchmarks.
the scores for the implementation are within the same range and does not vary too much from each other, it means the algorithm is stable enough for the robot in the test mazes to find a good solution within a short amount of time.
The final model performs even better in maze 3 with dimension 16 than in maze 2 with dimension 14. That shows that even though the dimension increases, the model can still find solutions within a short time and not bogged down by the increased number of cells. It proves that the model is robust enough to find an efficient solution for the problem.
The results are given by the tester which performs the actions and informs the robot it's current location and whether it has reach the goal or not. If the tester gives out a final score for the solution, it should be guaranteed that the robot has behaved correctly. So the results are trustworthy and align with what has been expected from the solution.

V. Conclusion

Free-Form Visualization



maze_04: score: 21.567, coverage: 40.97%, first run moves: 137, second run moves: 17

the maze has two shortest paths leading to the goal, however, the algorithm will always prioritise the path going ‘up’ because it uses a heuristic that favor cell whose manhattan distance is shorter.
the algorithm finds the optimal path of the maze.

Reflection

The first major challenge of this project is understanding the provided code. How the maze data are used and represented in the code are especially crucial.
At first I thought that in the code provided the starting position of the maze is [0,0] which is supposed to be the bottom left corner, but in the maze.py file the wall matrix is constructed from top left to bottom right.
But later I figure it out that the wall matrix does start at "top left" which is [0,0]. But the movements {'up': [0, 1], 'right': [1, 0], 'down': [0, -1], 'left': [-1, 0]} does not correspond with the wall matrix. when it move 'up'[0,1], it actually move 'right' in the matrix.
the test maze data are constructed as if they were turned clockwise 90 degree.

Coding the solution takes several steps. First mapping the maze requires me to come up with an algorithm that utilize the sensors information and the maze dimension so that the robot does an efficient job of mapping the maze without having to backtrack a lot and reach the goal in reasonable time.
Then searching through the mapped area to find the optimal path using A* algorithm is relatively straightforward.
But debugging takes a while because there are a lot of things to keep track of in order for the algorithm to work properly, such as the heading and the rotation of the robot.

Improvement

If the scenario took place in a continuous domain, The robot will need a localization method that can produce continuous data about its position in the maze, such as kalman filtering or particle filtering.
An if it’s like the real world, there will be error in the sensor data, and the robot movement. The robot’s movement will be continuous too, which means that it will need a PID controller to continuously calculate the error and adjust the movement of the robot. It will also need to apply SLAM to construct or update a map of the uncertain environment while simultaneously keeping track of the robot's location within it.