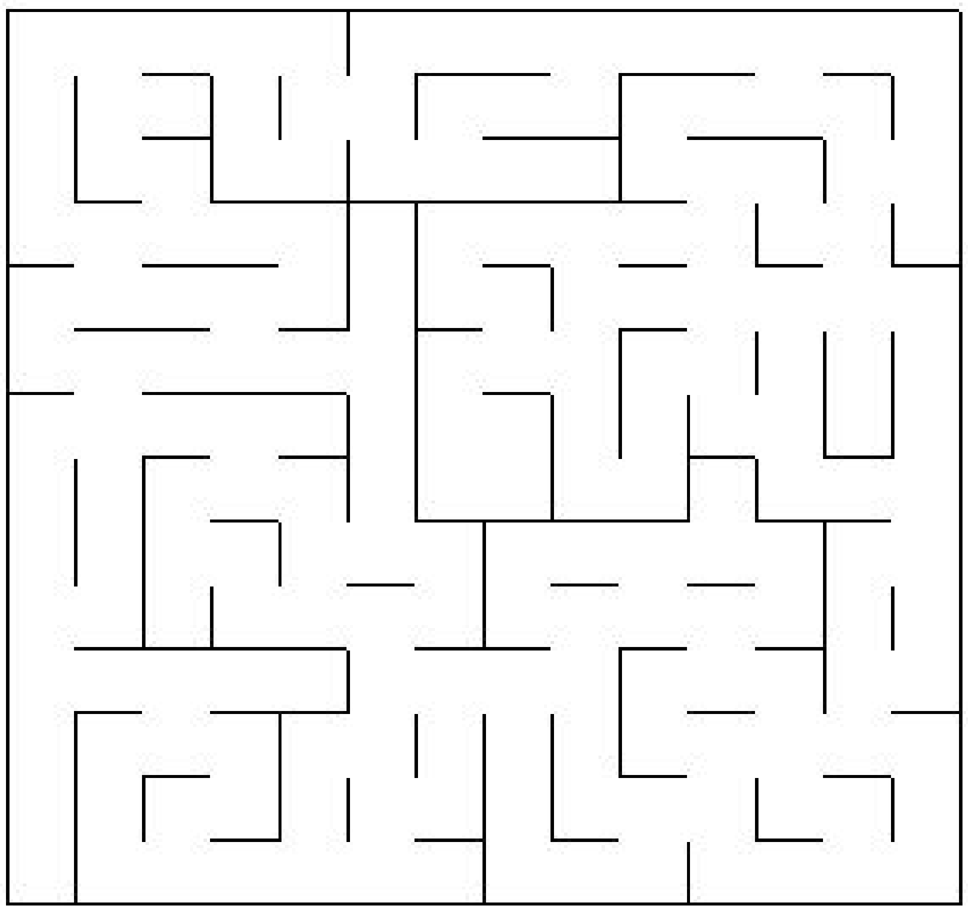
Machine Learning Nanodegree:

Shashwata Mandal

Capstone Report

August 19, 2017

Plot and Navigate a Virtual Maze



Definition

Project Background

The Micromouse is a famous competition, wherein a robot mouse is tasked with plotting a path from a corner of the maze to its center. The robot mouse may make multiple runs in a given maze. In the first run, the robot is made to map out the entire maze. In the this step, the robot visits each and every navigable portion of the maze no matter how long it takes. The main aim of this step is to figure out the best paths to the center. In subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, based on what it learned previously.
The following video is an example of the Micromouse competition.
Having been interested in autonomous robotics, I have always been motivated to problems of this genre. This problem should be solved because the idea behind it governs a large section of the industrial automation industry, where optimization like this can save up a lot of energy hence cost. Refer to Related Works section.

Problem Statement

The maze in this problem is an n x n grid, where n is either 12, 14 or 16. The mouse begins its journey from the bottom left corner of the maze and must navigate a path to the center of the grid, which is a 2 x 2 square.
In each cell, the mouse can sense its distance from the nearest wall in three directions (left, forward and right). It can move at most 3 units in any direction (depending on where the walls are) and it has the option to rotate 90 degrees to the left or right before moving. A rotation and a movement defines one time step.
The main goal of the mouse is to find the shortest route to the center of the grid. The mouse is allowed a minimum of two runs at the maze. The first run has a scoring of 1/30 time point per cell movement whereas each of the successive runs has a scoring of 1 point per cell movement. The objective is to minimize this score.

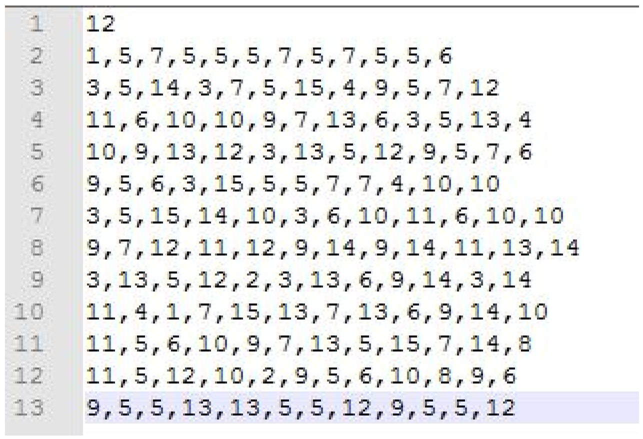
Evaluation Metrics

The mouse has two runs through the maze. It has a limit of 1,000 time steps combined, and it can reset to the starting position any time after it has reached the goal in the first run. In the first run, each step cost 1/30 of a point, where as in the second run each step costs an entire point. Thus, it is beneficial to take the extra steps to map out the entire maze in order to take the minimum number of steps in the consecutive runs. The final score is the summation of the first round cost and the second round cost. A lower score is considered to be better because the score is a cost score. Ie. the score measures the cost. Hence more the score worse the performance. For example, if the robot uses 900 first round moves that cost 30 points (900/30 = 30) and the second round uses 30 moves then the cumulative score becomes 60.

Analysis

Data Exploration and Visualization

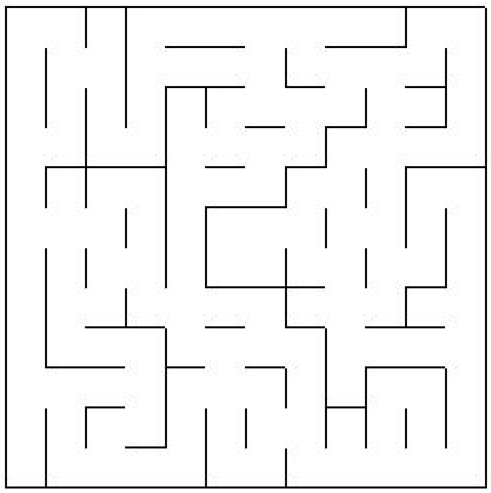
Each maze is constructed from the encoded data in a text file which a **n x n** grid of number 1-15, where each number represents the presence or absence of walls in the cell. For example a number 12 would represent a cell with the top closed and rest of the sides open. The mechanics of the encoding is as follows. If the cell is open on the left, the value increases by 8. If the cell is open on the bottom, the value increases by 4. Again if the cell is open on the right, the value increases by 2. Finally if the cell is open on the top, the value increases by 1. Hence 12 = 0 * 1(**top**) + 1 *2(**right**) + 1 * 4(**bottom**) + 1 * 8(**left**). These values are unknown to the mouse at the beginning. Hence the mouse has to reconstruct this information using the data passed from the sensors as it navigates through the maze.
The following is a sample file content defining a 12 x 12 grid.



```
12
1,5,7,5,5,5,7,5,7,5,5,6
3,5,14,3,7,5,15,4,9,5,7,12
11,6,10,10,9,7,13,6,3,5,13,4
10,9,13,12,3,13,5,12,9,5,7,6
9,5,6,3,15,5,5,7,7,4,10,10
3,5,15,14,10,3,6,10,11,6,10,10
9,7,12,11,12,9,14,9,14,11,13,14
3,13,5,12,2,3,13,6,9,14,3,14
11,4,1,7,15,13,7,13,6,9,14,10
11,5,6,10,9,7,13,5,15,7,14,8
11,5,12,10,2,9,5,6,10,8,9,6
9,5,5,13,13,5,5,12,9,5,5,12
```

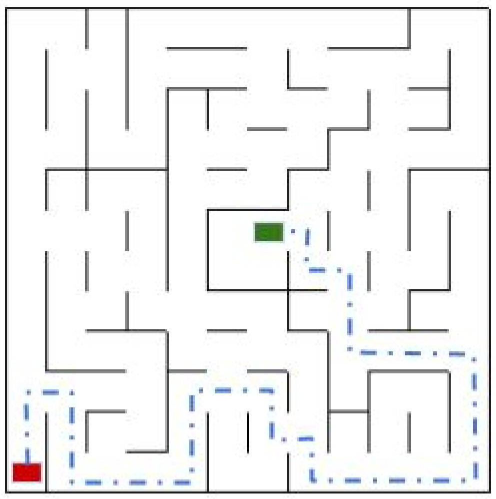
The first row denotes the **n** in (**n x n**).

The following is a rendered view of the maze generated from the text file.



Note: The rows in the text file corresponds to the column in the maze, so the 1 in the first row would represent the starting position in the bottom left corner of the maze.
The mouse has three sensors : front, left and right, which passes proximity information of the mouse ie. its calculated how far away is the nearest wall is any direction. For example, in the starting position, the left and right sensors would pass the value 0 because there are walls directly on the either side, and the forward sensor would pass the value 11 because it can move all the way from the bottom to the top of the maze unobstructed.

The shortest path to this particular maze is as follows:



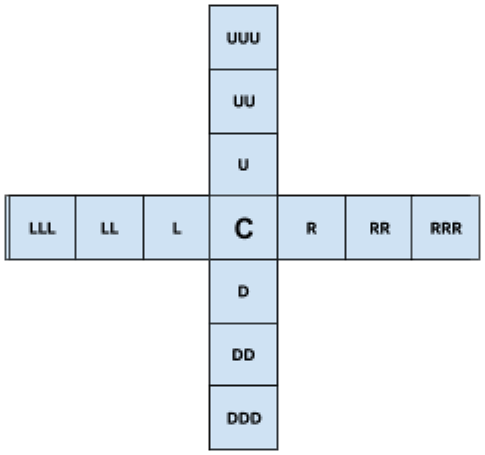
The final task of this

Datasets and Inputs

The datasets are provided in the form of **test_maze_##.txt** input files which are provided to the system. On the first line of the text file is a number describing the number of squares on each dimension of the maze n. On the following n lines, there are **n** comma-delimited number describing sides of the square are open for navigation. Each number represents a four bit number, where bits represent open (if equal to 1; no wall) or close (if equal to 0; walled). The number is to be converted to binary for evaluation thereby getting a binary in the form of **LDRU** where U (1s register) represents upward-facing side, R (2s register) represents the right side, D (4s register) the bottom side and L (8s register) the left side. For example, the number 10 =$(1010)_b$ means that a square is open on the left and right , with walls on top and bottom (0*1 + 1*2 + 0*4 + 1*8 =10). Note that, due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom-left) corner of the maze. For this problem, 3 sample mazes of sizes 12, 14 and 16 shall be used for testing.

Algorithms and Techniques

The real test in this problem is to figure out the map of the maze because the maze is entirely unknown to the robot when it first begins its journey. There are several algorithms for finding the shortest path through a maze, but the majority of them would require prior knowledge of the connection graph of the maze ie. the cell to cell connections. Therefore, in order to perform an optimized run to the destination, the mouse has to initially explore and map the maze. Once this is done, a pathfinding algorithm can be applied.
For the exploration and mapping, a modified randomized depth first search algorithm has been applied where priority will be given to towards unvisited cells.The robot is given a list of possible movements and it makes a random choice from the list.
After each movement, the walls positions of the current cell will be available through the sensors reading and this information will be maintained in a separate grid (**val_grid**) which is essentially a regenerated map of the maze. Once the entire maze is reconstructed, the mouse has already reached the goal points at least once. It then initiates a reset of its run and returns to the starting position to begin its second run.
Once the robot comes back to its starting position, the shortest path algorithm is applied. In this case, a mixture of two algorithms have been used. For measuring the graph weightage, a variation of the flood fill algorithm has been used. Once all the cell in the maze have been weighed as per the shortest distance from the starting point, a backtrack algorithm has been used to figure out the path used for attaining this shortest distance. However there is one more step involved between these two steps. The goal cells consist of four target cells. However reaching once completes our objective. Hence the goal cell with the shortest distance needs to be selected.
The flood fill algorithm here works as follows. First the starting cell is marked as 0 since its distance from itself is 0. After that a series of iterations are performed. Each iteration has one objective. Visit each cell and find the current shortest distance to the cell till the previous computation. The value is calculated by checking 12 cells - **UUU**, **UU**, **U**, **DDD**, **DD**, **D**, **LLL**, **LL**, **L**, **RRR**, **RR** and **R** (where **U** denotes 'up', **D** denotes 'down', **L** denotes 'left' and **R** denotes 'right'). These values are checked because the mouse can mose almost 3 cells per turn. Now for every updation of values, another matrix called **path_grid** is maintained which records the cell for which the current cell value is updated.



The following is a visualization of the weight matrix for maze 1:

```
[4, 5, 7, 9, 10, 10, 10, 11, 11, 11, 15, 16]
[4, 5, 6, 9, 10, 10, 10, 12, 13, 14, 14, 17]
[3, 5, 7, 9, 9, 11, 12, 11, 12, 15, 16, 17]
[3, 4, 7, 8, 9, 10, 10, 10, 15, 15, 16, 16]
[3, 4, 4, 5, 8, 9, 9, 16, 15, 15, 15, 14]
[2, 3, 3, 6, 8, 17, 17, 16, 14, 15, 18, 13]
[2, 4, 4, 5, 8, 18, 18, 15, 14, 16, 17, 13]
[2, 4, 4, 6, 7, 7, 7, 15, 14, 15, 14, 13]
[1, 4, 4, 3, 8, 7, 8, 8, 13, 13, 13, 12]
[1, 2, 2, 2, 5, 6, 6, 9, 14, 11, 11, 12]
[1, 3, 5, 3, 5, 7, 7, 8, 11, 11, 11, 12]
[0, 3, 4, 4, 4, 7, 7, 9, 10, 10, 10, 11]
```

Once the flood fill is complete, a second grid will be created for the actions taken. This will be stored in **action_grid** with the optimal action for each cell. This is performed by a simple back track algorithm ie. the algorithm attempts to track the path which led to the goal by plotting a course to the next step from the previous step which is available in the **path_grid** matrix. The purpose of this step is to translate weights to a logical decision.

Here is a visualization of the action grid for the first maze:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 'left', 'left', 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 'up', 'left', 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 'up', 0, 0, 0]
[0, 0, 0, 0, 0, 'right', 'right', 'down', 'up', 'left', 'left', 'left']
['right', 'down', 0, 0, 'right', 'up', 0, 'down', 0, 0, 0, 'up']
['up', 'down', 0, 0, 'up', 0, 0, 'down', 0, 0, 0, 'up']
['up', 'right', 'right', 'right', 'up', 0, 0, 'right', 'right', 'right', 'right', 'up']
```

In order to have a clearer view of the decision steps, the following arrow matrix is generated from the **action_grid**.

```
['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '<', '<', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '^', '<', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '0', '^', '0', '0', '0']
['0', '0', '0', '0', '0', '>', '>', 'v', '^', '<', '<', '<']
['>', 'v', '0', '0', '>', '^', '0', 'v', '0', '0', '0', '^']
['^', 'v', '0', '0', '^', '0', '0', 'v', '0', '0', '0', '^']
['^', '>', '>', '>', '^', '0', '0', '>', '>', '>', '>', '^']
```

If the robot is made to follow this path from the starting, it will find it's way to the goal. The robot can use a maximum of 3 steps per move. This algorithm takes into account that as well; meaning if it finds a clear path ahead where it can use 2 or 3 steps at a time to cover longer distances, it will. The path we outlined earlier, is the same path which was generated optimally.
The robot can take into account 3 steps at a time. So if there are 2 or 3 consecutive arrows pointing in the same direction, it is to that many steps. With these instructions, even though the mouse must pass through 30 cells in maze 1, it only requires 17 time steps to get to the goal.

Benchmark Model

In order to build a successful algorithm, it is necessary to set up some preliminary benchmarks so that we can determine what to expect from an optimal solution. Since a second round move cost 30 times more than a move in the first round, it is necessary to get as much as possible out of the first round. Depending on the size of the maze, the ideal path is approximated to maze 20-50 time steps. There are a maximum of 1000 time steps allowed and if the mouse uses 900 moves for the first round and 20-50 steps in the second round, then expected scores come to around 50-80. That is considered to be the initial benchmark.
The benchmark set is an approximate one. For the first step 900 steps are considered because 90% of the maximum number of step should be given to the discovery of the grid since this step is random. Hence it needs to be given the maximum possible steps. However since the cost for this is minimal (1/30), the cost for the first run is considered around 30. For the second part the remaining number of step is 100. Now considering the fact that the maze is already discovered and the shortest route has been calculated, the max time steps should be at least half of the remaining time ie. 100/2 = 50. There is no minimum steps in this run but we can consider around 20% of the remaining steps for this process. Then the score in the second round should be around 20-50 since each step accounts for 1 time unit. Hence the estimated score is around 50-80.

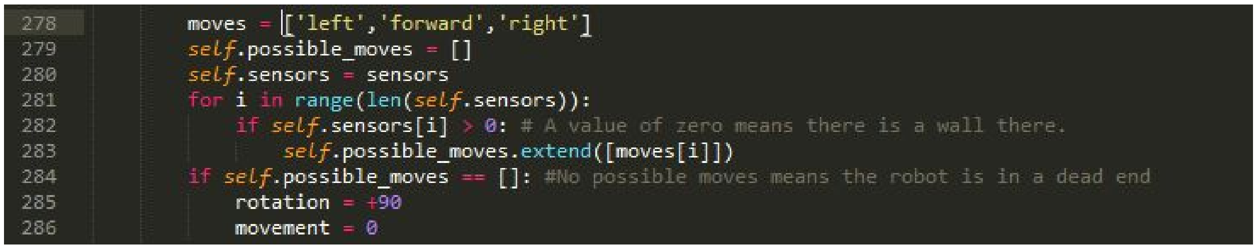
Methodology

Data Preprocessing

Because the specifications of both the maze environment and the robot are provided, no further preprocessing of data is necessary to solve the problem.

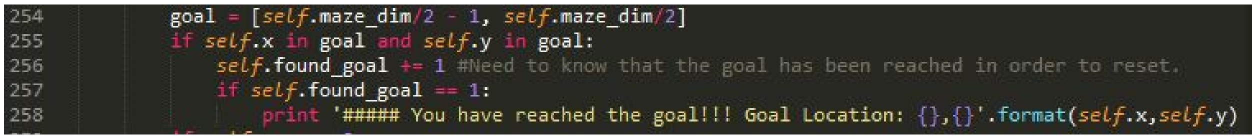
Implementation

The entire implementation phase can be divided into 3 parts - First navigational and mapping run, a shortest path algorithm for figuring out the best path and the final run.
The first step was primarily targeted at getting the mouse to move through the maze without crashing into walls or getting stuck in a dead end, and to update the mouse's current location and facing direction (heading) after each movement.



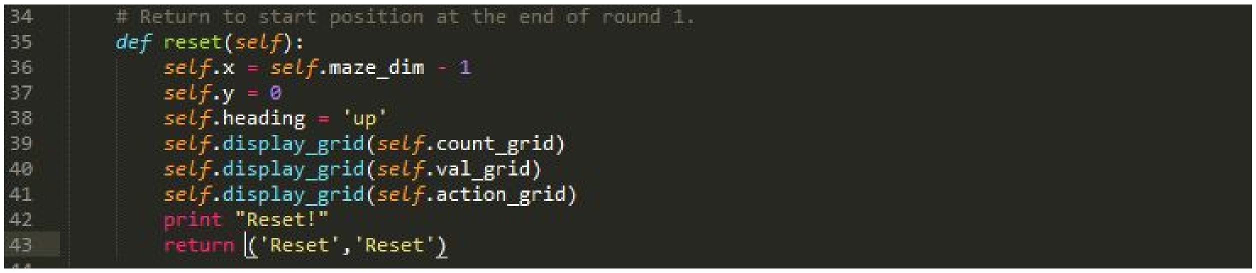
```python
moves = ['left','forward','right']
self.possible_moves = []
self.sensors = sensors
for i in range(len(self.sensors)):
    if self.sensors[i] > 0: # A value of zero means there is a wall there.
        self.possible_moves.extend([moves[i]])
if self.possible_moves == []: #No possible moves means the robot is in a dead end
    rotation = +90
    movement = 0
```

Then a random choice is taken (to remove any sort of bias) from the possible actions and the robot decides on a rotation and movement. If there are no possible moves then the mouse will perform a 90° turn because it detects a dead end.
The robot should identify when at least one of the goal destinations have reached. This was achieved with the following code snippet.



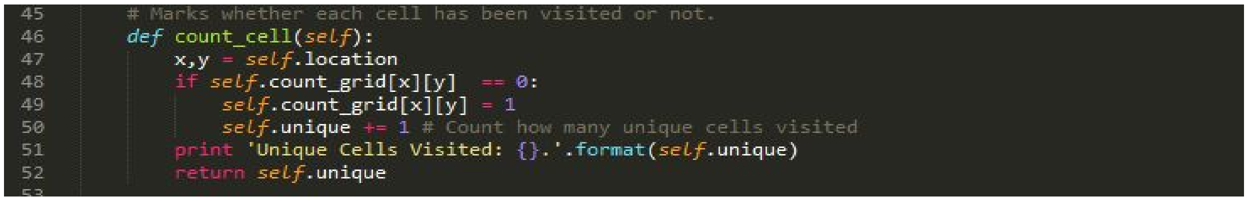
```python
goal = [self.maze_dim/2 - 1, self.maze_dim/2]
if self.x in goal and self.y in goal:
    self.found_goal += 1 #Need to know that the goal has been reached in order to reset.
    if self.found_goal == 1:
        print '##### You have reached the goal!!! Goal Location: {},{}'.format(self.x,self.y)
```

Certain helper function have also been created to help with the exploration and mapping of the maze:



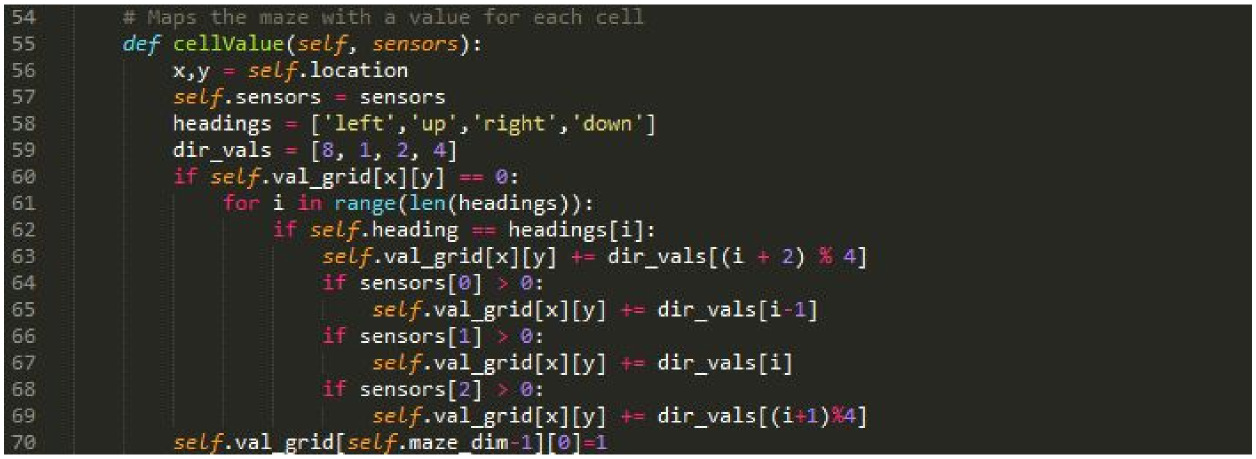
```python
# Return to start position at the end of round 1.
def reset(self):
    self.x = self.maze_dim - 1
    self.y = 0
    self.heading = 'up'
    self.display_grid(self.count_grid)
    self.display_grid(self.val_grid)
    self.display_grid(self.action_grid)
    print "Reset!"
    return ('Reset','Reset')
```

The above reset() function is used to reset the grid and bring it back to the starting points.



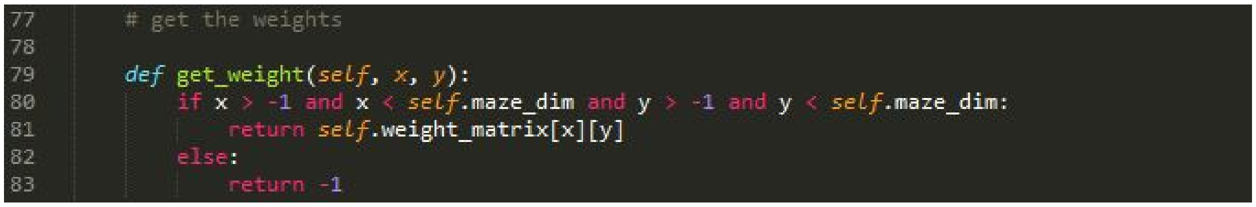
```python
# Marks whether each cell has been visited or not.
def count_cell(self):
    x,y = self.location
    if self.count_grid[x][y]  == 0:
        self.count_grid[x][y] = 1
        self.unique += 1 # Count how many unique cells visited
    print 'Unique Cells Visited: {}.'.format(self.unique)
    return self.unique
```

The count_cell() function is used to keep track of the number of new cells that have been visited.



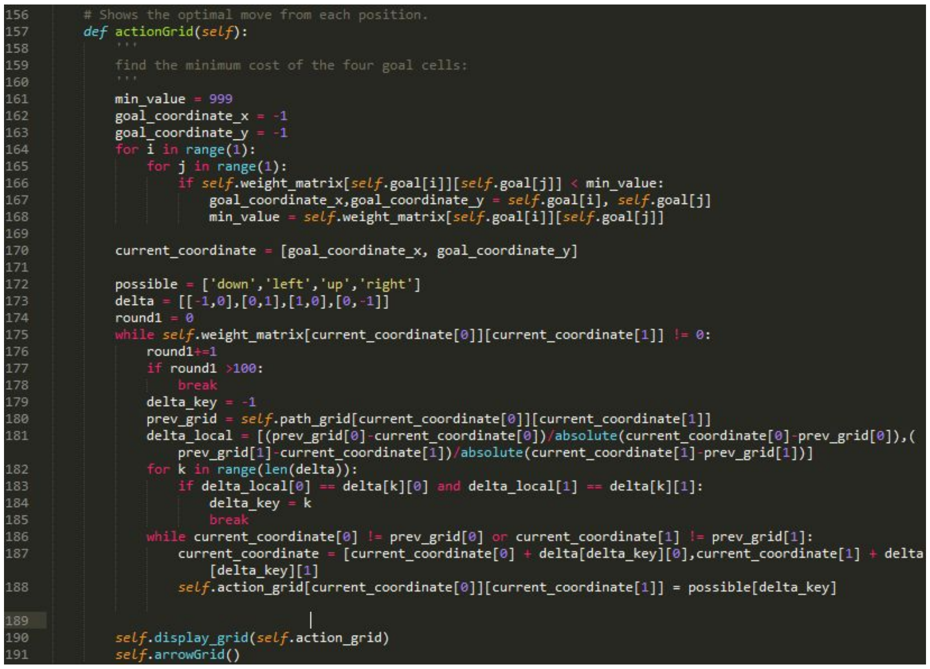
```python
    # Maps the maze with a value for each cell
    def cellValue(self, sensors):
        x,y = self.location
        self.sensors = sensors
        headings = ['left','up','right','down']
        dir_vals = [8, 1, 2, 4]
        if self.val_grid[x][y] == 0:
            for i in range(len(headings)):
                if self.heading == headings[i]:
                    self.val_grid[x][y] += dir_vals[(i + 2) % 4]
                    if sensors[0] > 0:
                        self.val_grid[x][y] += dir_vals[i-1]
                    if sensors[1] > 0:
                        self.val_grid[x][y] += dir_vals[i]
                    if sensors[2] > 0:
                        self.val_grid[x][y] += dir_vals[(i+1)%4]
        self.val_grid[self.maze_dim-1][0]=1
```

This function is used to retrieve the weights and prevents error on invalid coordinates.



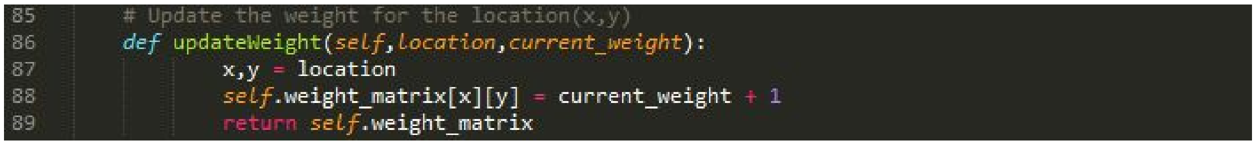
```python
    # get the weights

    def get_weight(self, x, y):
        if x > -1 and x < self.maze_dim and y > -1 and y < self.maze_dim:
            return self.weight_matrix[x][y]
        else:
            return -1
```

The action grid function is used to create the action matrix for the mouse. Once the weights are available for the robot to learn from, the job of the action grid function is to read that information and break it down to machine readable instruction. The consequence of every action is already present in the weight matrix hence the function tries to connect the current cell to the cell which was the predecessor to it when the weight was allotted to it. In this way we can make sure we get back the shortest mapped sequence of steps to the goal cells.



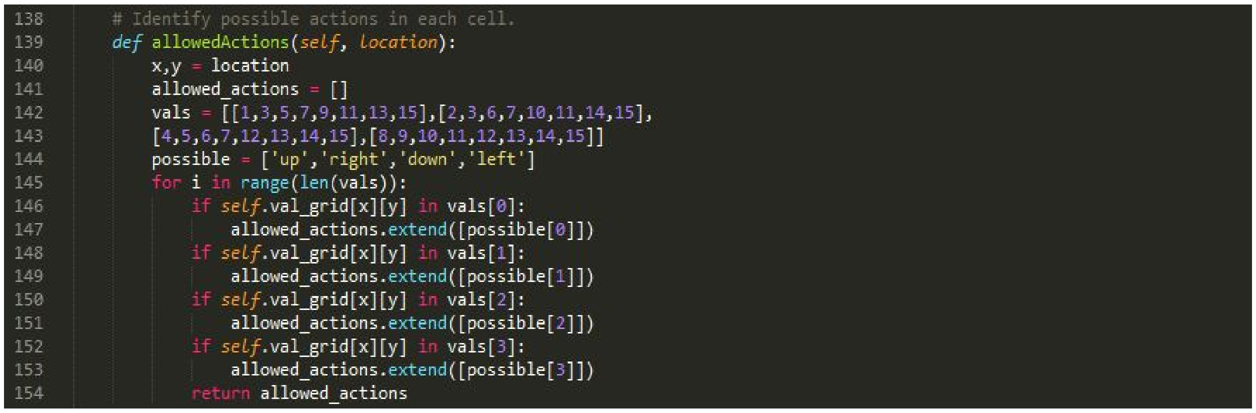
```python
    # Shows the optimal move from each position.
    def actionGrid(self):
        '''
        find the minimum cost of the four goal cells:
        '''
        min_value = 999
        goal_coordinate_x = -1
        goal_coordinate_y = -1
        for i in range(1):
            for j in range(1):
                if self.weight_matrix[self.goal[i]][self.goal[j]] < min_value:
                    goal_coordinate_x,goal_coordinate_y = self.goal[i], self.goal[j]
                    min_value = self.weight_matrix[self.goal[i]][self.goal[j]]

        current_coordinate = [goal_coordinate_x, goal_coordinate_y]

        possible = ['down','left','up','right']
        delta = [[-1,0],[0,1],[1,0],[0,-1]]
        round1 = 0
        while self.weight_matrix[current_coordinate[0]][current_coordinate[1]] != 0:
            round1+=1
            if round1 >100:
                break
            delta_key = -1
            prev_grid = self.path_grid[current_coordinate[0]][current_coordinate[1]]
            delta_local = [(prev_grid[0]-current_coordinate[0])/absolute(current_coordinate[0]-prev_grid[0]),(
                prev_grid[1]-current_coordinate[1])/absolute(current_coordinate[1]-prev_grid[1])]
            for k in range(len(delta)):
                if delta_local[0] == delta[k][0] and delta_local[1] == delta[k][1]:
                    delta_key = k
                    break
            while current_coordinate[0] != prev_grid[0] or current_coordinate[1] != prev_grid[1]:
                current_coordinate = [current_coordinate[0] + delta[delta_key][0],current_coordinate[1] + delta
                    [delta_key][1]
                self.action_grid[current_coordinate[0]][current_coordinate[1]] = possible[delta_key]


        self.display_grid(self.action_grid)
        self.arrowGrid()
```

The updateWeight function is used to update the weight matrix cell by one. Whenever a cell is provided with a weight (weight of a potential predecessor), it takes the value and updates it.



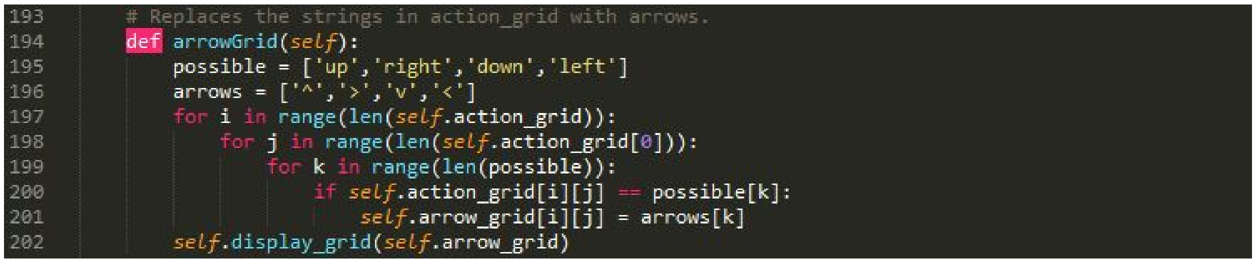
```python
    # Update the weight for the location(x,y)
    def updateWeight(self,location,current_weight):
            x,y = location
            self.weight_matrix[x][y] = current_weight + 1
            return self.weight_matrix
```

The allowed action function read the val_grid matrix (ie. the matrix recording the cell wall information), and returns the set of possible actions in that cell.



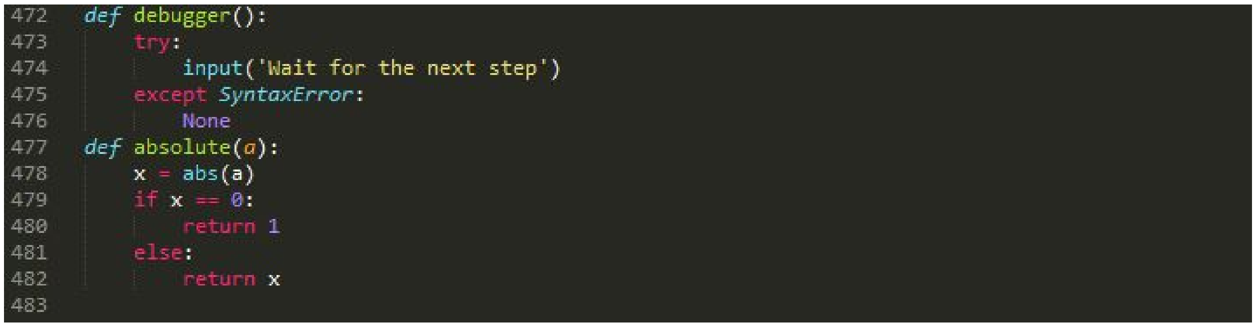
```python
    # Identify possible actions in each cell.
    def allowedActions(self, location):
        x,y = location
        allowed_actions = []
        vals = [[1,3,5,7,9,11,13,15],[2,3,6,7,10,11,14,15],
        [4,5,6,7,12,13,14,15],[8,9,10,11,12,13,14,15]]
        possible = ['up','right','down','left']
        for i in range(len(vals)):
            if self.val_grid[x][y] in vals[0]:
                allowed_actions.extend([possible[0]])
            if self.val_grid[x][y] in vals[1]:
                allowed_actions.extend([possible[1]])
            if self.val_grid[x][y] in vals[2]:
                allowed_actions.extend([possible[2]])
            if self.val_grid[x][y] in vals[3]:
                allowed_actions.extend([possible[3]])
            return allowed_actions
```

The arrow grid function is used to convert the action grid actions to human readable arrows for better understanding of the path followed.



```python
    # Replaces the strings in action_grid with arrows.
    def arrowGrid(self):
        possible = ['up','right','down','left']
        arrows = ['^','>','v','<']
        for i in range(len(self.action_grid)):
            for j in range(len(self.action_grid[0])):
                for k in range(len(possible)):
                    if self.action_grid[i][j] == possible[k]:
                        self.arrow_grid[i][j] = arrows[k]
        self.display_grid(self.arrow_grid)
```

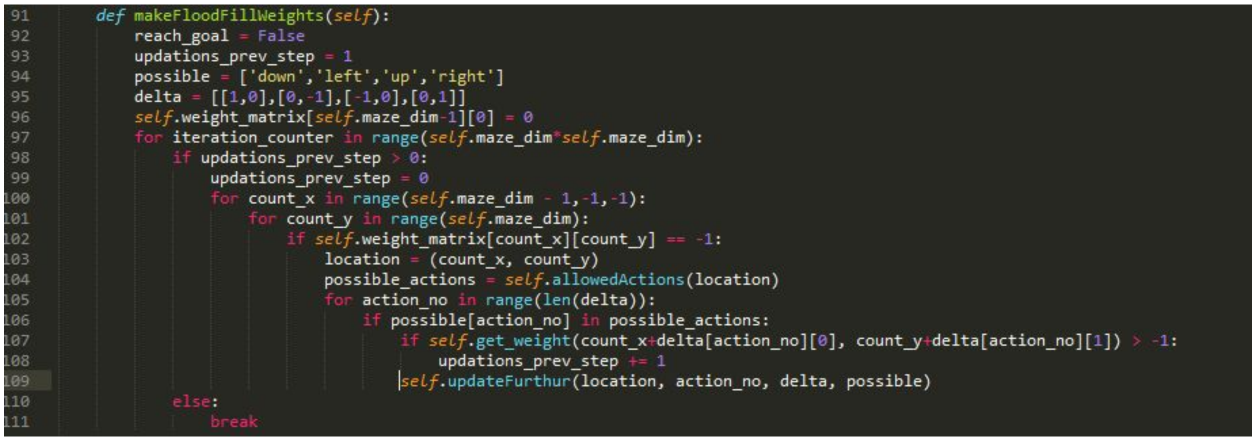
The following methods are used for debugger and as helper methods:



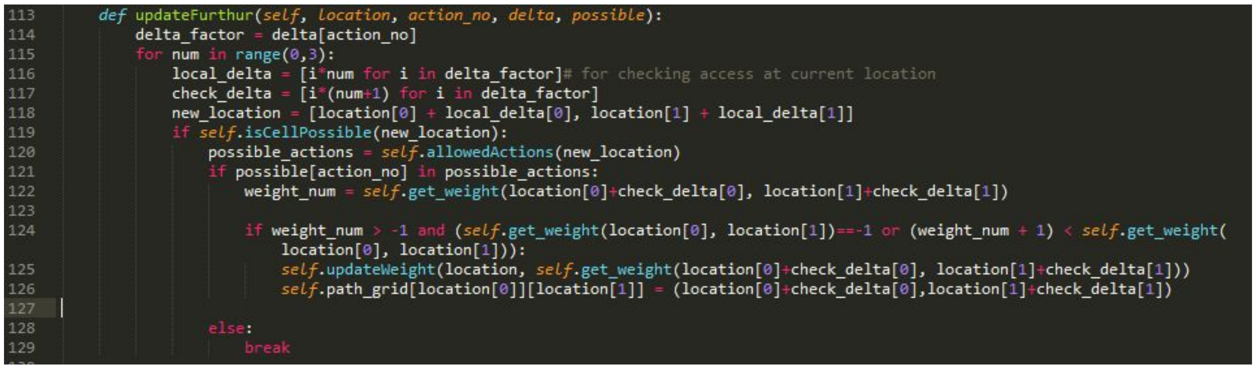
```python
def debugger():
    try:
        input('Wait for the next step')
    except SyntaxError:
        None
def absolute(a):
    x = abs(a)
    if x == 0:
        return 1
    else:
        return x

```

The make flood fill weights and update further functions are used to generate the weight matrix for the robot. The general process is simple. The whole matrix is iterated over multiple times. During each iteration, each cell in the matrix is checked for a weight update ie. whether it is eligible for a weight in that turn. First the cell is checked for possible moves in that cell (to check its accessibility). Next along each possible move, up to 3 cells are checked for accessibility in that direction. All accessible cells with the least weight is allowed to update the value of the current cell.

The following codes are for the flood fill algorithm:

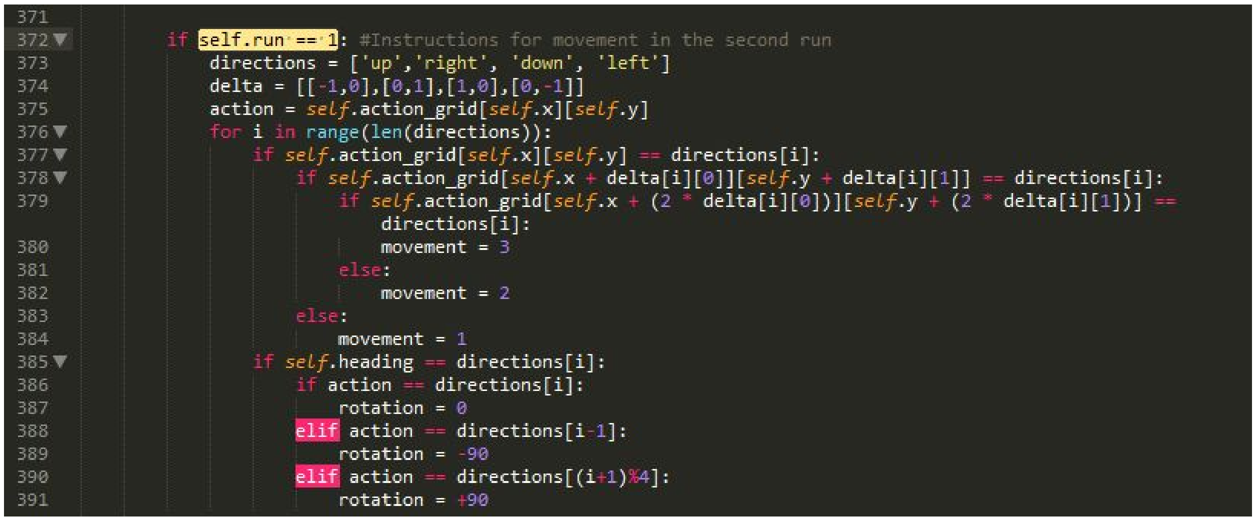


```python
    def makeFloodFillWeights(self):
        reach_goal = False
        updations_prev_step = 1
        possible = ['down','left','up','right']
        delta = [[1,0],[0,-1],[-1,0],[0,1]]
        self.weight_matrix[self.maze_dim-1][0] = 0
        for iteration_counter in range(self.maze_dim*self.maze_dim):
            if updations_prev_step > 0:
                updations_prev_step = 0
                for count_x in range(self.maze_dim - 1,-1,-1):
                    for count_y in range(self.maze_dim):
                        if self.weight_matrix[count_x][count_y] == -1:
                            location = (count_x, count_y)
                            possible_actions = self.allowedActions(location)
                            for action_no in range(len(delta)):
                                if possible[action_no] in possible_actions:
                                    if self.get_weight(count_x+delta[action_no][0], count_y+delta[action_no][1]) > -1:
                                        updations_prev_step += 1
                                    self.updateFurthur(location, action_no, delta, possible)
            else:
                break
```



```python
    def updateFurthur(self, location, action_no, delta, possible):
        delta_factor = delta[action_no]
        for num in range(0,3):
            local_delta = [i*num for i in delta_factor]# for checking access at current location
            check_delta = [i*(num+1) for i in delta_factor]
            new_location = [location[0] + local_delta[0], location[1] + local_delta[1]]
            if self.isCellPossible(new_location):
                possible_actions = self.allowedActions(new_location)
                if possible[action_no] in possible_actions:
                    weight_num = self.get_weight(location[0]+check_delta[0], location[1]+check_delta[1])

                    if weight_num > -1 and (self.get_weight(location[0], location[1])==-1 or (weight_num + 1) < self.get_weight(
                        location[0], location[1])):
                        self.updateWeight(location, self.get_weight(location[0]+check_delta[0], location[1]+check_delta[1]))
                        self.path_grid[location[0]][location[1]] = (location[0]+check_delta[0],location[1]+check_delta[1])

                else:
                    break
```

Once the algorithm performs its function, the second run is initiated. The following codes provide the instructions for movement in round 2:



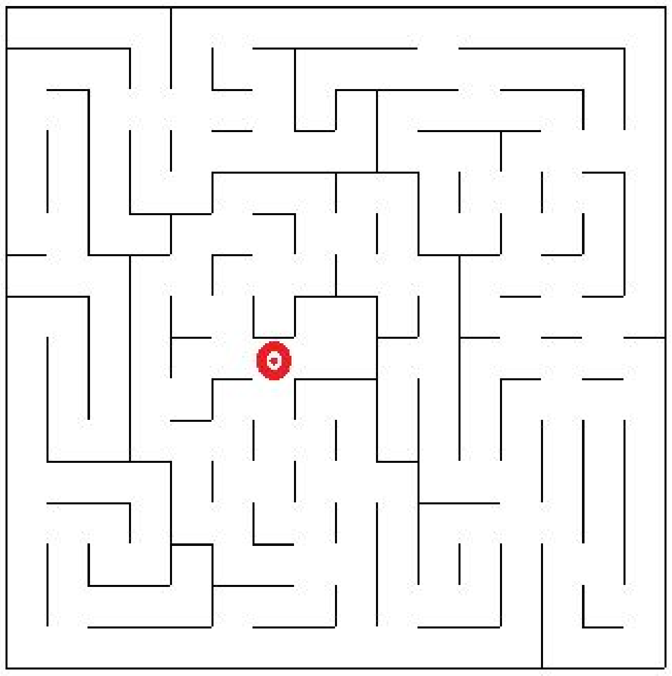
```python

        if self.run == 1: #Instructions for movement in the second run
            directions = ['up','right', 'down', 'left']
            delta = [[-1,0],[0,1],[1,0],[0,-1]]
            action = self.action_grid[self.x][self.y]
            for i in range(len(directions)):
                if self.action_grid[self.x][self.y] == directions[i]:
                    if self.action_grid[self.x + delta[i][0]][self.y + delta[i][1]] == directions[i]:
                        if self.action_grid[self.x + (2 * delta[i][0])][self.y + (2 * delta[i][1])] ==
                            directions[i]:
                            movement = 3
                        else:
                            movement = 2
                    else:
                        movement = 1
                if self.heading == directions[i]:
                    if action == directions[i]:
                        rotation = 0
                    elif action == directions[i-1]:
                        rotation = -90
                    elif action == directions[(i+1)%4]:
                        rotation = +90
```

The main complication was generated because of the 3 steps per time clause. This was logically difficult to form. If the max steps were 1, they could be solved by checking for the neighbouring cells with weight less by one. However when the robot can jump over cells, it became difficult. This was solved using the path_grid which kept track of the predecessor at each step.

Refinement

The algorithm used here and the code used should theoretically find the most optimal solution if the entire maze is mapped properly.
However the challenge is when facing a larger maze. For example, in a n=16 maze, the robot has a small chance of struggling to explore and map out the entire maze. Usually the robot still manages to find a situational shortest path, but if the number of unexplored cells increases, the model might fail.

The following diagram displays one such scenario. If the robot never occupies the cell marked with the red **Bullseye**, it will be unable to initiate the flood fill properly and therefore fail to provide the most optimal solution.



This particular problem would arrive especially if the first round navigation was attempted to be made more optimistic by trying to cover more ground using the maximum travel distance of 3 units per unit time. However this created a problem in this particular case.. Hence this particular problem has been removed by adding the instruction that while the robot is inside the goal, every movement should be just one unit in length, so that the cell with the **Bullseye** is never missed out.
Addition refinements made to improve the model including avoiding multiple visits to dead or explored cells and not getting caught travelling in loops in the maze. No model is completely perfect, hence the performance of this model needs to be reevaluated repeatedly to identify logical glitches. However this model does hold up fairly well against the benchmark set earlier.

Results

Model Evaluation and Validation

In order to successfully validate a model, it must undergo several rounds of testing to indicate its performance. Since the model with the simulation has no separate iterative methods of testing multiple rounds, this must be done manually. Here we are considering 10 rounds of observation for each of the three mazes. The standard deviation for each is noted down as well.

Maze size (**nxn**)	First Round Moves	S.D for First Round Moves	Second Round Moves	S.D for Second Round Moves	Score	S.D for Score
Maze 1 (n = 12)	661.70	199.09	17.01	0.1	39.09	6.63
Maze 2 (n = 14)	879.8	120.44	22.7	1.33	52.06	4.64
Maze 2 (n = 16)	936.6	28.06	25	2.21	56.55	2.22

It can be noticed that the number of steps increases with the increase in size of the maze and this is true since the number of steps should increase to cover more ground in a larger maze. It can also be observed that the standard deviation decreases in case of first round moves and the score with increase in maze size. This is because the maze is forced to reset at 950 steps for the first round. Hence more the size more is the chance of hitting that max number of first round steps. Hence the SD decreases with increase in maze size. The increase in SD of the second round is also very important since it increases. This is largely because the mouse if unable to finish scanning the entire grid hence it is forced to give a pseudo optimal route to the goal. This chances with change in the exploration and mapping direction. Hence as the maze size increases, the routes to the goal also increases with each having a different and largely varying weight.

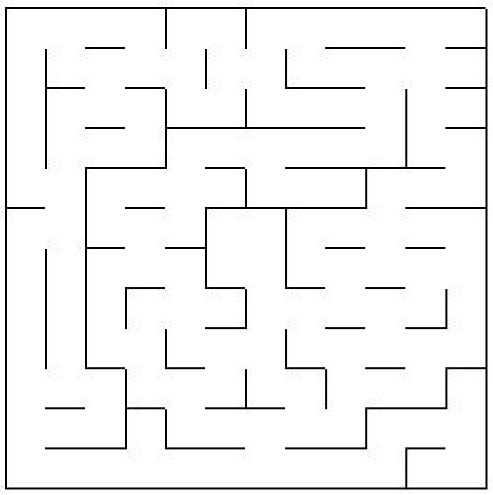
Justification

The results are justifiable. The benchmark for the final scores set earlier was between 50-80 depending on the maze. The scores achieved from the test runs are actually even better than expected. One even has an average score lesser than expected. This shows that the flood fill algorithm was indeed a good choice of algorithm for finding the shortest path to the goal.

Conclusion

Free-Form Visualization

This model was tested on an additional model. This was a 12 x 12 maze made primarily to force the robot into making more steps during navigation. Let us see the results of a set of 10 test runs made in this maze.
The following maze is a visualization of that newly constructed maze.



The robot has performed even better in the 10 trials in this maze. The average number of steps required was 722.13 with a standard deviation of 105.83. The average number of second round moves made was 7 with a standard deviation of 0 meaning it always found the best path. The average score came out to be 31.10 with a standard deviation of 3.52.

Reflection

This project was a relatively challenging project. The specification were provided using a coded syntax and in order to visualize the maze the syntax had to decoded. However the most challenging part was to identify the shortest path and at the same time minimize the score. The mapping is very time consuming but had to be completed in order to get the best solution in the second part.
Any attempt to find the shortest path to the goal directly is useless because the maze is unknown to the robot. I have actually implemented the algorithm on a real life mouse robot earlier but there are significant differences when trying to code up a simulation.
For example in real time robot the program control tends to be with a single infinite loop from which the control is delegated to smaller program controls like turn_right() or turn_left(). However in this case the program control is not available to the robot and the moves are carried out via a single next_move() function. This was a challenge for me.

The movement code had to be broken down into simple manageable pieces in order to control the robot movement. This required sketching up a pseudo code before actually sitting to write the code. The shortest path algorithm was relatively simple because I have actually used it on a robot before and apart from the space complexity, it does manage to provide the most efficient path. However even in this case there was a challenge. Normally a robot can move one step at a time hence path is calculated using a simple flood fill and backtrack algorithm. However in this case the flood fill algorithm needed modifications. It was to be made eligible for handling up to 3 units of fill at a time. Hence the simple backtrack using the adjacent weight was not possible. Hence a separate matrix had to be maintained which kept track of all the updation and which cell updated which cell. Once this step was overcome it became relatively easy.

Improvement

When we think about what this particular model would be in a real life physical maze, there are many improvements that need to be made.
The robot here makes discrete movement moving from one cell at a time, rotating and movement in one unit time; that too upto 3 units at a time. In real life most of this won't be possible. For example turning. In turning there is an added cost. It requires physical movement of wheel. Hence in order to find the fastest (and not the shortest) path to the goal, the rotations should be given some penalty weights. Secondly moving a unit distance forward would definitely take approximately three times less time that moving 3 unit distances. Also and most importantly when a robot is logically in a cell, it is in the cell; however when a robot is physically in a cell, it needs to be at the center of the cell. If not the sensor readings would waver.
The sensor reading would also require some noise handling. Sensors tend to pick up noise and behave erratically at times. Lack of proper noise handling would result in improper mapping of the maze. The sensors should also measure up to one cell farther at a time. This is because the number of cell the robot can measure at a time depends on the angle in which the robot is standing at a time. In case the robot is not orthogonal to the walls of the cell, the reading of the distant cell walls would be incorrect. Hence it is better to measure the walls of one cell a time.
Further the concept of restart complicates things. In order to run this exact problem in a physical would there needs to be some sort of permanent storage to save the map data. In real life the robot will stop once the mapping in complete and needs to be brought to the starting position by hand. At this time lack of storage devices would result in the loss of mapping data.
An advanced mapping and navigation technique called SLAM (simultaneous localization and mapping) method may also be used to save battery power for the robot.