Machine Learning Capstone Project Report Robot Motion Planning

Prakash Rai

September 17, 2017

0.1 Project Overview

This project belongs to Motion Planning domain. Motion Planning is a term used in robotics for the process of breaking down a desired movement task into discrete motions that satisfy movement constraints and possibly optimize some aspect of the movement[1]. Motion planning has applications in many important domains like: Self driving cars, Robot navigation, automation, robotic surgery etc.

It is inspired by Micro-mouse competition. Micro-mouse competition is an event where a small robot attempts to navigate an unfamiliar maze.

We will be using python to implement virtual robot which will try to achieve same objectives on a virtual maze.

Udacity provides some starter code for this problem [2]. They are:

- **robot.py** - This script establishes the robot class. This is the only script that you should be modifying, and the main script that you will be submitting with your project.
- **maze.py** - This script contains functions for constructing the maze and for checking for walls upon robot movement or sensing.
- **tester.py** - This script will be run to test the robots ability to navigate mazes.
- **showmaze.py** - This script can be used to create a visual demonstration of what a maze looks like.
- **test_maze_##.txt** - These files provide three sample mazes upon which to test your robot. Feel free to create your own mazes using the specifications above.

0.2 Problem Statement

Primary purpose of this project is to program a robot which in a given maze, can find a path between two given locations, which satisfies following constraints-

- efficient, if not optimal,
- satisfies all surrounding constraints.

[1]Definition taken from Wikipedia.

[2]https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5IUM5OpAXJYiutFuSIjTzV_E-vdE/pub

In micro-mouse competitions, generally starting location is bottom left corner of maze and goal is center area of maze.

From above statement, it can be clearly seen that the above problem is a minimization problem under some constraints, hence it can be expressed in mathematical terms. This problem can be reproduced by reproducing the environment described below. Metric to measure complexity of problem and solution, could be **number of steps**, where each step can have movement of at most 3 spaces. More details about metrics is given in **Metrics** section.

The robot generally starts in a corner and must find its way to a designated goal. The robot is allotted two attempts with the maze - the first attempt is an exploratory attempt where the robot will maps out its environment in it's memory. The second attempt is where the robot will attempt to traverse an optimal route to the goal area as fast as possible.

0.2.1 Datasets and Inputs

Inputs will be:

- Maze
- Current position
- Destination

No data preprocessing will be required in this case, because data is already in a state from where it can be imported directly.

0.2.2 Possible Solution

In layman terms, the solution can be explained as follows. The robot mouse may make multiple runs in a given maze. In the first run, the robot mouse tries to map out the maze to not only find the center, but also figure out the best paths to the center. In subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned [3].

This problem can be solved using search algorithms like grid search algorithm, interval based search, reward based search, sampling based algorithms and geometric search.

Some specific algorithms to solve this problems are: BFS, DFS, A-star search, Dijkstra or Uniform Cost Search Algorithm.

0.3 Evaluation Metrics

A potential metric for this problem will be weighted sum of number of steps in path exploration and number of steps in path optimization. Exploration will be

[3]Whole paragraph taken from *MLND Capstone Project Description - Robot Motion Planning.*

given less weight than optimization.

$$score = a * steps\_in\_path + b * steps\_in\_maze\_exploration \tag{1}$$

where a = 1 and b=$\frac{1}{30}$. [4]

For example: If there are 150 exploration steps and 15 optimization steps, where each step has a movement of max 3 spaces, then score will be 15+$\frac{150}{30}$=20

I have chosen this metric because it will always give a score between lower benchmark and upper benchmark model[5] if some exploration constraint [6] is followed.

This metric is quantifiable as well as measurable.

0.4 Data Exploration

For this problem, only dataset we have is *testcase* files. Three txt files were given.

- test_maze_01.txt
- test_maze_02.txt
- test_maze_03.txt

0.4.1 Maze specifications

The maze exists on an n x n grid of squares, n even. The minimum value of n is twelve, the maximum sixteen. Along the outside perimeter of the grid, and on the edges connecting some of the internal squares, are walls that block all movement. The robot will start in the square in the bottom- left corner of the grid, facing upwards. The starting square will always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the center of the grid is the goal room consisting of a 2 x 2 square; the robot must make it here from its starting square in order to register a successful run of the maze.

Mazes are provided to the system via text file. 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 will be n comma-delimited numbers describing which edges of the square are open to movement. Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s

[4]Values are assigned intuitively
[5]Explained in benchmark section
[6]explained in explanation subsection of benchmark model section

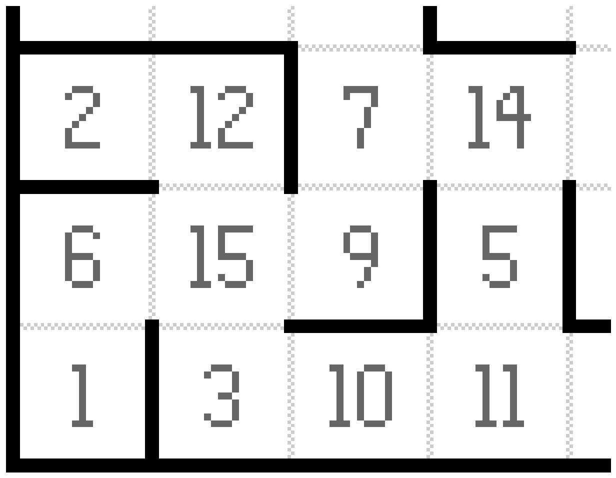


Figure 1: Example maze

register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom

$$(0 * 1 + 1 * 2 + 0 * 4 + 1 * 8 = 10). \tag{2}$$

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.

General wall configuration is shown in Figure 2.

0.4.2 Robot Speccations

The robot can be considered to rest in the center of the square it is currently located in, and points in one of the cardinal directions of the maze. The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor; for example, in its starting position, the robots left and right sensors will state that there are no open squares in those directions and at least one square towards its front. On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units. It is assumed that the robots turning and movement is perfect. If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robots new location and/or orientation to start the next time unit.

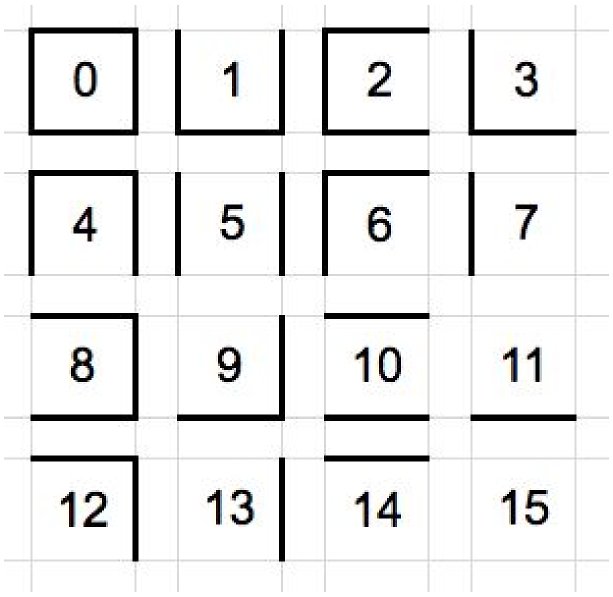


Figure 2: Walls visualization

More technically, at the start of a time step the robot will receive sensor readings as a list of three numbers indicating the number of open squares in front of the left, center, and right sensors (in that order) to its next_move function. The next_move function must then return two values indicating the robots rotation and movement on that timestep. Rotation is expected to be an integer taking one of three values: -90, 90, or 0, indicating a counterclockwise, clockwise, or no rotation, respectively. Movement follows rotation, and is expected to be an integer in the range [-3, 3] inclusive. The robot will attempt to move that many squares forward (positive) or backwards (negative), stopping movement if it encounters a wall.

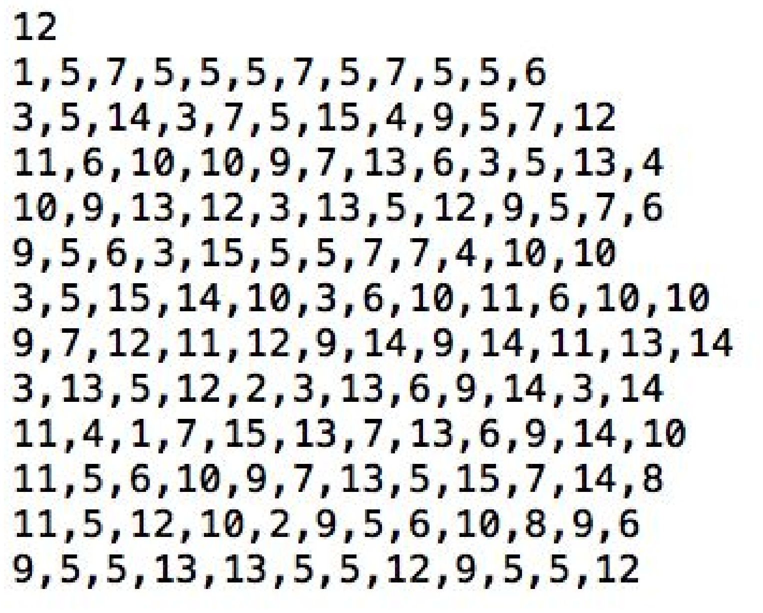
It is assumed that data given in text file is absolutely correct, adequate and no noise is present. Hence, no data preprocessing is required for this problem.

0.4.3 Exploring test_maze_01

Raw content of test_maze_01.txt is shown in Figure 3.

Graphical test_maze_01 is shown in Figure 4

Note that bottom left corner of image will correspond to (0,0) co-ordinate in test file.



```
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
```

Figure 3: Raw test maze 01

0.5 Exploratory Visualization

Following matrices have been used in exploration and optimization run:

- Wall Map: Stores location of walls surrounding each cell. Storage technique similar to *test_maze_01.txt* is used. Initialized with 0. A detailed explanation on algorithm is given in *Mapping maze into robot memory* subsection of *Implementation* section
- Traversal Grid: Stores number of times a cell has been traversed. Initialized with 0.
- Heuristic Grid: Stores Manhattan distance between a given co-ordinate and goal area cell(s). Initialized with Manhattan distance heuristic value. Remains constant throughout the code.
- Path Value: Stores actual distance(in terms of number of steps) between a given co-ordinate and goal area cell(s). Initialized with 99.
- Policy Grid: Stores optimal direction at each cell to reach the goal. Initialized with ' '(blank space).

After successful completion of exploration trial, states of *wall_map* matrix, *heuristic_grid* matrix, *traversal_grid* matrix and *path_value* matrix are shown in Figure 5, Figure 6, Figure 7 and Figure 8 .

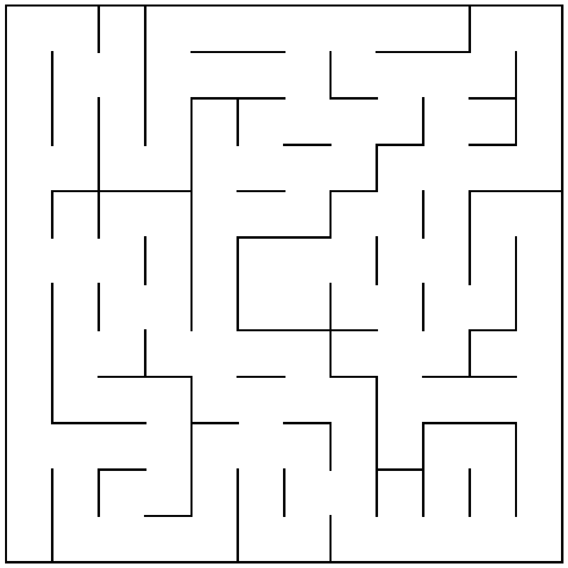
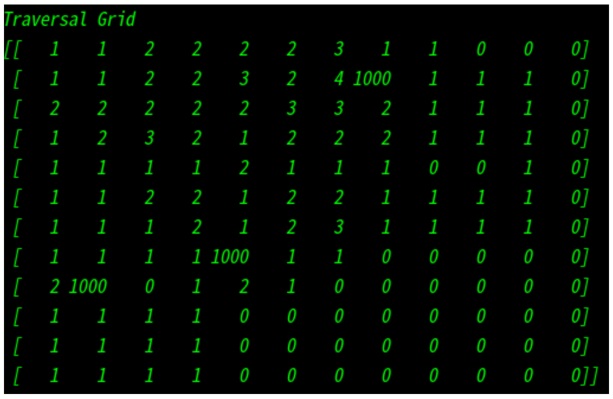
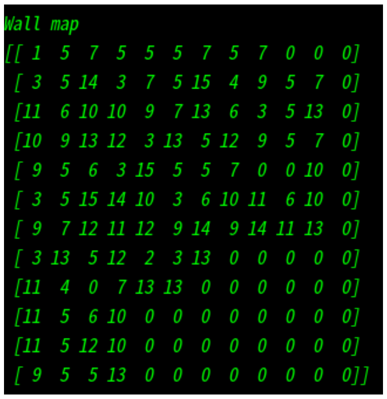


Figure 4: test maze 01 GUI



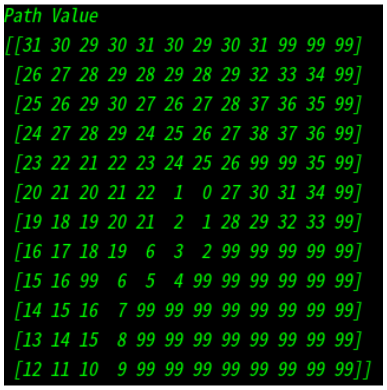
```
Traversal Grid
[[   1    1    2    2    2    2    3    1    1    0    0    0]
 [   1    1    2    2    3    2    4 1000    1    1    1    0]
 [   2    2    2    2    2    3    3    2    1    1    1    0]
 [   1    2    3    2    1    2    2    2    1    1    1    0]
 [   1    1    1    1    2    1    1    1    0    0    1    0]
 [   1    1    2    2    1    2    2    1    1    1    1    0]
 [   1    1    1    2    1    2    3    1    1    1    1    0]
 [   1    1    1    1 1000    1    1    0    0    0    0    0]
 [   2 1000    0    1    2    1    0    0    0    0    0    0]
 [   1    1    1    1    0    0    0    0    0    0    0    0]
 [   1    1    1    1    0    0    0    0    0    0    0    0]
 [   1    1    1    1    0    0    0    0    0    0    0    0]]
```

Figure 5: Traversal grid for *test_maze_01.txt*. 0 means those cells are not explored. 1000 means dead end.



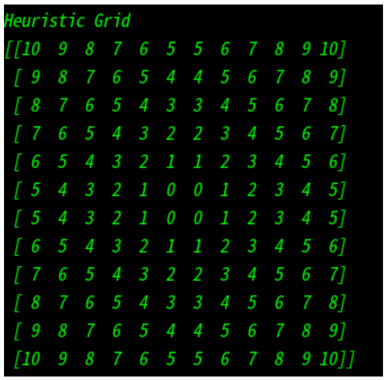
```
Wall map
[[ 1  5  7  5  5  5  7  5  7  0  0  0]
 [ 3  5 14  3  7  5 15  4  9  5  7  0]
 [11  6 10 10  9  7 13  6  3  5 13  0]
 [10  9 13 12  3 13  5 12  9  5  7  0]
 [ 9  5  6  3 15  5  5  7  0  0 10  0]
 [ 3  5 15 14 10  3  6 10 11  6 10  0]
 [ 9  7 12 11 12  9 14  9 14 11 13  0]
 [ 3 13  5 12  2  3 13  0  0  0  0  0]
 [11  4  0  7 13 13  0  0  0  0  0  0]
 [11  5  6 10  0  0  0  0  0  0  0  0]
 [11  5 12 10  0  0  0  0  0  0  0  0]
 [ 9  5  5 13  0  0  0  0  0  0  0  0]]
```

Figure 6: Wall map for *test_maze_01.txt*. 0 means those cells are not explored. Binary representation of other values give location of wall surrounding that cell.



```
Path Value
[[31 30 29 30 31 30 29 30 31 99 99 99]
 [26 27 28 29 28 29 28 29 32 33 34 99]
 [25 26 29 30 27 26 27 28 37 36 35 99]
 [24 27 28 29 24 25 26 27 38 37 36 99]
 [23 22 21 22 23 24 25 26 99 99 35 99]
 [20 21 20 21 22  1  0 27 30 31 34 99]
 [19 18 19 20 21  2  1 28 29 32 33 99]
 [16 17 18 19  6  3  2 99 99 99 99 99]
 [15 16 99  6  5  4 99 99 99 99 99 99]
 [14 15 16  7 99 99 99 99 99 99 99 99]
 [13 14 15  8 99 99 99 99 99 99 99 99]
 [12 11 10  9 99 99 99 99 99 99 99 99]]
```

Figure 7: Path Value for *test_maze_01.txt*. 99 means those cells are not explored. 0 means goal location.



```
Heuristic Grid
[[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]]
```

Figure 8: Heuristic grid for *test_maze_01.txt*. Goal locations are marked with 0.

0.6 Algorithms and Techniques used

Following major algorithms were being used in the implementation:

- A-Star Search,
- Dynamic Programming.

0.6.1 A-Star Search

Given a graph, A-star search is used to find efficient, if not optimal path, between two given nodes. For jumping from current node to next node, it uses priority queue data structure, where priority is decided by sum of edge weight and a heuristic function. Heuristic function is based on some heuristic, whose primary motive is to: given the current node, determine the next immediate node which minimizes some metric between source and destination. The node which minimizes the most is given higher priority than other nodes. Reason behind using A* search is that is exploits domain knowledge of problem.

In this problem, Manhattan Distance heuristic has been used.

0.6.2 Dynamic Programming

Given a map of the maze and a goal location, Dynamic Programming will given an optimal path from any possible starting location. For any cell in the maze, Dynamic Programming will output an optimum action, known as the Policy.[7].

It is based on a principle that shortest path between two different nodes, **A** and **B** in a graph can be written as union of shortest path between **A** and **C** and shortest path between **C** and **B**. That's why it calculates shortest distance between all nodes and goal area.

0.6.3 Justification

I have used heuristics to speed up maze exploration and Dynamic Programming to speed up 2nd run i.e. maze optimization.

In most of real world cases, A-star works faster than other shortest path algorithms which don't use heuristics(For example, Dijkstra, BFS, DFS etc). Due to this, I have used A-star with slight modification. In case space complexity becomes an issue, we can use Beam search, which drops off some nodes who have lower probability of being next node.

DP method is very computationally involved in terms of space and very fast in terms of time complexity.In our case grid size is small, so space complexity doesn't matters much, but time complexity does. Due to this reason, I have used it.

[7]Definition taken from Udacity Introduction to Artificial Intelligence course https://www.udacity.com/course/artificial-intelligence-for-robotics–cs373

0.7 Benchmark Model

Benchmark score can be weighted average of number of steps taken on optimal path to reach to the goal and number of steps required to traverse the whole maze in order to find optimal path. It will provide an upper bound and lower bound on performance of a model.

Mathematically, it can be expressed as

$$upper\_bound\_score = a * steps\_in\_optimal\_path + 5 * b * steps\_in\_optimal\_maze\_exploration \tag{3}$$

$$lower\_bound\_score = a * steps\_in\_optimal\_path + b * steps\_in\_optimal\_maze\_exploration \tag{4}$$

where a = 1 and b = $\frac{1}{30}$ [8]. Note that, above equation will hold true only when 100 % exploration is allowed. In case 100% exploration is not allowed, we have to decrease number of steps in optimal maze exploration accordingly.

Assuming that dimension of maze is n x n, number of steps in optimal maze exploration will be n^2 which is same as dimension of the maze.

Also note that number of exploration trials are limited to 1000.

For maze 1, optimal number of steps for reaching the center is 17 and optimal number of steps for maze exploration is 144, if 100% exploration is allowed. Upper_bound_score should be $17 + \frac{5*144}{30} = 41$. If 70% exploration is allowed, then Upper_bound_score should be $17 + \frac{5*144*0.7}{30} = 33.8$. Similarly, lower bound score will be $17 + \frac{144}{30} = 21.8$ if 100 % exploration is allowed and 20.5 if 70% exploration is allowed.

0.7.1 Explanation

Any implementation should explore whole maze by visiting each cell at least one time and at most five times[9]. Exactly five times corresponds to upper bound case and exactly one time corresponds to lower bound case. If number of steps for exploration is not in these bounds, then our model is not good.

0.8 Data Exploration

It is assumed that data given in text file is absolutely correct, adequate and no noise is present and sensor readings are always correct. Hence, no data preprocessing is required for this problem.

[8]Values are assigned intuitively

[9]Statistical Inference from past experience and little bit research.

0.9 Implementation

This problem can be broken into following subproblems.

- Mapping the maze in robot memory.
- Robot movement while maze exploration.
- Searching for fastest path.

0.9.1 Mapping maze into robot memory.

As mentioned in exploratory visualization section, information about maze grid is stored in *wall_map* matrix. This matrix stores information in similar way as *test_maze_##.txt* file. Algorithm for storing information is:

1. Get list of sensor readings.
2. If sensor reading is greater than 0, then make it's value 1, else make it 0.
3. If robot is moving backwards, store that value in some variable. and append it to sensors list. Here, a instance variable *down* is used.
4. Using *wall_conversion_matrix_dict*, convert new sensor reading list to decimal number.
5. Store decimal value at corresponding cell location in *wall_map* matrix.

0.9.2 Robot Movement while maze exploration

Robot Movement is determined by using a modified version of A-star search algorithm. A-star search has been introduced in *Algorithms and Techniques use* Section.

For path exploration, following algorithm is used. It proceeds as follows:-

1. Given current location coordinates and sensor readings, it retrieves all possible locations which robot could visit. It maintains a list *possible_moves()*
2. Then, it checks whether retrieved location is dead end or not.
3. If not, then it appends following information about retrieved cell location in *possible_moves()*.
    - Number of times cell has been traversed. This information is being stored in *grid_traversal* matrix.
    - Manhattan distance between current cell and goal area.
    - Current coordinates.
    - Sensor readings.

4. If it is dead end, then it checks all possible locations to move from that cell.
5. If that cell has no opening other than which leads to dead end, then that location is marked as DEAD_END and robot is moved backwards by 1 step.
6. If that cell has at least 1 opening, which doesn't leads to dead end, then step 3 is executed for that retrieved location.
7. If step 5 doesn't occurs, then from *possible_moves* list, it chooses the cell which has been least traversed. In case of a tie, heuristic grid is consulted and location with minimum heuristic value is selected. Information about heuristic grid is stored in *heuristic_grid* matrix.
8. *heuristic_grid*, *traversal_grid* and *policy_grid* are updated accordingly.

Initially, I tried to use original A-star search algorithm where priority is decided by sum of value of corresponding *path_value* coordinates and *heuristic_grid*. But, this didn't worked out because initial path value of all cells was 99, which proved to be a bias for undiscovered nodes. I tried several weighted average of above two quantities, but none of them worked better than current model.

0.9.3 Searching for fastest path.

Once exploration phase is over, dynamic programming is used to determine shortest distance between a given cell and goal. Algorithm for finding fastest path is:[10]

1. Initialize all path_values to 99.
2. Set path_values of goal location to 0.
3. Go through all the cells and make sure that each one has a value that is just one more than its smallest accessible neighbour.
4. Repeat step 3 until condition is satisfied for all cells.

State of matrices after successful completion of exploration trial is shown in Figure 5, Figure 6, Figure 7 and Figure 8.//

This sums up major steps taken in implementation of this project. Some improvements were done in order to improve the performance of some functions, which are mentioned in *Refinement* section.

[10]Inspired by Udacity Intro to AI course and algorithm given on http://www.micromouseonline.com/micromouse-book/mazes-and-maze-solving/solving-the-maze/

0.10 Refinement

For refining performance and code maintenance, following steps were taken:

- **Breaking the code into multiple files:-** Primary objective of this step was to break the code into subproblems, which can be solved, fine-tuned, tested and maintained separately. Measures were taken to make this solution loosely coupled to any other entity, so that it can easily be used with other models.
- **Maintaining a separate file for constants and global lists and dictionaries:-** This allows logic to be clear and separable from variable description(s). It improves code readability. Also, for using global variables, one has to import a single file which doesn't contains any methods, hence there is no risk of undesired function call by mistake.
- **Dead end marking:-** Initially, my maze exploration algorithm didn't had any way of telling the robot that current way will lead to a dead end. As a result, robot always fell into endless loop once it encounters a path which leads to dead end. After some debugging and hit and trial, I introduced Step 4, Step 5 and Step 6 in maze exploration algorithm, which worked on all test cases and produced efficient, if not optimal, results.
- **Removed redundant directions in global variables:-** There were many undesired aliases for directions in *tester.py*. For example: **'u'** for up , **'r'** for right etc. They were causing confusion and having code readability issues. Also, they were using extra memory, which is undesired on an Arduino board. Hence, I removed them.
- **Introducing exploration_percent variable for dynamically controlling end of exploration trial:-** Initially, I ran my robot until whole maze is discovered. But, after watching a video of Udacity Introduction to Artificial Intelligence video[11], I decided to cap the exploration trial at 60%. This resulted in significant improvement in my model score.

0.11 Model Evaluation and Validation

Intuitively, this model should work efficiently because this is designed by using A-star search and Dynamic programming, each of which has a nice reputation in solving such kind of problems. Only problem is setting *exploration_percent* value.

This model is evaluated with different *exploration_percent* values. Results are shown on table below,

[11] https://classroom.udacity.com/courses/cs373/lessons/48646841/concepts/487161640923

Evaluation Results at *exploration percent* = 60			
Maze Dimensions	Upper Benchmark Score	Lower Benchmark score	Model Score
12	31.4	19.88	21.633
14	42.6	26.92	35.333
16	48.6	28.12	37.667

Evaluation Results at *exploration percent* = 70			
Maze Dimensions	Upper Benchmark Score	Lower Benchmark score	Model Score
12	33.8	20.36	21.800
14	45.866	27.57	36.500
16	52.86	28.97	38.900

Evaluation Results at *exploration percent* = 80			
Maze Dimensions	Upper Benchmark Score	Lower Benchmark score	Model Score
12	36.2	20.84	23.000
14	49.133	28.226	37.333
16	57.133	29.826	41.067

From above table, it can be seen that on three sample test mazes, model works best when exploration rate is set as 60%. This scenario will surely be different in other cases. One may use different statistical techniques to determine best exploration rate. For now, after analyzing the performance on the given sample test data, I will go with 60% exploration rate.

From above table, it can be seen that model score lies in between upper benchmark score and lower benchmark score for all three test cases. Hence, it can be concluded that my model is valid.

0.12 Model Justification

Model score is closer to lower benchmark score for 12 x 12 grid. It might possible due to small grid size and less complicated path. But, as we increase the maze size, complexity of the maze increases and difference between benchmark scores and lower benchmark score starts increasing. Also, model score increases with increase in *exploration_rate*. This is true here because luckily, model is able to find shortest path before exploration trial ends. In real world, this situation might be different and one might consider to increase *exploration_rate* for getting better knowledge of maze grid.

On a final note, I think my model can be used as a solution to this problem because it worked efficiently on all test mazes and can be adjusted to any environment only by tuning the *exploration_percent*.

0.13 Free Form Visualization

In this section, an important quality I would like to discuss about is dead end detection and preventing robot from visiting it again and again. Solution to this problem improved performance of code significantly.

Without this feature, testing on test_maze_3.txt will fail and performance of robot on test_maze_2.txt and test_maze_1.txt will be relatively poor.

The problem statement is: Given that my robot senses that there is dead end ahead and there is no way to get out except moving more than one step in backward direction, what steps should it take such that it gets out of that block of the maze.

Initially, I implemented a solution which worked as follows:-

1. If robot detects a dead end, it moves a step back.
2. At next step, it checks that at least two sensor readings are non zero.
3. If yes, then it randomly choses a direction and move.
4. Else it moves one step backwards.

This seemed to work fine for the mazes, but I was not happy with the results. Reasons were:

- In step 3, robot choses random direction, which means it can again go to path which leads to dead end, which in turn will increase number of steps in optimization trials for no reason and degraded robot's performance significantly.
- In step 2, I have to keep track that robot has detected dead end, otherwise robot won't even start because in starting only one sensor value is nonzero. For keeping track of it, I have to maintain several boolean variables which were creating some confusion in code.

To overcome these challenges, I decided to mark those nodes and while running A-star search, I decided to include only those nodes which were unmarked. These steps are shown in Step 4, 5 and 6 in *Robot movement while maze exploration* subsection of *Implementation* section.

This proved to be turning point in my model performance. Model score dropped by 1.2 in case of 12 x 12 maze, 3.6 in 14 x 14 maze and 5. 8 in 16 x 16 maze. It was a huge improvement in code performance.

0.14 Reflection

In layman terms, solution could be summarized as follows:-

1. Start from corner, see what's ahead of you.
2. If only one opening is there, then move in that direction.
3. If more than one opening is there, then make an educated guess and move in that direction.
4. While moving, mark all paths which lead to dead end and in future if you encounter any dead ends, then don't even consider to go on that path.
5. Also, while traveling, write number of steps you traveled from starting location, so that you may retrace your way back to goal in one shot.
6. After you have done enough exploration of the maze, return to starting position and using the numbers you have written down on path, trace down the shortest path.

When I took this project, I had no knowledge of any algorithm in AI domain. So, the first challenging task which I faced was to make myself familiar with algorithms of this domain. All thanks to Udacity Introduction to Artificial Intelligence course, due to which I was able to quickly grasp some important concepts.

Coming to problem, other challenge was to develop a design for this project. Initially, I thought it would be better to have all the code in a single file, but then when I started coding, it was becoming difficult to test and maintain the code. Hence, I stopped coning and started to think on design which aimed to make testing and maintenance easier.

Implementing the design wasn't a big job. However, biggest challenge was waiting. Challenge of improving performance and specifically, dead end detection(discussed in *Free form visualization* section.).

A minor problem was to decide a metric for upper bound score for this model. In project proposal, I had used the following formula:

$$upper\_bound\_score = \frac{a * steps\_in\_optimal\_path + 2 * b * steps\_in\_optimal\_maze\_exploration}{a + b} \tag{5}$$

but this metric failed to rank the model performance. Hence, I decided to take help from my experienced colleagues and done a little bit of research on Internet myself. After some time, I was able to come up with formula:

$$upper\_bound\_score = \frac{30 * steps\_in\_optimal\_path + 5 * steps\_in\_optimal\_maze\_exploration}{30} \tag{6}$$

Finally, I can say that I enjoyed the project a lot. Doing a project in a completely new domain was somewhat adventurous for me. I learned a lot from this project.

0.15 Improvements

This project was a simulation and everything was assumed to be perfect. In real life, one can't rely on sensor readings because even high quality sensors have some probability of giving false readings.

In case of real world, we have to take account of sensor errors and probability of failure. Including sensor errors and probability of failure in problem domain will require us to use concepts like localization, Kalman filters, particle filters, SLAM, reinforcement learning etc. Moreover, we will have to implement this concept on hardware, hence using PID will also improve this model and protect robot from collision and smooth rotation around the walls.

Another restriction which can be relaxed is movement direction of robot. We can allow robot to move in diagonal direction and then use some path smoothening algorithm to convert zigzag path to a smooth diagonal path. This will allow robot to cover distances more quickly. Although, programming will be complex, but what is fun in Artificial Intelligence and Computer Science if you throw out complexity.