PLOT AND NAVIGATE A VIRTUAL MAZE

Agustin Baltasar Maletti

Udacity 30 November of 2018

Capstone project
Machine Learning engineer
Nanodegree

Components

Definition
Project overview
problem statement
Metrics

Analysis
Data exploration
Exploratory visualization
Algorithms and techniques
Benchmarks

Methodology
Data preprocessing
Implementation
Refinement

Results
Model evaluation and validation
Justification

Conclusion
Free form visualization
Reflection
Improvement

DEFINITIONS

Project overview:

Origin of project:
The project is based on the competition of micro mouse robot for solving mazes[i]. The objective of this competition is to found the center of a maze in the minor time possible. The first time the robot initiate his race is for exploring purpose, for identifying where each wall is and also where is the goal. In this exploring phase if the robot reach the goal is available to continue exploring. After this exploring phase comes the race phase which objective is to find the goal in the fastest way as possible.
Project itself:
In our case I program a robot class capable of explore and solve different maze shapes. In the exploration the objective is gather information about the environment included the goal, them in the exploiting phase the objective is reach the goal in the minimum steps.
The robot class can execute four different models with different characteristics, which I'll explain in detail later, the models are:

1. Random
2. Random with detector dead end feature
3. Random with detector dead end and visited counts features
4. A star dynamic optimizer with detector of dead end and visited counts features

My main model A star dynamic optimizer, will use the A_star algorithm for exploring and Dynamic programing technique for the optimization run.

Problem Statement:

The objective is to reach the goal area in the center of the map as fastest as possible. The mazes are squares of equal rows and columns with a dimension of 12x12, 14x14 and 16x16. The goal area is in the center of each maze being a 2x2 area.
The virtual mouse will start in the bottom left corner and from that will start his exploratory episode. Them if it found the goal and fulfill the exploration requirement, restart again in the initial position and start his optimization episode. This divide our problem in two subproblems:

- First episode: exploratory problem
- Second episode: optimization route problem

The purpose of the exploratory episode is to gain knowledge about the location of the goal, the walls and the general shape of the maze, in this episode if the mouse reach the goal is allowed to continue exploring.
In the second episode the optimization of the route means that the mouse most reach the goal in the minimum steps as possible. Each episode is limited to a number of 1000 steps. The mouse is configured as such that it could make three movement forward or backward [3, -3] as maximum movement on each step. Also is allowed to rotate in 90 degree clockwise and counterclockwise -90 degrees. Also the mouse have three sensor, one in front, other in

the left and other in the right side. This sensors measure the distance to the walls.
In the virtual mechanism of displacement first come the rotation and them the movement.
The general composition of the problem is in a discrete space.
Other problem is that turning right 90 degrees does not mean that the heading of the robot is ending pointing right. For example if the head of the mouse is looking down and turn right 90 degrees this end the position of the head looking to the left.
Other problem is that the design of the maze include dead ends roads, and this means that in that group of tiles our robot will receive a tuple from the sensors with three zeros. Meaning that he have walls in front, on the right and in the left, this if not treated correctly could break our algorithms. For this I will have to implement a dead end detector.
Other problem is that the design of the maze include loops in the route that the mouse can make. For this I'll have to include a memory of every visited place.

Metrics:

The metric that will guide my evaluation will be the score generated by each run in the robot interaction.
Score = Quantity of steps on the first episode * 1/30 + quantity of steps in the second episode
This score is composed by two terms, the first terms represent the exploratory phase and have a lower weight in the score, and the second term is the quantity of step used to reach the goal and is the most important in the score formula.
Is important to note that as more as we explore in the first phase worst will be our score.

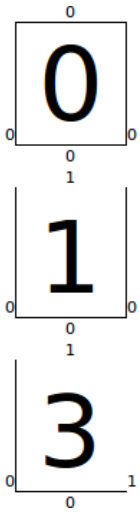
ANALYSIS

Data exploration and exploratory visualization:

The dataset come from three txt file that represent each maze configuration. In the first line of each file we have the dimension of the maze 12, 14, and 16. In the subsequent lines we have comma separated numbers that represents the passages in the maze.

Tile composition:
This representation is committed with the binary values of each number.
The numbers ares from one to fifteen, encapsulating four binary register each one.
The first register correspond to the top square edge, the second register to the right edge, the forth to the bottom edge and the eighth to left edge.

Each register in the tile will be multiply by 0 or one, the sum of the product make the final number that represent each tile. One thing to keep in mind is that a one multiplying a register mean that is free of wall in that directions and a binary value of 0 means that is a wall in that direction.

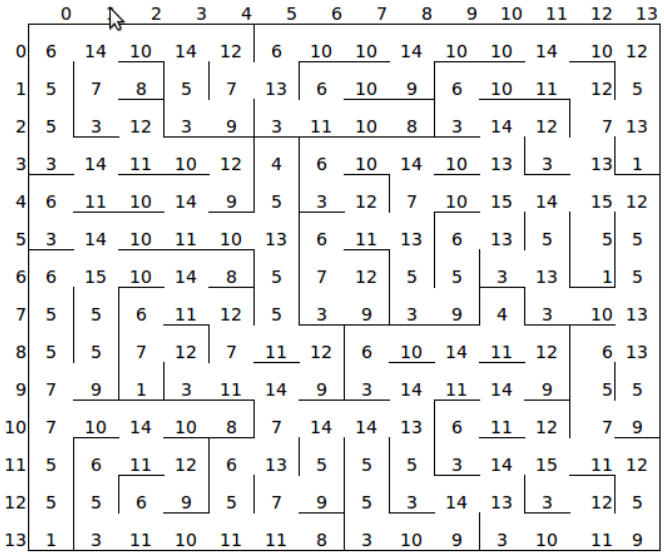


$$0 \quad 0 \quad 0 \quad 0$$
$$1 \quad 1 \quad 1 \quad 1$$

$$1_{equal} \qquad 1*1+0*2+0*4+0*8$$

$$3_{equal} \qquad 1*1+1*2+0*4+0*8$$

The result of combining this tiles is a maze as in this example:



Tile classification:

The variety of the tiles allow us to classify them in eight couples:

	0000	1111	
Full tile	0	15	Full empty tile
	0001	0100	
Lower dead end	1	4	Upper dead end
	0101	1010	
Paralel vertical	5	10	Paralel horizontal
	1000	0010	
Right dead end	8	2	Left dead end
	0011	1001	
Left lower corner	3	9	Right lower corner
	0110	1100	
Upper left corner	6	12	Upper rihgt corner
	0111	1101	
Right vertical line	7	13	Left vertical line
	1011	1110	
Lower hirizontal line	11	14	Upper horizontal line

I also classify the tiles in four list based on the allowed direction of movement, them I use this concept in my implementation and also help me to better understand the whole problem.

The tiles with binary number of [1, 3, 5, 7, 9, 11, 13, 15] allow the robot to move up.
The tiles with binary number of [2, 3, 6, 7, 10, 11, 14, 15] allow the robot to move right.
The tiles with binary number of [4, 5, 6, 7, 12, 13, 14, 15] allow the robot to move down.
And the tiles with binary number of [8, 9, 10, 11, 12, 13, 14, 15] allow the robot to move left.

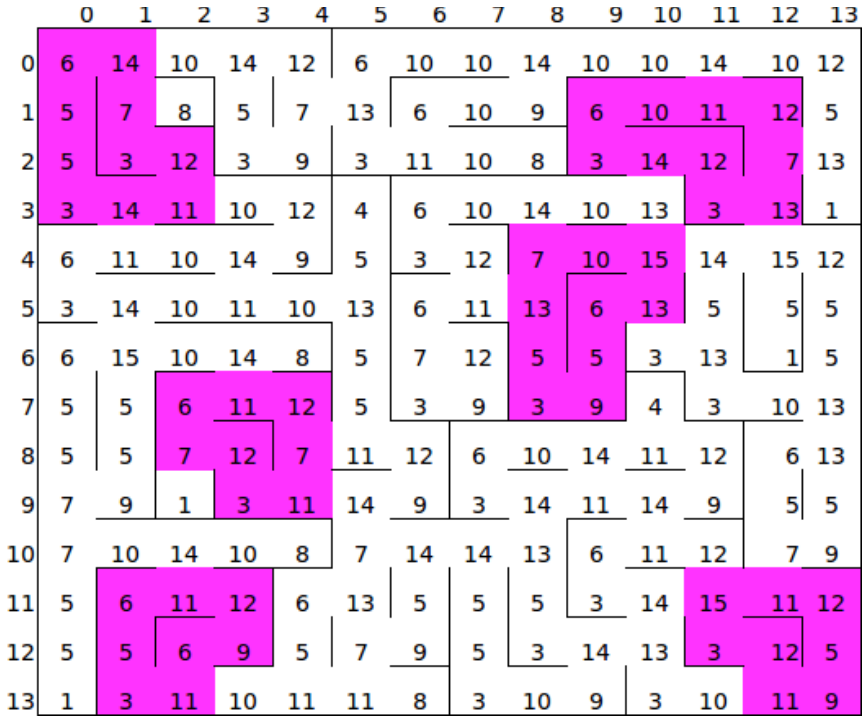
Maze characteristics:
The resultant design of each maze creates conditions that our algorithm must have in count, the dead ends geography and the loops in the possibles routes.
In this images are highlighted in orange the dead ends path and in yellow the goal of the maze is important that our robot can recognize and avoid the dead end path:

Maze 01:

X Index Y	0	1	2	3	4	5	6	7	8	9	10	11	Y
0	6	12	4	6	10	10	14	14	10	8	6	12	
1	5	7	13	7	10	10	13	3	14	14	9	5	
2	5	5	5	5	4	6	11	14	9	7	8	5	
3	7	9	3	9	7	11	14	9	6	15	10	9	
4	5	4	6	12	7	10	9	6	13	5	6	12	
5	7	15	13	5	5	6	14	13	7	13	5	5	
6	5	5	7	13	5	3	9	3	13	7	9	5	
7	5	7	9	3	15	10	12	2	15	9	2	13	
8	5	3	10	12	3	14	11	12	7	10	10	13	
9	7	14	10	13	6	15	12	5	1	6	12	5	
10	5	5	6	9	5	5	7	13	4	5	5	5	
X	11	1	3	11	10	9	3	9	3	11	11	11	9

In the next image of the maze two I highlight in magenta the possible loops in the route of the robot, is important that the robot avoid entering in loops for this I'll implement a grid for counting the visit in each tile.

Maze 02:



In the next image is highlighted in green a path to the goal.

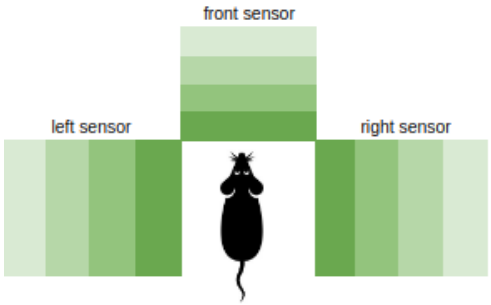
Maze 3:

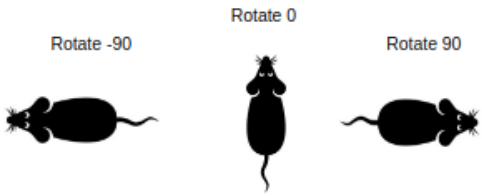
The robot mouse data:

The mouse data is composed by the localization, the sensors, and the action. The action tuples is composed by rotation and movement values. The localization is a tuple of two values for coordinates x and y in the maze, for example (11,0) is the start position. The sensor is a tuple of three numbers for the distance to walls, for example the first tuple sensor is (0,11,0), meaning that the mouse have walls in the left and right side and a free way of 11 tiles in the front. The action tuple is composed of rotation (90, -90 or 0) and movement, limited to (3,-3), for example a (0, 2), mean that the mouse don't rotate and move forward 2 tiles.

The rotation come first, this means that first change the heading of the mouse and them move. The movement is made by adding the correspondent value to the coordinates of the mouse.

Is assumed that the movement of the mouse and the sensors are 100% accurate.





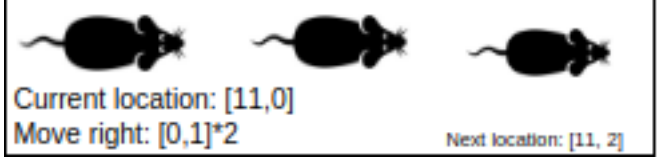
Current location: [11,0]
Move right: [0,1]

Next location: [11, 1]

Next location: [10,0]

Current location: [11,0]
Move up [-1,0]

The value of movement is a factor of the sumation between the coordenates and the delta [x,y]value of the direction



Algorithms and techniques:

Initial reflexion: The problem is to found the optimal route to the goal. If we know where the goal is and the composition of the walls we could apply the dynamic programming technique for calculating the value function and them construct a path following a policy, I learned this techniques from the course of robotics from Udacity.

Also we have to take in account that the time is in the score formula. In this sense exploration must be fast. But fast means lesser exploration, and lesser exploration may not find the optimal path.

General strategy for founding the optimal path:

In the first run we must explore, and for that I can use A star with an heuristic grid given. This will create a tendency to the goal in the exploration behavior.

But this heuristic grid will be optimistic because don't have in account the geography of the walls. Them apply dynamic programming technique for gather the value of each tile known as value function and the policy. Them reconstruct the path to the goal following the policy.

Above I explain in more detail the main algorithm.

For the benchmark models:

The benchmark models will use the same algorithm for exploration and for exploiting phase, I don't expect greater results in score on this models.

1. Random move
2. Random move with detection of dead end
3. Random move with detector dead end and visited counts

Random move: will take the direction to the next tile from a list of valid tiles. The valid tiles heres are any tile except the tiles out of grid and the tiles in wall direction. The selection of which tile is the next will be made by a randomize priority. This is: assigning a random number to a tuple composed by the next coordinates of the movement and the index of rotation, (random number, x, y, rotation index), after that sort the list of tuples and pop the next movement. For example:

rotation={'left':0,'front':1, 'right':2}

current position: 11,0

We have this two tuples: (2, 10,0,1), (1, 10, 1, 2). This two tuple represent move up, and move right. The first element of each tuple is the number assigned with randomness.

Random detector dead end: This model use the technique of movement explained above and an additional feature for detecting dead end path. When the robot fall into a dead end, will record the location of it in a rejected list of tiles and make the reverse movement.

Random detector dead end and visited checks: This model use the technique of movement explained above, also have the detector of dead end feature, and also will record

the number of visit to each location. It will use this memory for breaking ties when the randomized number assigned in each possible movement are equals.

Main algorithm for solving the maze:

For the exploring phase I will use the next algorithms:

1. A star with given heuristic, detector dead end, and visited counts

For the exploiting phase I will use:

1. Dynamic Programming techniques
2. Construct the optimal path following the policy

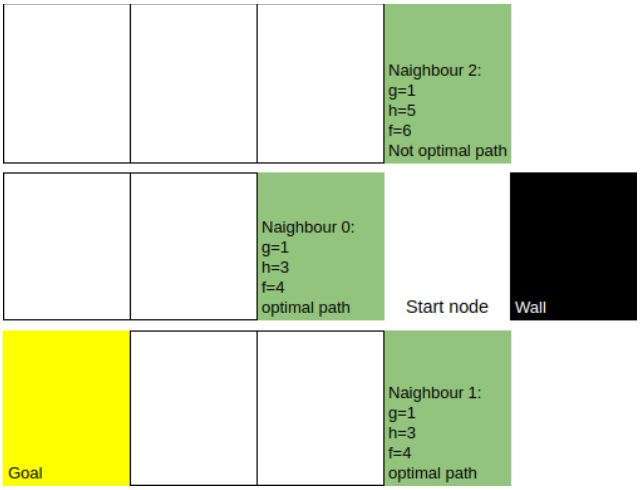
A Star with given heuristic, detector dead end, and visited count: A star is developed on the graph theory , and that defines a graph as a mathematical structure used for model relationship between object using nodes and arcs.[ii] In our case the nodes will be the possible location of the robot and the arcs will be the cost function in each node. Path-finding is the process of checking each node in a graph.The main characteristic of A star is that is focused to find an specific location, the goal.

This mean that will expand the fewest nodes possibles in the search. A search algorithm must make an informed decision on which node expand. For doing that A star use an evaluation function F(n) = G(n) + G(n)[iiiivvvi]. The G cost is the cost of reaching a determinate node from the source node. H cost is the heuristic value of the target node to the goal location. The heuristic come from information on the problem domain. In our case we will pass to the algorithm a grid with initial heuristic values as in the course of Robotic from Udacity. Is import to know that this initial heuristic will be very optimistic about the path because don't consider the location of the walls, but still useful because will guide the exploration near the goal.
Below the initial heuristic used for exploration purposes:

```
[[10  9  8  7  6  5  5  6  7  8  9 10]
 [ 9  8  7  6  5  4  4  5  6  7  8  9]
 [ 8  7  6  5  4  3  3  4  5  6  7  8]
 [ 7  6  5  4  3  2  2  3  4  5  6  7]
 [ 6  5  4  3  2  1  1  2  3  4  5  6]
 [ 5  4  3  2  1  0  0  1  2  3  4  5]
 [ 5  4  3  2  1  0  0  1  2  3  4  5]
 [ 6  5  4  3  2  1  1  2  3  4  5  6]
 [ 7  6  5  4  3  2  2  3  4  5  6  7]
 [ 8  7  6  5  4  3  3  4  5  6  7  8]
 [ 9  8  7  6  5  4  4  5  6  7  8  9]
 [10  9  8  7  6  5  5  6  7  8  9 10]]
```

The algorithm use two list for storing open nodes which are node not visited yet, and closed nodes already visited. In each visit made the algorithm will look for neighbor of the current node, run the evaluation function on them and choose the one with lower cost.
The algorithm could be described on higher level following the next steps[vii]:

1. Mark initial node as open and calculate F(start)
2. Select the open node n whose value of F is the smallest
3. If n is the goal terminate the algorithm
4. Otherwise mark n closed and search for neighbor of n



My implementation with A star also will have the detector dead end feature and also will leave a count on how many times visit each tile. Also is important to note that our A star algorithm will not check all four direction, instead will check only the direction where the sensor are pointing.

Exploiting phase:

Dynamic programing technique: The Dynamic programming approach create the optimal path from every location to the goal, this is called policy function, and it map states to actions. In our case will map coordinates [x, y] to move directions for reaching the goal. The technique consist in breaking down the problem in a recursive manner for obtaining the value function. In each step of updating the value function use the next formula:

$Q(x, y) = \min_{x' y'} Q(x', y') + step\_cost$

Which can be read as: taking the optimal neighbor, considering his value function and adding the step cost.

For example in the next images is represented a grid world of 5x5 with the value function in each tile, the black tile is a wall.

8	7		5	4
7	6		4	3
6	5	4	3	2
5	4	3	2	1
4	3	2	1	0

Them for founding the optimal policy we have to minimize the value function that leave us to the goal as in hill-climbing technique.

For example in the next image is represented the optimal policy for the 5x5 grid world:

>	v		v	v
>	v		v	v
>	v	>	>	v
>	>	>	>	v
>	>	>	>	Goal

Construct the optimal path following the policy: finally I will construct the optimal path following the policy created by the dynamic programing algorithm.

Benchmark:

I'll provide three random models for comparing to them, the first one use random move, the second is random move plus detection of dead end and third is random move with detection of dead end and visited count for braking ties. I don't expect greater results on this model and for that I will run several test on them and choose the best for comparing it to my main algorithm for solving the mazes.

METHODOLOGY:

Data preprocessing:

As the data of the maze and the sensors of the robot are provided and 100% accurate is not necessary to preprocessing any data.

Implementation:

Some of the important variables

```
self.heading = 'up'
self.location = [11, 0]
self.back_sense = 0
```

This are my principal variables in my implementation, changing them I'm changing location and the pointing of the heading of my mouse, that is my rotation and my move methods. The self.back_sense = 0 is a variable that I use for updating the information of walls behind the mouse, this means that the value of zero represent that there is a wall in the back of the mouse. The move method update this value.

Rotate method:

The rotate method work with the global variable self.heading and the dictionary dir_sensor. The dictionary has a key of heading direction of up, right, down and left. Inside each key we have a list of the sensors direction which are three because the mouse don't have sensor in the back.

I use this dictionary for changing the rotation which also have three option of rotation on -90, 0 and 90 degrees, or as I implemented in rotate method I called left, front and right.

```
dir_sensor = {'u':['l','u','r'], 'r':['u','r','d'], 'd':['r','d','l'], 'l':['d','l','u'],
              'up':   ['l','u','r'],  'right':['u', 'r','d'], 'down': ['r', 'd', 'l'], 'left': ['d','l','u']}
```

The input of the method is left, front or right strings. The output is -90, 0 or 90.

```
def rotate(self, direction):
    ''' direction is an int 0, 1, 2 that correspond to left front right of the current heading'''

    rotation_index = [-90, 0, 90]

    direction_index = {'left': 0, 'front':1, 'right':2}

    orientation = direction_index[direction]

    new_heading = dir_sensor[self.heading][orientation]

    self.heading = new_heading

    next_rotation = rotation_index[orientation]

    return next_rotation
```

Move method:

The move method use the self.heading of the robot and modify the self.location.

Calling the self.heading inside the dir_move dictionary will return us a tuple of the delta value of x and y to the correspondent direction of move. Them multiply it with the movement factor and them adding to the current location will give as the new location. In the exploration run I always use movement of one, in the exploiting phase I allow the randoms models and the main model to use multiple movement per step.

This method also check if the next location is inside the maze grid if the heading direction to move is free of wall before making the updating in the self.location.

Finally this method update the self.back_sense to one, because if we could go forward in the next tile behind we have not a wall.

dir_move = {'up':[-1, 0], 'right':[0,1], 'down': [1, 0], 'left':[0, -1],
'u':[-1, 0], 'r':[0, 1], 'd':[1, 0], 'l':[0, -1]}

```
def move(self, movement=1, exploration=False):
    x = self.location[0]
    y = self.location[1]

    delta_x = dir_move[self.heading][0]
    delta_y = dir_move[self.heading][1]

    x2 = x + delta_x*movement
    y2 = y + delta_y*movement

    if self.check_wall(self.heading) == False:

        if self.check_grid_boundry(x2,y2):

            self.location[0] = x2
            self.location[1] = y2
            self.back_sense = 1
    return movement
```

Now I would like to explain the construct binary tile method but first I need to explain check sensor method for better understanding.

Check sensor method: I use this method to transform the information received from the sensor in a binary information of zero or one. Them I use this for knowing when the mouse have wall in his front left or right.

```
def check_sensor(self, sensors):
        sensor_active = [0,0,0]

        for i in range(len(sensors)):
            if sensors[i] > 0:
                sensor_active[i] = 1

        return sensor_active
```

Construct binary tile method:

This method first call the check sensor method for transforming the sensor information in zeros or ones. Them according to the direction of the heading up, right, down or left create the number that represent the tile and store it in a grid called self.binary_tile. In the analysis part I explain in detail how each tile is composed. Is important to note that in this method we also use the self.back_sense variable that tell as if we have or not a wall behind.

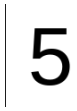
For example in the next tile construction when the heading is Up:

given the sensor [0,1,0]

given the self.back_sense = 1 (no wall behind)

left sensor 0 * 8 + front sensor 1 * 1 + right sensor 0 * 2 + self.back_sense*4 = 5

This tile:




```
 def construc_binary_tile(self, sensors):
     sensor_active = self.check_sensor(sensors)

     if self.heading == 'up' or self.heading == 'u':   ## left, front, right, back
         bin_tile = sensor_active[0]*8 + sensor_active[1]*1 + sensor_active[2]*2 + self.back_sense*4

     if self.heading == 'right' or self.heading == 'r':   ## left, front, right, back
         bin_tile = sensor_active[0]*1 + sensor_active[1]*2 + sensor_active[2]*4 + self.back_sense*8

     if self.heading == 'down' or self.heading =='d': ## left, front, right, back
          bin_tile = sensor_active[0]*2 + sensor_active[1]*4 + sensor_active[2]*8 + self.back_sense*1

     if self.heading == 'left' or self.heading == 'l': ## left, front, right, back
         bin_tile = sensor_active[0]*4 + sensor_active[1]*8 + sensor_active[2]*1 + self.back_sense*2

    self.binary_tile[self.location[0]][self.location[1]] = bin_tile
     return None
```

Reverse method: This method is for the reverse movement, it use the reverse dictionary for getting the inverse of the heading.

```
dir_reverse = {'u': 'd', 'r': 'l', 'd': 'u', 'l': 'r',
               'up': 'd', 'right': 'l', 'down': 'u', 'left': 'r'}
```

```
def reverse(self, movement):
    x = self.location[0]
    y = self.location[1]

    x2 = x + dir_move[dir_reverse[self.heading]][0]*movement
    y2 = y + dir_move[dir_reverse[self.heading]][1]*movement
    self.location[0] = x2
    self.location[1] = y2

    return -movement
```

Dead end detector method: this method is for detecting the dead end path, every time before making a move the robot check his tile. Only I put here a part of the code for the up heading. I use a set() object for storing the dead end locations.

For example If the head of the mouse is pointing right the only possible dead end is the number 8 value in the self.binary_tile. Also if in the tile in the back of the mouse is a 10 the mouse is in a deep dead end. For normal dead end the value for reverse is one for deep dead end the value for reverse is two.


```
def detect_dead_end(self):
    if self.heading =='right' or self.heading == 'r':
        if self.binary_tile[self.location[0]][self.location[1]] == 8:
            if self.binary_tile[self.location[0]][self.location[1]-1] == 10:
                self.dead_ends.add(tuple(self.location))
                self.dead_ends.add(tuple([self.location[0], self.location[1]-1]))
                self.reverse_value =2
                return True
            else:
                self.dead_ends.add(tuple(self.location))
                self.reverse_value =1
                return True
```

A star loop: I only put a part of the code of A star here for general explanation purposes. First call the check sensor method for knowing where we have walls, and in consequence which are the possible neighbors. The program loop in the three sensor and only allow to pass the index of the sensors that are actives. Them create the coordinates of the neighbor tile and check if is it inside the grid of the maze. Unpack the features of the neighbor and

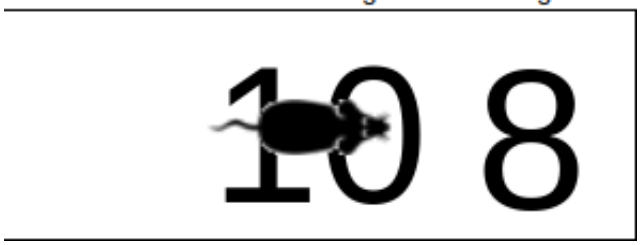
append it to the open list. Them sort the list, reverse it and pop one element for movement.

```
sensor_active = self.check_sensor(sensors)

        open_list = []

        for i in range(len(sensor_active)):
            if sensor_active[i] == 1:

                x2 = x + dir_move[dir_sensor[self.heading][i]][0]
                y2 = y + dir_move[dir_sensor[self.heading][i]][1]

                if self.check_grid_boundry(x2, y2):
                    g2 = self.g + cost
                    v = self.visited[x2][y2]
                    h = heuristic_grid[x2][y2]
                    f = h + g2
                    open_list.append([v, f, x2, y2, i])
```

Check memory dead end: this method is implemented for an specific situation in exploration. This method check when the robot is above a tile 5 if the upper tile is a 4 or if the lower tile is a 1.

Also check in tile 10, right an 8 or left a 2. This check allow the robot in exploration to early know the existence of a dead end path.

Check if the front tile is an eight for avoiding it




```
def check_memory_for_dead_end(self):
        x = self.location[0]
        y = self.location[1]
        ## if we are in 5 or 10 tile we must be alert becouse a dead end could be near, and for
prevention we try to remember the composition of the maze
        if self.binary_tile[x][y] == 5:
            try:
                ## Look in the upper direction
                x2 = x + delta[0][0]
                y2 = y + delta[0][1]
                ## Look in the lower direction
                x3 = x + delta[2][0]
                y3 = y + delta[2][1]

                if self.binary_tile[x2][y2] == 4:
                    self.dead_ends_second_list.add(tuple(self.location))
                elif self.binary_tile[x3][y3] == 1:
                    self.dead_ends_second_list.add(tuple(self.location))
                else:
                    pass
            except IndexError:
                print(' Tile dont visited yet we dont have it in memory')
```

Check wall method :

For the check wall method I use a list with the name allowed_up = [1, 3, 5, 7, 9, 11, 13, 15] and if heading is pointing up I check if the self.binary_tile in the current position is in the allowed list of tiles composition. For example 1 is in allowed list for heading up so the return of the method is False because there is no wall in the upper direction. The same is implemented for the others directions.

Check grid boundary method: this method return True if the location or the given coordinate is inside the grid bounds.

```
    def check_wall(self, heading, x=None, y=None):
        if x is None and y is None:
            x = self.location[0]
            y = self.location[1]
        ## return false if there is not wall in the direction of the heading
        if   heading == 'u' or heading == 'up':
            if self.binary_tile[x][y] in allowed_up:
                return False
            else:
                return True
    def check_grid_boundary(self, x2, y2):
        if (x2 >= 0  and x2 <= (self.maze_dim-1) and y2 >= 0 and y2 <=
(self.maze_dim-1)):
            return True
        else:
            return False
```

Check goal method: This method is for checking if a given x, y coordenates is in the goal bounds. If none is passed will check if the current position is in the goal bounds, also have a feature for checking if is the first time that we visit the tile and is in the limits of the goal.

```
def check_goal(self, x=None, y=None, check_visited=False):
        if x is None and y is None:
            x = self.location[0]
            y = self.location[1]

        if x in self.goal_bounds and y in self.goal_bounds:
            if check_visited:
                if self.visited[x][y] == 1:
                    return True
                else:
                    return False
            else:
                return True
        else:
            return False
```

Dynamic programing method: I learned this technique from the course of Artificial intelligence for robotics from Udacity. This method will set the value of the goal to zero and them loop over all the grids tiles and in each tile check all possible neighbors in the four directions. Creating the value of the tile checking if its lower than the existent value and setting the flag of change as True. This method do this until no change is reported. Update the value in the value grid end set the correspondent symbol for the policy.
Below is the code:

heading =['up', 'right', 'down', 'left'] policy_simbol = ['^', '>', 'v', '<']

```
def dinamic_value_generator(self, cost_step=1):
    value_to_goal = [[ 99 for col in range(self.maze_dim)]for row in range(self.maze_dim)]

     change = True
     while change:
        change =False

        for x in range(self.maze_dim):
            for y in range(self.maze_dim):

                if  self.check_goal(x,y):
                    if value_to_goal[x][y] > 0:
                        value_to_goal[x][y] = 0
                        self.policy[x][y] = '*'
                        change = True

                else:
                    for i, direction in enumerate(heading):
                        if self.check_wall(direction, x, y) == False:
                            x2 = x + dir_move[direction][0]
                            y2 = y + dir_move[direction][1]
                            if self.check_grid_boundry(x2, y2):
                                value2 = value_to_goal[x2][y2] + cost_step

                                 if value2 < value_to_goal[x][y]:
                                    change =True
                                    value_to_goal[x][y] = value2
                                     self.policy[x][y] = policy_simbol[i]
```

This method will create the next two grid:

```
--- Q_value----
[[33 34 35 34 33 32 31 32 99 99 99 99]
 [32 33 34 33 32 31 30 33 32 99 99 99]
 [31 32 35 34 27 28 29 30 31 99 99 99]
 [30 31 36 35 26 27 28 29  4  5 99 99]
 [29 28 27 26 25 26 27  2  3  6  9 10]
 [28 27 26 25 24  0  0  1  4  5  8 11]
 [29 28 25 24 23  0  0  2  3  6  7 10]
 [30 27 26 23 22 21 20  5  4 99 10  9]
 [29 28 29 28 21 20 19 18  5  6  7  8]
 [28 27 28 27 20 19 18 17 99 15 14  9]
 [29 26 25 26 21 20 17 16 15 14 13 10]
 [30 25 24 23 22 19 18 15 14 13 12 11]]
```

```
---Policy---
[['v' 'v' 'v' '>' '>' '>' 'v' '<' ' ' ' ' ' ' ' ' ' ']
 ['v' 'v' '<' '>' '>' '>' 'v' '^' 'v' ' ' ' ' ' ' ' ']
 ['v' 'v' '^' '^' 'v' 'v' '<' 'v' '<' ' ' ' ' ' ' ' ']
 ['v' '<' '^' '^' 'v' '<' 'v' '<' 'v' '<' ' ' ' ' ' ']
 ['v' 'v' '>' 'v' 'v' '<' '<' 'v' '<' '^' 'v' '<']
 ['>' '>' 'v' 'v' 'v' '*' '*' '<' '^' '<' 'v' '^']
 ['^' '^' '>' 'v' 'v' '*' '*' '^' '<' '^' '<' 'v']
 ['^' '>' '^' '>' '>' '>' 'v' '>' '^' ' ' '>' 'v']
 ['v' '^' '>' 'v' '>' '>' '>' 'v' '^' '<' '<' '<']
 ['>' 'v' '>' 'v' '>' '>' 'v' 'v' ' ' '>' 'v' '^']
 ['^' 'v' 'v' '<' '^' '^' '>' 'v' 'v' 'v' 'v' '^']
```

Construct path method:

First check if is not the star symbol that represent the goal. Them get the index of the direction marked on the policy from the dir_sensor dictionary. Use rotation method with this index for changing the heading of the robot. Then realize the movement in the new heading direction. The movement will be repeated until the symbol of the policy change. For example if in our grid policy we have > > > this mean three movement to the right.

policy_dict = {'^':'u', '>':'r', 'v':'d', '<':'l'}

policy_simbol = ['^', '>', 'v', '<']

```
def construct_optimal_path(self, sensors):
    x = self.location[0]
    y = self.location[1]
    movement_ = 1
    if self.policy[x][y] != '*':

        direction_policy = policy_dict[self.policy[x][y]]

        index_rotate = dir_sensor[self.heading].index(direction_policy)

        rotation = self.rotate(rotate_action[index_rotate])

        new_heading = self.heading

        while movement_ < 3:

            policy_symbol_ = self.policy[x][y]
            x += dir_move[self.heading][0]
            y += dir_move[self.heading][1]

            if self.policy[x][y] == policy_symbol_:

                movement_ +=1
            else:

                break

        self.move(movement=movement_)

        return rotation, movement_
    else:
        return 0, 0
```

The construct path method create the next grid:

```
----Path made---
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' 'M' '16' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' '15' '14' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '13' ' ' ' ' '12']
 ['1' '2' ' ' ' ' '5' ' ' '6' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' '7' '8' ' ' ' ' ' ' ' ']
 ['0' '3' ' ' ' ' '4' ' ' ' ' '9' ' ' ' ' '10' '11']]
```

Problems in the beginning:

At the beginning I try to translate the whole maze in zeros and ones, I try to decompose each tile in four tile, this force me to duplicate the length of the maze, this guide to some bigger problems for example: what is the real location of the robot, and other problem about the moving factor, how much moves are equal in my grid to the maze grid provided. Finally a desist from this idea. And implement the binary tile constructor.

For using the program:

You can use the next command.

python tester.py test_maze_01.txt a_star_dinamic_optimizer

Also after optimizer you can add a number that represent the requirement exploration percentage in the first phase.

python tester.py test_maze_01.txt a_star_dinamic_optimizer 80 verbo

also if you pass verbo after that the program will be much more expressive.

The next report will be returned :

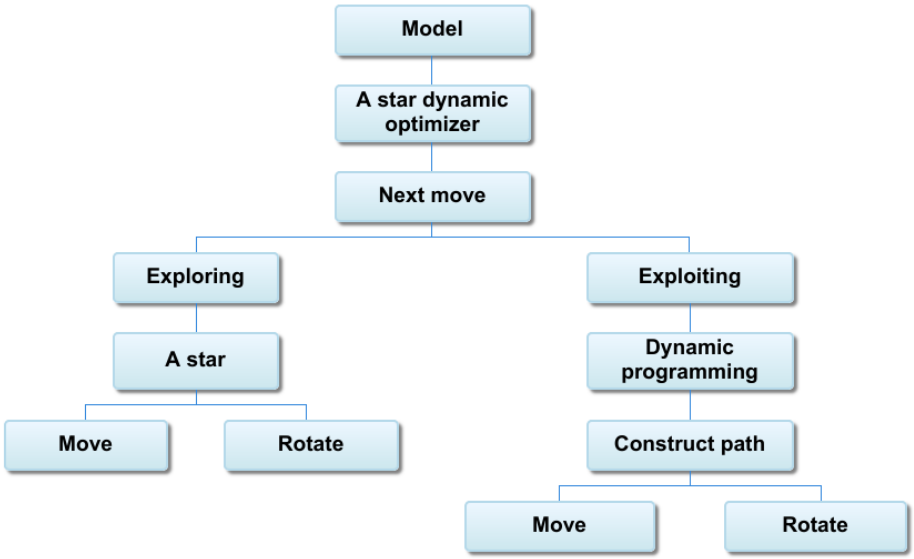
```
----First Heuristic used for exploration----
[[10  9  8  7  6  5  5  6  7  8  9 10]
 [ 9  8  7  6  5  4  4  5  6  7  8  9]
 [ 8  7  6  5  4  3  3  4  5  6  7  8]
 [ 7  6  5  4  3  2  2  3  4  5  6  7]
 [ 6  5  4  3  2  1  1  2  3  4  5  6]
 [ 5  4  3  2  1  0  0  1  2  3  4  5]
 [ 5  4  3  2  1  0  0  1  2  3  4  5]
 [ 6  5  4  3  2  1  1  2  3  4  5  6]
 [ 7  6  5  4  3  2  2  3  4  5  6  7]
 [ 8  7  6  5  4  3  3  4  5  6  7  8]
 [ 9  8  7  6  5  4  4  5  6  7  8  9]
 [10  9  8  7  6  5  5  6  7  8  9 10]]
Path made in exploration
[['>' 'v' 'R' 'v' '<' '<' '<' '<' ' ' ' ' ' ' ' ']
 ['^' 'v' '<' '>' '>' '>' 'v' '^' '<' ' ' ' ' ' ']
 ['^' 'v' 'v' '^' 'R' 'v' '<' '>' '^' ' ' ' ' ' ']
 ['v' '<' '>' '^' '>' '>' 'v' '^' '>' 'v' ' ' ' ']
 ['v' 'R' 'v' '<' 'v' '<' '<' '>' '^' 'v' 'v' '<']
 ['v' '<' 'v' '^' 'v' 'v' '>' '^' 'v' '<' 'v' '^']
 ['v' '^' 'v' '^' 'v' '>' '^' '^' 'v' '^' '<' '^']
 ['v' '^' '<' '^' '>' '>' 'v' 'R' 'v' ' ' 'M' '<']
 ['v' '^' '<' '<' '^' '<' '>' 'v' '>' '>' '>' '^']
 ['>' 'v' '>' 'v' '>' '^' '<' 'v' ' ' '>' 'v' '^']
 ['R' 'v' 'v' '<' '^' 'v' '^' 'v' 'R' '^' 'v' '^']
 ['R' '>' '>' '>' '^' '>' '^' '>' '>' '^' '>' '^']]
Binary Tile Grid dicover
[[ 6 12  4  6 10 10 14 14  0  0  0  0]
 [ 5  7 13  7 10 10 13  3 14  0  0  0]
 [ 5  5  5  5  4  6 11 14  9  0  0  0]
 [ 7  9  3  9  7 11 14  9  6 15  0  0]
 [ 5  4  6 12  7 10  9  6 13  5  6 12]
 [ 7 15 13  5  5  6 14 13  7 13  5  5]
 [ 5  5  7 13  5  3  9  3 13  7  9  5]
 [ 5  7  9  3 15 10 12  2 15  0  2 13]
 [ 5  3 10 12  3 14 11 12  7 10 10 13]
 [ 7 14 10 13  6 15 12  5  0  6 12  5]
 [ 5  5  6  9  5  5  7 13  4  5  5  5]
 [ 1  3 11 10  9  3  9  3 11 11 11  9]]
```

```
Visited grid made
[[2 2 3 1 1 1 1 1 0 0 0 0]
 [2 4 5 2 2 2 2 1 1 0 0 0]
 [2 3 1 1 1 2 2 1 1 0 0 0]
 [4 3 1 1 2 3 3 1 1 1 0 0]
 [3 4 3 3 3 2 2 1 1 1 1 1]
 [5 8 4 3 3 2 4 3 2 2 1 1]
 [3 3 4 3 3 2 2 2 3 1 1 1]
 [3 4 3 2 5 3 3 1 2 0 1 2]
 [3 2 2 2 2 2 3 2 2 1 1 3]
 [5 4 2 3 3 3 3 2 0 1 1 1]
 [3 2 2 2 2 1 3 3 1 1 1 1]
 [2 3 3 2 3 1 1 2 2 1 2 2]]
####### Exploit Report ######
Exploit Found Goal in location [5, 6] in step: 16
--- Q_value----
[[33 34 35 34 33 32 31 32 99 99 99 99]
 [32 33 34 33 32 31 30 33 32 99 99 99]
 [31 32 35 34 27 28 29 30 31 99 99 99]
 [30 31 36 35 26 27 28 29  4  5 99 99]
 [29 28 27 26 25 26 27  2  3  6  9 10]
 [28 27 26 25 24  0  0  1  4  5  8 11]
 [29 28 25 24 23  0  0  2  3  6  7 10]
 [30 27 26 23 22 21 20  5  4 99 10  9]
 [29 28 29 28 21 20 19 18  5  6  7  8]
 [28 27 28 27 20 19 18 17 99 15 14  9]
 [29 26 25 26 21 20 17 16 15 14 13 10]
 [30 25 24 23 22 19 18 15 14 13 12 11]]
---Policy---
[['v' 'v' 'v' '>' '>' '>' 'v' '<' ' ' ' ' ' ' ' ']
 ['v' 'v' '<' '>' '>' '>' 'v' '^' 'v' ' ' ' ' ' ']
 ['v' 'v' '^' '^' 'v' 'v' '<' 'v' '<' ' ' ' ' ' ']
 ['v' '<' '^' '^' 'v' '<' 'v' '<' 'v' '<' ' ' ' ']
 ['v' 'v' '>' 'v' 'v' '<' '<' 'v' '<' '^' 'v' '<']
 ['>' '>' 'v' 'v' 'v' '*' '*' '<' '^' '<' 'v' '^']
 ['^' '^' '>' 'v' 'v' '*' '*' '^' '<' '^' '<' 'v']
 ['^' '>' '^' '>' '>' '>' 'v' '>' '^' ' ' '>' 'v']
 ['v' '^' '>' 'v' '>' '>' '>' 'v' '^' '<' '<' '<']
 ['>' 'v' '>' 'v' '>' '>' 'v' 'v' ' ' '>' 'v' '^']
 ['^' 'v' 'v' '<' '^' '^' '>' 'v' 'v' 'v' 'v' '^']
 ['^' '>' '>' '>' '^' '>' '^' '>' '>' '>' '>' '^']]
```

```
----Path made---
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' 'M' '16' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' '15' '14' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '13' ' ' ' ' '12']
 ['1' '2' ' ' ' ' '5' ' ' '6' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' '7' '8' ' ' ' ' ' ' ' ']
 ['0' '3' ' ' ' ' '4' ' ' ' ' '9' ' ' ' ' '10' '11']]
```

Also you can use the next commands:
python tester.py test_maze_01.txt a_star_dinamic_optimizer 90

The last number is the percentage limit of exploration area in the first run, and can be any number. But the exploration run only end If we found the goal and reach the exploration limit. For better understanding about the logic of the implementation about the main algorithm here is a chart:



Refinement:

The process of refinement was developed as the model improve. I present the next models:

- Random move model
- Random move with dead end detector
- Random move with dead end detector and visited counts
- A star with given heuristics dead end detector and visited counterclockwise

```
def random_search(self, sensors, cost=1, d_end_detector=False, visited_count=False, exploration=False):
```

The random models is method that receive True or False flags for activation of his features: Also in the exploiting phase activate the movement capacity at the maximum of 3 tiles.

The general mechanism of all random models is creating a list of neighbor as in the a star algorithm. This list is created only where the sensors report 1 meaning that there is not wall in that direction. After that I generated a random number for assigning priority to the pop call.

```
for i in open_list:
    i.insert(0, random.randint(0, len(open_list)))
```

I insert in the tuple of the neighbor as is showed in the next piece of code:
But in my process of debugging the whole program I receive a lot of Error Index coming from the random models, thats because some times the list of possibles neighbor became empty. And for solve this I add the next piece of code for create random rotation and movement when the list is empty. In this way my randoms models always can call a rotation and a movement although with list of neighbor empty.
Here the sort of the list, the reversing and them calling pop if is possible. If the list is empty create a random action.

```
        open_list.sort()
        open_list.reverse()
        try:
            next_move = open_list.pop()
        except IndexError:

            random_action = rotate_action[random.randrange(0,3)]
            rotation = self.rotate(random_action)
            movement = self.move()
            return rotation, movement
```

This different features in the randoms models can be called using the next commands:
python tester.py test_maze_01.py random
python tester.py test_maze_01.py random_detector
python tester.py test_maze_01.py random_detector_visited

Also I was experimenting with the A star algorithm, and is possible to pass the Q value function as heuristic grid, and as expected with this more precise heuristic the A star also is capable of founding path to the goal.

This can be done with the next command:
python tester.py test_maze_01.py a_star_dinamic_a_star

Also If we pass a number after the selection of the model we change the exploration limit in the first run. For example:
python tester.py test_maze_01.py random_detector 60

Dead ends improvements:

Some features as the dead end detector was improve a little with the method check memory dead end. The dead end detector have a big role in my implementation, the method create a list of rejected tiles and check it before adding a neighbor to the list. This prevent me from dead end loops.

Improvement while adding features:

As I improve adding features to each model I add it to the exploration method and to the exploiting method. Here is the example of a part of the code of the exploration method:

```
def explore(self,sensors, model):
    if self.model[0:6] == 'a_star':
        rotation, movement = self.a_star(sensors, exploration=True)

    elif self.model == 'random':
        rotation, movement = self.random_search(sensors, exploration=True)

    elif self.model == 'random_detector':
        rotation, movement = self.random_search(sensors, d_end_detector=True,
exploration=True)

    elif self.model == 'random_detector_visited':
        rotation, movement = self.random_search(sensors, d_end_detector=True,
visited_count=True, exploration=True)

    self.step_count +=1

    return rotation, movement
```

The exploration method is responsible for passing the movement and the rotation to the next_move method that communicate our robot with the tester. First check if the binary tile was already constructed, if not construct it. Check if is in the goal the current position. And them the options of the models with the correspondent flags activated in his calls. The complete code of this explanation is in the file robot.py.

Improvement on the A_star for exploring: In my test I experience that the A star explore in wrong directions, this made that I only could found the goal only with a 80% of exploration ratio 242 steps(in maze 01) and that increased my score. Them I include a piece of code for considering a different cost for movement that implies turn right or turn left, and a lower cost for movements that do not require rotate the robot.

g_cost = {0: 1, 1: 0.99, 2: 1}

After that I was able to reach the goal in the exploration phase with only 115 steps.

Improvement on the reverse:

With the check walls behind method I always check the back of the mouse before doing reverse.

I**mprovement on the randoms models:**

At the beginning I allow the robot to choose the next movement and rotation in a pure random selection, but this make it lose many steps crashing to walls. Them I decide improve it, and make the robot choose randomly only in the directions that are free of walls.

Improvement in the randoms models with multiples movements:

I apply the check_possible_movement method, for knowing how much movements the robot can do without crashing with a wall, them I randomize the selection of the quantity of movement between zero and the maximum report by the method.

RESULTS

Model evaluation and validations:

I will run my evaluation in the next form. First I will check all the results of the main algorithm, that is a A star for exploring phase and a Dynamic programing technique for exploiting phase. I will made test with different limits of allowed percentage of exploration in the first run. That because the exploration have a big impact in the optimal path that the Dynamic programing method can can constructs.

For the randoms models I have very poor results as expected. For that I put no requirement of exploration in the first phase. And I run 10.000 test for each model in each maze for having a representative sample of each model. Also all the test use 1000 steps in the tester.

Main Algorithm results:

A Star with Dynamic Programming Results Maze 01:

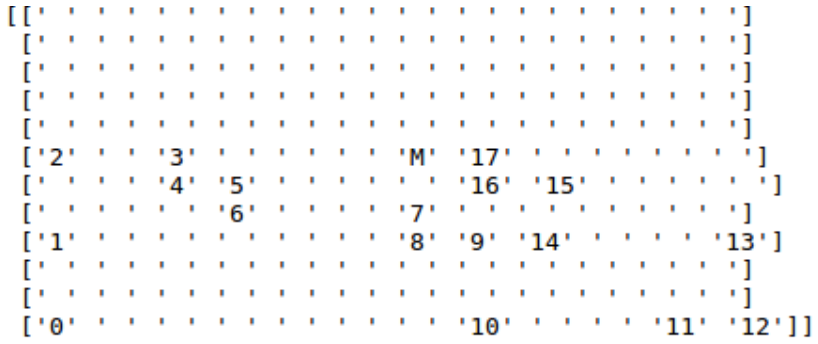
Maze 01							
Run 1				Run 2			
Found Goal	% of explore	Found goal in step	Total step in exploration	Found goal	Steps	path length	Score
True	min(65.97%)	115	115	True	24	40	27.83
True	70%	115	127	True	24	40	28.233
True	80%	115	171	True	18	34	23.700
True	90%	115	195	True	18	34	24.500
True	99%	115	296	True	20	30	33.200

The first row in the test is always the minimum exploration required for founding the goal in the first phase.

As we can see If we explore less the chances of founding a better path are lower, and if we explore a lot also increment our score by the exploring phase steps. Also is important to note that in the last row while we the path is shorter it took more step for run, and that because its have more turn right and turn left actions in the policy, this makes more steps and also

increment our score. For this test the better score was the third row. I know that is not the optimal path but is the better that my final implementation could do.

Third row path made:

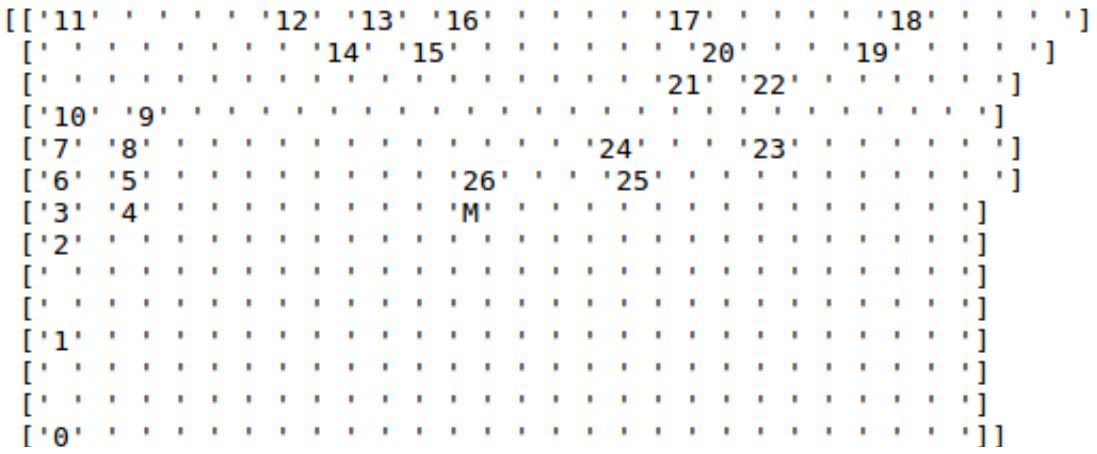


A Star with Dynamic Programming Results Maze 02:

Maze 02							
Found Goal	% of explore	Run 1 Found goal in step	Total step in exploration	Found goal	Steps	Run 2 path length	Score
True	min(50%)	118	118	True	29	47	32.933
True	70 %	118	178	True	27	43	32.933
True	80 %	118	222	True	27	43	34.400
True	90 %	118	406	True	27	43	40.533
True	99%	118	918	True	27	43	57.600

In the second maze above a 70% of exploration we found a path with 43 steps of length and our algorithm is capable of doing in 27 steps. While the first and the second row have the same score I will considerer better the second row because the steps that took accomplish the the path is shorter.

Second row path made:

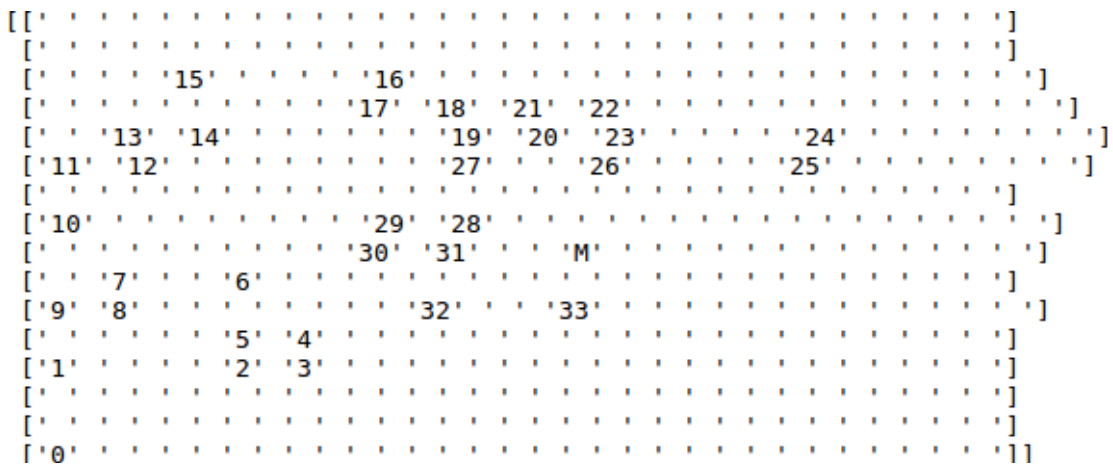


A Star with Dynamic Programming Results Maze 03:

Maze 03							
Run 1				Run 2			
Found Goal	% of explore	Found goal in step	Total step in exploration	Found goal	Steps	path length	Score
True	min(29.68 %)	93	93	True	34	55	37.100
True	70 %	93	323	True	34	55	44.767
True	80 %	93	354	True	34	55	45.800
True	90 %	93	453	True	25	49	40.1
True	99%	93	862	True	25	49	53.733

This test is very good because its explain clearly the relationship between exploration and optimization. While the algorithm found a better path in the fourth row it took to much exploration for founding it and that increment the score. For the other side the algorithm was very fast in founding the goal in the exploration phase, only 93 steps, 29% of exploration percentage in the biggest maze. In this case the better row was the fastest, while is not the shortest path, is better score.

First row path made:



Randoms models results:

For this analysis as I mentioned, I will not set a required percentage of exploration for passing the first phase, I leave the random algorithms run free and pass the first run when the goal is found.

From all the tests I will select the best score for later comparing with the best score of the main algorithm.

Also for each model I show the average of each feature and percentage of completing the two phases.

The generated data from each model have the next form:

found Goal	step found	step explore	%explore	Found goal	step found exploit	score
TRUE	161	161	62.500.000	TRUE	191	196.366.667
TRUE	360	360	67.361.111	TRUE	376	388.000.000
TRUE	283	283	59.027.778	TRUE	529	538.433.333
TRUE	252	252	61.111.111	TRUE	296	304.400.000
TRUE	130	130	50.000.000	TRUE	804	808.333.333

Random model on maze 01:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
277	58%	397	407.11	1534	10000	15.34%

As we can see the score is very poor in the general sample, over the total of 10000 test only 15% of the test accomplish the two runs and mark score. From all this test I found the best score was made by the test 1105, here the results for comparing:

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	61.80%	159	TRUE	40	45.3

Random model with dead end detector on maze 01:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
279	59.52%	395	405	1745	10000	17.45

As we can see the number are not much better. The only measurement that improve is the success in making the two runs, this time is 17.45 %, only 2% above the first random model. The best test in this sample is the test 144 here the results:

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	63.88%	303	TRUE	40	50.1

Here the number are very similar to the first random model, but spend more steps in the exploration phase and that increment the score. For this model I expect a better score comparing it with the first random, but it seems that the dead end feature that I add to the model do not compensate the randomness of the whole model.

Random model with dead end detector and visited count on maze 01:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
274	59.06	390	400	1730	10000	17.30

Again the results are not great as expected. Now I see that the randomness in the algorithms is much heavier than the features that program in them. I select the test 972 from the sample for comparing purposes, here the results.

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	50%	163	TRUE	39	44.43

Random model maze 02:

As seen below in a bigger maze my randoms algorithms perform with higher scores and the percentage of success of the test is minor.

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
277	49.28	449	448	880	10000	8.8%

Below the numbers of the test 443:

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	41.83%	143	TRUE	75	79.76

Random model with detector of dead end maze 02:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
256	49.36	466	474	591	10000	5.90

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	39.79	85	TRUE	99	101.83

Random model with dead end detector and visited count maze 02:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
273	50.86	419	428	128	10000	1.28

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	76.53	425	TRUE	89	103

As we can see as bigger the maze worse is the score. The features of detecting dead end a counting visited do not affect positively the score. The general success in making the two runs also decrease.

Random model maze 03:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
345	51.89	388	400	246	10000	2.45%

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	49.21	403	TRUE	53	66.43

Random model with dead end detector maze 03:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
286	45.87	256	266	46	10000	0.46

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	43.75	218	TRUE	77	84.26

Random model with dead end detector and visited count maze 03:

Averages						
Run 1		Run 2		test success %		
found goal in step average	average explore %	found goal in step	average score	made score	total test	made score %
325	50.92	292	303	53	10000	0.53

Best test					
Run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
TRUE	64.06	753	TRUE	72	97.1

As I can see as the maze is bigger the percentage of success in doing the two runs is lower. The numbers of the three randoms model on the third maze are the worst, this is logic because of the bigger shape of the maze.

Justifications:

Clearly the main model that use the A star for exploring and the Dynamic programming technique for exploiting is the better of all models in all mazes. Here the resume of all tests.

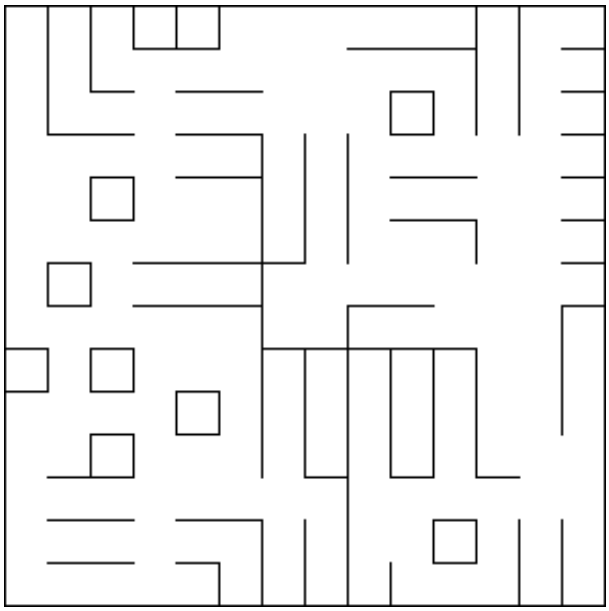
Algorithm	maze	Steps in run 2	Score
A star with Dynamic programming	1	18	23.700
A star with Dynamic programming	2	27	32.933
A star with Dynamic programming	3	34	37.100
Random	1	40	45.3
Random	2	75	79.76
Random	3	53	66.43
Random with dead end detector	1	40	50.1
Random with dead end detector	2	99	101.83
Random with dead end detector	3	77	84.26
Random with dead end detector and visited count	1	39	44.43
Random with dead end detector and visited count	2	89	103
Random with dead end detector and visited count	3	72	97.1

With respect to the main algorithm I can say that is a solid solution to the mazes, and also is fast in founding the goal. At the beginning of the project I was founding the goal using many steps of the exploration phase. After the refinement of the program the algorithm was able to found the goal much faster in the exploration phase. Also the founded path to the goal in the exploiting phase report a strong score.

CONCLUSION

Free form of visualization:

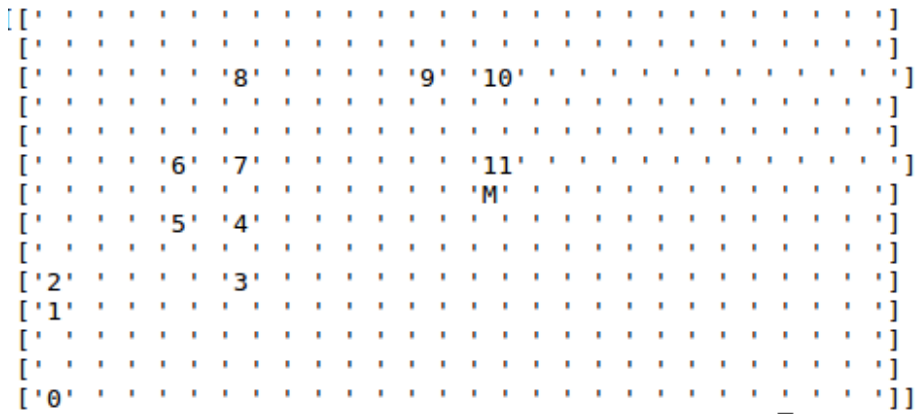
Here is maze that I design, I add some loops and several dead end paths in a 14X14 square:



The entrance to the center of the goal is in the upper right section of the maze that will obligate to our exploration algorithm to turn around all over the maze to found the goal, also there are several dead ends and loops to avoid.
Here the results:
The A star algorithm have to make 104 steps for founding the goal, and the Dynamic programming technique create a path with 12 steps.

Maze 04					
run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
yes	33 %	104	yes	12	15.46

And the path made:



Random move:

Maze 04					
run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
yes	33 %	169	yes	332	337

Random with detector of dead end:

Maze 04					
run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
No	17.85%	n/a	No	12	n/a

Random with detector of dead end and visited count:

Maze 04					
run 1			Run 2		
Found Goal	% of explore	Found goal in step	Found goal	Steps	Score
no	9.69%	n/a	no	n/a	n/a

As we can see the performance of the randoms models are very poor.

Reflections:

The whole project represent a lot of challenges for me. I divide the process of solving the problem in several parts. At the beginning I inform my self as much as possible about path finding algorithm and the theory behind this type of problems . This help me to understand the problem in a highest level. Them I begin with the programming work. I decide to create a class robot and I wanted to be able to switch on or off features for the algorithms. I also wanted to make the common method to all algorithm in a separate method for being able to reuse it. My general approach in the programming task was to write a modular class as easy to read as I could do.

At the beginning my bigger challenge was understand the codification about the maze and make my robot capable of interpreting his sensors. After that I focus on writing the move and rotation method that later I will reuse on the algorithms, this save to re write common task to all algorithms. Also applied this modularity concept to the the dead end detector and all other checks method that belong to the class.

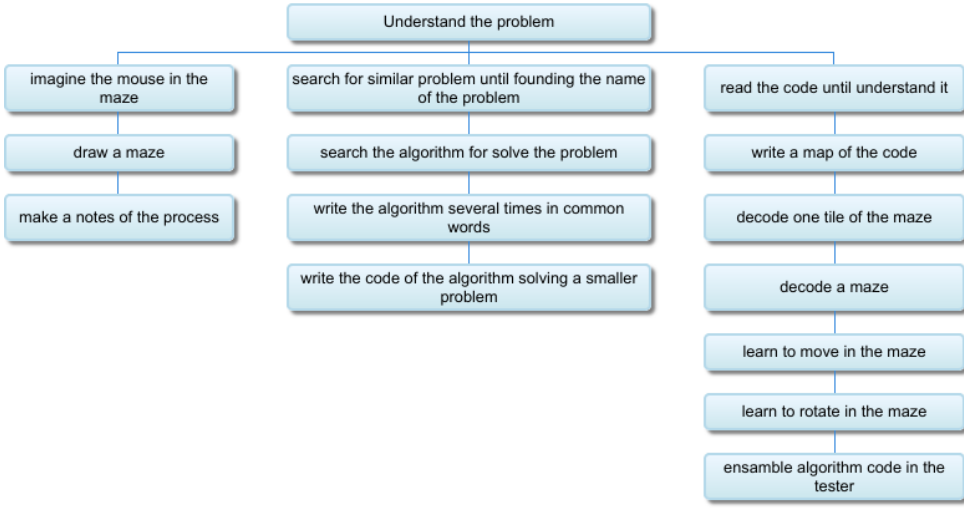
After my robot being able to move and rotate I create a very basic random decision maker to drop it in the maze and see what happens, obviously the performance was very poor, but this make me gain intuition about the possibles problems.

Them I write the A star algorithm for exploring and in his test I could found the dead end and the loops problem, that I solve writing the methods for this specific situations. The general process of writing, testing and debugging my code was accompanied with several logs prints and grids for following the robot behavior. Only seen this information I was able to understand and solve problems on the code and the problem itself.

The project itself make learn the tight relationship between exploring and optimization, make me think that if we explore less the robot don't be able to found the best path. But also exploring a lot is using the time.

I found the Dynamic programming technique very useful because is capable of founding the better path from all the position to the goal but using a high computational cost. I believe that my final robot class make a good work, founding the goal fast and choosing an reasonable optimal path, also I believe that is easy to read and to follow the logic.

Here go a map of some task that help me to solve the problem. The fourth task of the second column I have learned from the course from Udacity cs373.



Improvements

In a continuous space my robot sensors will report a distance and an angle, and also have and error ratio from the sensor information. I think that it would be interesting to try to apply the Kelman particles filter that I learned from the course of artificial intelligence for robotic in Udacity[viii]. The whole concept of dieing particles is amazing. The error in the sensor is the noise. In a continuous space we could treat the problem in a more statistics point of view because we can't be 100% sure of everything. This is the simultaneous localization and mapping problem[ix]. The distance will be an approximation. The movement will consider the velocity. The shape of the robot I can use a bicycle robot that only can rotate the front wheels. For the heuristic to a point we could use the rooted euclidean distance. If we have to make the robot what I most think about is what components I would need, a motor and and drivers, the wheels, a frame, a micro controller Arduino may be and shield for the drivers of the motors, batteries, an ultrasonic sensor and other things to be known.

If the robot is a circle I will have to adapt my robot class to consider the location as the center of the circle. To that point I will add the radius of the circle and include it on the location group, as being the limits of the robot shape. If the walls have thickness I most subtract that value from the tile possible location. This are my initial intuitions, the problem in a continuous space open a new world of challenges.

Overall this project force me to learn more and I'm glad for that.

Thank you.

i https://en.wikipedia.org/wiki/Micromouse

ii https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

iii https://www.redblobgames.com/pathfinding/a-star/introduction.html,

iv https://en.wikipedia.org/wiki/A*_search_algorithm

v http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

vi https://udacity.com/courses/cs373/

vii https://www.cs.auckland.ac.nz/courses/compsci709s2c/resources/Mike.d/astarNilsson.pdf

viii https://udacity.com/courses/cs373/

ix https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping