Capstone Project **Machine Learning Engineer Nanodegree**	Emin Ozkan 06/03/2016

Definition

Project Overview

Robots are increasingly being a very important part of our lives. An important capability many robots need is the ability to navigate through known or unknown territory. From the robots that clean our floors to the ones that are involved in nuclear disaster zones, they need a way to get from one place to another while overcoming any obstacles that come their way. For example robots in Fukushima need to clean nuclear debris in every inch square of a disaster zone.

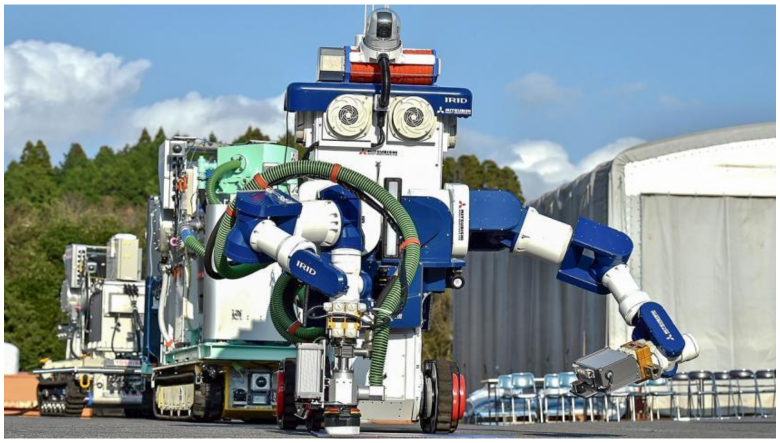


Figure 1: Giant robot that is used in Fukushima nuclear disaster zone

Self-driving cars need to be prepared for unexpected problems such as an accident in the road and plan for an alternative route. All of these problems call for an efficient method for searching and planning our ways.

Problem Statement

The main goal of my project is to produce a solution for the problem of robot navigation. The real world problem of robot navigation is simulated in a maze (shown on the right). Maze is an nxn grid of squares, with n being either 12,14 or 16. Each square may have walls on each of the four sides. These walls prevent the robot from moving to that square from the direction where a wall is present. The robot is supposed to efficiently navigate a maze in two different runs and reach a goal room, 2x2 square arena located at the center of the maze. In other words the robot is expected to reach the goal room using the least number of steps possible i.e. using the shortest path. However, scoring criteria is such that there is big penalty for each step during the second run, but small penalty for each step during the first run. Since the robot does not know the maze beforehand and penalty for exploration is small during the first run, first step is to teach the robot the maze in its entirety.

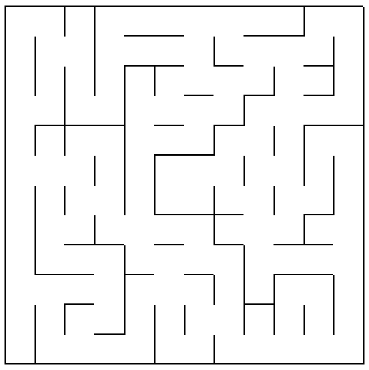


Figure 2: Test maze 1:12X12 grid

The strategy I will use to solve this problem is to make the robot visit as much of the maze as possible until the goal room is reached. One way to do this would be such that the robot does not stop until all the squares are visited. However, this may not be very efficient. The approach that I will use in my implementation is to gradually move away from the starting square. The specific method I will use is to make robot always choose the square that is farthest away from the start (max(gcost) function as explained below). So the goal of the robot in the first run will not necessarily be to reach the goal, but to move away from start. Once the robot reaches the goal, I will end the run. As you will see, in practice this approach worked very well with robot visiting most of the squares in the first run. In second run the robot knows about the maze almost completely, we can search for the shortest path using well known graph search algorithm breath first.

Metrics

The goal is to minimize the number of steps taken by robot to reach the goal. However penalty for exploration is much smaller during the first run. More specifically, the robot's score is determined by the number of time steps required to reach the goal in the second run plus one thirtieth of the time steps in the first run. Although the robot does not have to end the run if it reaches the goal room in the first run, the second run ends after the robot reaches the goal square. The maximum time steps allowed in each run is 1000. The goal is to get the lowest score possible.

Analysis

Data Exploration

First aspect of this project is the mazes that we will explore. Mazes are an nxn grid of squares, with n being either 12,14 or 16. We have three example mazes that are given to us as part of this problem. However if we want to we can generate more mazes according to specifications. In order to generate a maze we have to create a text file with the first line being the dimension of the maze (12, 14 or 16). Then in the second line we add n rows by n columns matrix to the text file. Because of array indexing in python, 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. Each number in the matrix specifies the number of open sides for each square. To calculate this number we first multiply 1(for top side), 2(for right side), 4(for bottom side) and 8(for left side) with either 0(if that side is walled) or 1(if that side is open) and add these four numbers together. We plug this number into the matrix in the text file for each square.

What about the specifications of the robot? The robot in this project is able to sense all the open squares in its front, right and left, but not in its back. Usually, the robot sensors come with a noise, but our robots sensors are perfect. The robot does not have a map of the environment; therefore map has to be constructed on the fly. The robot always starts at the bottom left corner of the maze. In each time step, the robot is allowed to make a single rotation and move up to three squares. Movement always follows the rotation. It doesn't matter whether the robot moves one square or three squares, a single time step will pass after each rotation and movement. The robot does not drift, therefore the movement is perfect.

Put in a different way, we are dealing with a two-phase maze solving problem. Because of the way the scores are determined, strategies will be very different during the first phase and the second phase. In first phase the goal is just to explore as much as possible (minimizing the same square visits). In the second phase the aim is just to get to the goal room as soon as possible. However if the maze is not explored well during the first run, the robot can get stuck in a solution that is not optimal. What does this problem model in real world? One way to think about it would be a rescue mission. The first robot in a rescue mission needs to explore the environment very well before other robots come for the help. This will expedite the rescue process.

Exploratory Visualization

Test_maze_01 and two possible solutions (blue and red) are illustrated below. This maze has two interesting properties. First both solutions require the robot to get to the other side of the goal room. Secondly, there is almost no overlap between the two solutions. What this means is that if the robot were to explore only the upper half and left side of the maze, it would not be able to find the more efficient route (the red one). Therefore I optimized the cost function used during the first run in order to make sure the robot finds the red route.

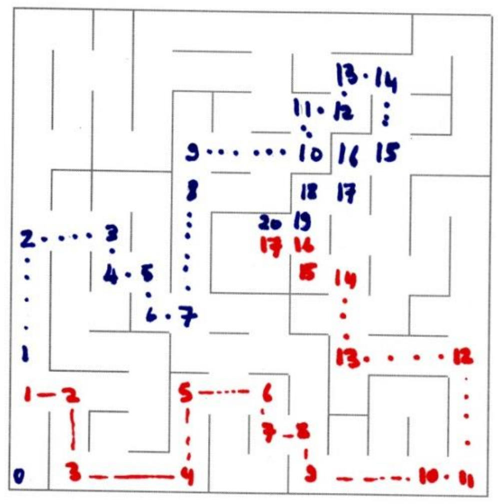


Figure 3: Two solutions for test_maze_01. The numbers indicate steps that are taken by robot to reach the goal room. Please note that the robot can jump up to 3 squares at time. Here the shortest solution is 17 steps whereas the second best solution is 20 steps.

```
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', 'G', 'L', 'L', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', 'U', 'L', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'U', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'U', 'L', 'L', 'L')
('R', 'D', ' ', ' ', 'R', 'R', 'D', ' ', ' ', ' ', ' ', 'U')
('U', 'D', ' ', ' ', 'U', ' ', 'R', 'D', ' ', ' ', ' ', 'U')
|('U', 'R', 'R', 'R', 'U', ' ', ' ', 'R', 'R', 'R', 'R', 'U')
```

Figure 4: Solution obtained by my robot (same as the red router above, 3 step jumps are not shown in this figure, although they are implemented by the robot)

<u>Solutions of the other mazes</u>:

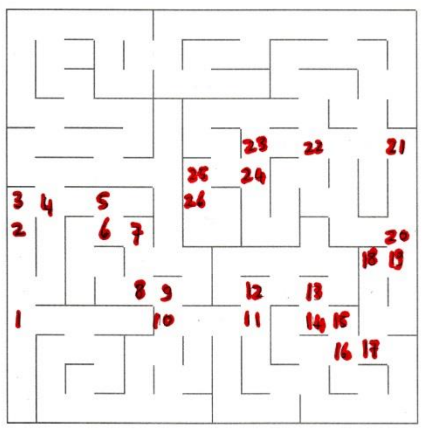


Figure 5: Optimal solution for test maze 02. The numbers indicate steps that are taken by robot to reach the goal room. Please note that the robot can jump up to 3 squares at time.

```
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'D', 'L', 'L', 'L', 'L', 'L')
(' ', ' ', ' ', ' ', ' ', ' ', 'D', 'L', 'L', ' ', ' ', ' ', ' ', 'U')
('R', 'R', 'R', 'D', ' ', ' ', 'G', ' ', ' ', ' ', ' ', ' ', ' ', 'U')
('U', ' ', ' ', 'R', 'D', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'U')
('U', ' ', ' ', ' ', 'D', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'R', 'U')
('U', ' ', ' ', ' ', 'R', 'D', ' ', ' ', 'R', 'R', 'D', ' ', 'U', ' ')
('U', ' ', ' ', ' ', ' ', 'R', 'R', 'R', 'U', ' ', 'R', 'D', 'U', ' ')
('U', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'R', 'U', ' ')
('U', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
```

Figure 6: Solution of the test maze 02 as reported by the robot

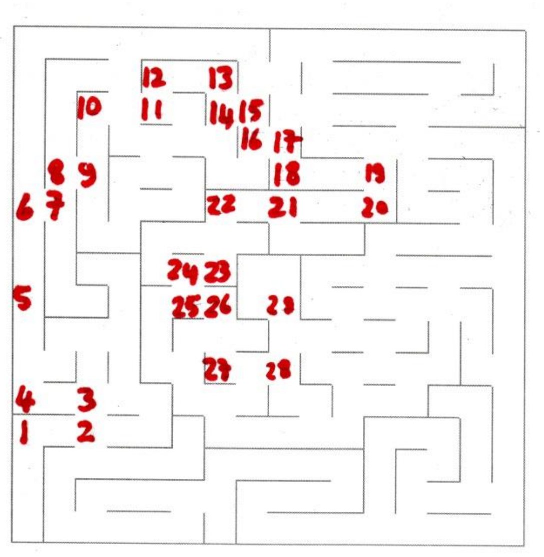


Figure 7: Optimal solution for the test maze 03

```
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', 'R', 'R', 'D', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', 'R', 'R', 'U', ' ', 'R', 'D', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', 'U', ' ', ' ', ' ', ' ', 'R', 'D', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', 'R', 'U', ' ', ' ', ' ', ' ', ' ', 'R', 'R', 'R', 'D', ' ', ' ', ' ', ' ')
('R', 'U', ' ', ' ', ' ', ' ', 'D', 'L', 'L', 'L', 'L', 'L', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', ' ', 'D', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', 'D', 'L', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', 'R', 'D', ' ', 'G', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', ' ', 'D', ' ', 'U', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', ' ', 'R', 'R', 'U', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', 'L', 'L', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('R', 'R', 'U', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('U', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
```

Figure 8: Solution of the test maze 03 produced by the robot

Algorithms and Techniques

There are two important techniques essential to implementation. First technique related to exploration run, and it is the calculation of the cost function. A cost function is a value defined for each square. In this project, I have experimented with three different cost functions G value, H value and F value. G value is the amount of squares away from the initial square. As the robot moves away from the initial square, g value increases. In order to calculate g value we first start with 0 for the start square. When we are first starting the exploration we see that there are open squares ahead of us. We can see for example that G value for the square

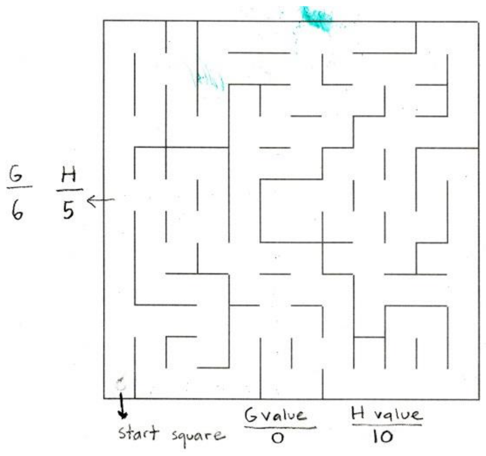


Figure 9: Example G value and H value calculation for two squares

just next to our starting square would be 1. Then G value 6 steps ahead would be 6 etc. H value is a heuristic value, basically a guess into how much we are away from the goal. Although we don’t know how many steps we are away from the goal, we can estimate this if we assume that there are no obstacles. In order to calculate H value, we simply add the amount of vertical and horizontal squares to the goal as if the maze is empty. This distance to the goal is also called manhattan distance. Example G value and H value calculations are shown for two different squares in test maze 01 in the figure above. F value is calculated by adding G value and F value of a square.

The second technique is related to second run and used to calculate the shortest path to the goal. Here I use breath first algorithm, but A* was also a possibility. Difference between breadth first and A* algorithm is that breadth first algorithm only uses G value for each square. A* algorithm on the other hand uses F value by combining both G value and H value. Here is a brief description of the bread first algorithm. We start at the start square (now called current square) and try to move in four directions (up, down, left, right). If there is a wall in any of the directions, we cannot move. However if it is possible to move in any of these directions, we add these new squares to a list called the open list. Open list means that these squares are available but unvisited. We add each square to the open list only once. During the process of adding each square to the open list we also calculate their g value by adding 1 to the g value of the current square. This is the case because it took us 1 more step to get to that square from the current square. Once we visit a square we add this square to the closed list, meaning we have already considered this square. We also remember which action we took to add each square to the open list in order to track our way back to the start. Once we are done with the above steps for the start square we pick the square with the lowest G value and make that the current square. We repeat this process until the goal has been found. Now we can track our way back to the start by looking up what action we took to get to that square. Both A* and breadth first algorithms guarantee the shortest paths to the solution. However breadth first algorithm is implemented here for two reasons. First computational costs are not considered as part of scoring criteria (A* would be faster). Second reason is practical. By considering only g value we don’t have to calculate a new heuristic table at each step, since the calculation of shortest path is performed to get to squares other than the goal during exploration phase.

Benchmark

The benchmark will depend on the complexity of the maze as well the scoring criteria. For the test_maze_01 the optimal solution is 17 steps. If we assume that the robot has a way to visit all the squares of the maze (144 squares in total) very efficiently (only once), total minimum score would be:

17+144/30= 17 +4.8 = 21.8

My implementation gets 21.933, which is in the ballpark for what you would expect from an optimal robot for this maze.

Is it possible to create a benchmark that is applicable to all mazes? We can assume that in the first run robot will have to visit all the squares. Therefore get a score of (nXn)/30. For the second run although the answer will depend on the complexity of the maze, we can think about the best and worst case scenario. The best case would be an empty maze, so the robot would traverse it easily in N/3 time steps. The worst case scenario would be a very complex maze where the robot would have to visit all the squares and make a rotation at each step. Therefore it would take NXN steps to reach the goal. If we take the geometric mean of these two numbers we get, this would be $N^{3/2}/3^{1/2}$. I use geometric mean here as it is more robust to outliers such as worst case maze. So, if we calculate the general solution for the test_maze 01 again, it would be:

$3^{1/2}*N^{3/2}+(NxN)/30 =41.6/1.7 + 144/30= 24.4 +4.8 = 29.2$

What about test_maze_2 and test_maze_3?

For test_maze_2, using the same formula we get 30.8+6.5 = 37.3

For test_maze_3 we get 37.6+8.5=46.3

Since Emin_maze is also 12X12, the benchmark would also be 29.2.

Please keep in mind that in reality mazes that are used in this problem are much easier, so it is possible to get lower scores than these benchmarks.

Methodology

Data Preprocessing

No data preprocessing was necessary in this project, because both specifications about the maze and the robot is provided in a noise free robust manner. For example, if sometimes sensors did not work, we would have to deal with the missing data. Or if there was any noise in the sensors, we would have to apply filters before interpreting the sensors. All of these would count toward preprocessing; however this is not the case here.

Implementation

Based on these specifications, I designed the robot to operate in two different modes. The first mode is called exploration mode, while the second mode is called target mode. Exploration mode is only used during the first run, while the target mode is used both during the first run and the second run. Because the current implementation of the exploration mode already covers most of the maze until the goal room is found, the robot ends the first run once it enters the goal room.

Aim of the robot in the exploration mode is to explore the maze as completely and efficiently possible. In the exploration mode robot moves only one step in each direction. This may seem inefficient, but it is the only way to make sure that the robot does not miss any junctions as it is navigating the environment. If the robot were to move three steps at a time, and if there were a junction during the second step ahead, the robot would not be able to sense it during the move.

Here’s in detail steps we take during exploration run. The first step is to read all the sensor data. We read the heading and the available squares on each of the three directions and update the maze map and the open list according to this data. This goal is achieved using update_maze_map and update_open_list functions.

```
self.update_maze_map(sensors)

self.update_open_list(sensors)
```

Here we have to take a step back and talk about update_maze_map and update_open_list functions. update_maze_map is a very important backbone of the robot. In my implementation maze map is a 3 dimensional array.

```
self.maze_map = np.zeros((maze_dim,maze_dim,4),dtype=int)

self.maze_map.fill(-1)
```

First two dimensions indicate the specific squares of the maze. The third dimension indicates the walls of each square. When we construct the maze, we assign the value -1 to indicate that we assume that they are closed. However, during the process of updating the maze, we assign this value to 1 to indicate that they are open. As you can see below the process of updating is highly dependent on the current heading. For example if we are heading right, the available squares on our front are actually to the right of the maze. Here is just a short section from update_maze_map

```
if re.match(heading_name,'right'):

        #update according to left reading

        for i in range(0,(left_open_count)):

            maze_map[current_x,current_y+i,0]=1

        for i in range(1,(left_open_count+1)):

            maze_map[current_x,current_y+i,2]=1
```

A similar process happens during obtaining open lists. We take into account our heading. G value updates are also done during obtaining open lists. Here is just a short section from update_open_list

```
  if re.match(heading_name,'up'):

        #update according to left reading

        for i in range(1,(left_open_count+1)):

            available_next = [current_x-i,current_y]

            if not (( available_next in self.open_list) or  (available_next in self.closed_list)):
                    self.open_list.append(available_next)

                if self.gcost_array[available_next[0]][available_next[1]] > (current_gcost + i):

                    self.gcost_array[available_next[0]][available_next[1]] = current_gcost + i
```

Once maze map and open lists are updated, next step is to decide what the immediate available squares are. The available squares that are one step ahead are called immediate open squares and are stored in immediate_open_list.

```
immediate_open_list = self.get_immediate_open_list(sensors)
```

get_immediate_open_list function is very similar to get open_list function, except only squares that are one step ahead are considered:

```
if re.match(heading_name,'up'):

            #update according to left reading

            if left_open_count:

                immediate_open_list.append([current_x-1,current_y])
```

The robot then picks the immediate open square according a cost function (discussed below). Once a square is picked (called immediate_target), we calculate rotation and movement necessary to get to that square using next_immediate_move function.

```
immediate_target                                                                  =
immediate_open_list[immediate_open_gcosts.index(max(immediate_open_gcosts))]

                    rotation,   movement   =   self.next_immediate_move(heading_name,   current_x,
current_y, immediate_target)

                    self.location = immediate_target
```

However, it is possible that there won't be any available open squares on either side of the robot. This can happen either because we hit a physical dead end, or we already visited all the available squares immediately on the front, right or left. When the robot gets stuck in a dead end, the robot switches to target mode in order to get to a previously unvisited square according to the same cost function. Here in the code below we see that once the immediate open list is empty we switch modes and pick a target. Here this target is called away_target because it is not a target we can immediately reach. We have to perform at least two steps. However because we can only make one movement and rotation at a time, we have to calculate what is immediate movement and square(called to_away_target) that will take us there. This is calculated using get_shortest_path function. This function performs Breadth first algorithm and tell us the first movement and rotation we have to make to get to our away_target.

```
if len(immediate_open_list)==0: %empty immediate open list:
```

```
            self.mode = 'Target'

            self.away_target = self.open_list[open_gcosts.index(max(open_gcosts))]

            rotation, movement, to_away_target = 
self.get_shortest_path(sensors,self.away_target)

                        self.location = to_away_target
```

Once we reach an away target, we switch back to exploration mode:

```
if self.location  == self.away_target:

                self.mode='Exploration'
```

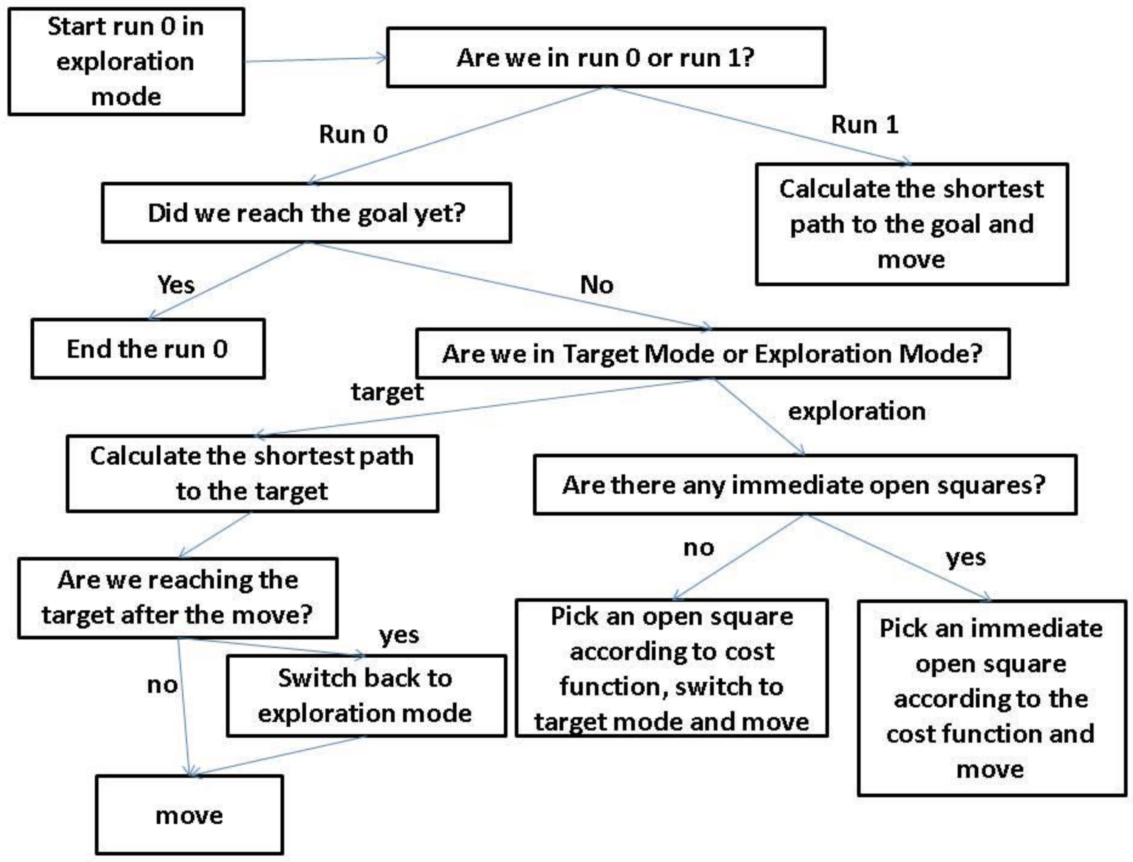
Once we enter the goal room, the exploration phase ends

```
if (self.location in self.goal_room):

                rotation = 'Reset'

                movement = 'Reset'

                self.goal_door = self.location
```

Now we can switch to second phase. The goal in the target mode is to get to a specific square fastest way possible. It is called target mode because we have a target square we want to reach. This specific target square could be a previously unvisited square during the first run, but it is the goal square during the second run. One key difference between two operating modes is that robot in the target mode moves up to three squares if possible. Therefore the primary purpose is not to explore, but to use shortest path given what we know about the maze at that moment. In order to calculate the shortest path, the robot uses ‘breadth first’ algorithm as described in “Artificial Intelligence for Robotics” course in Udacity. There were also other choices here, such as A*, but both algorithms are guaranteed to give us the shortest path. A* is only computationally faster, but which algorithm we choose here will not affect the performance of the robot, since both algorithms are performed in one time step. Entire second run actually takes only a couple lines of code:

```
elif self.run_index==1:

            self.mode = 'Target'

            self.away_target = self.goal_door

            rotation, movement, to_away_target = self.get_shortest_path(sensors,self.away_target)

            self.location = to_away_target
```

The decision tree utilized by my robot is shown below:


Start run 0 in exploration mode
Are we in run 0 or run 1?
Run 0
Run 1
Did we reach the goal yet?
Calculate the shortest path to the goal and move
Yes
No
End the run 0
Are we in Target Mode or Exploration Mode?
target
exploration
Calculate the shortest path to the target
Are there any immediate open squares?
no
yes
Are we reaching the target after the move?
yes
no
Switch back to exploration mode
Pick an open square according to cost function, switch to target mode and move
Pick an immediate open square according to the cost function and move
move

Results

Model Evaluation and Validation

The cost function during the first run is essential to the performance of the robot. As I demonstrate in a table below, I have experimented with different cost functions. Here I have manipulated the cost function used by the robot during exploration in order to see how they affect the performance of the robot. Please note that the robot always moves optimally during the target mode (given the knowledge it has at the time). When comparing different cost functions during the exploration mode it is clear that we can find ones that are the best for a given maze. However, we should be very careful not to ***overfit***. I highlight the word overfit here, because it is so important. For example if we use Min(fcost), we get a very low score for test maze 01. However, these results do not generalize to other mazes. The same is also true for Min(hcost) cost function. We get a very good score for the difficult maze 03, but terrible score for the other mazes.

Scores

Cost function used during the exploration	*Test_maze_01*	*Test_maze_02*	*Test_maze_03*	*Emin_Maze*	*Mean score*
Min(gcost)	28.9	37.7	38.833	13.867	**29.82**
Min(hcost)	27	42.5	34.633	26.37	**32.62**
Min(fcost)	24.267	39.2	38.233	13.467	**28.79**
Max(gcost)	21.933	30.267	36.70	14.3	**25.8**
Max(hcost)	30.2	44.267	40.2	16.867	**32.88**
Max(fcost)	22.067	32.233	37.267	15.833	**26.85**
Benchmark	29.2	37.3	46.3	29.2	**35.5**

Justification

After examining the mean scores for all the mazes we can see that the cost function max(gcost) achieves the best performance. The mean score for this cost function is 25.8 versus the closest competitor is max(fcost) function with 26.8. Therefore I have chosen Max(gcost) as the best performing algorithm given it got the lowest overall score.

In terms of benchmark, generally the cost functions do very well for most mazes. Interestingly test maze 2 is an exception. This indicates that test_maze 2 is a particularly challenging maze, requiring robot to particularly examine most of the maze before a solution has been found. Only max(gcost) function and max(fcost) function manage to beat the benchmark.

Conclusion

Free-Form Visualization

Below, I provide a maze that highlights the importance of choosing the right parameters according to the maze provided. The characteristic of this maze is that there is a very big cost difference between the two solutions of the maze. Therefore the robot needs to be a very efficient explorer and not get stuck in the upper part of the maze.

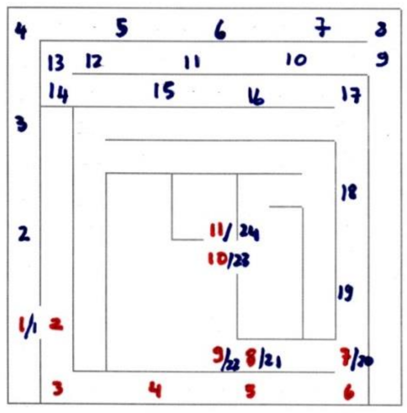
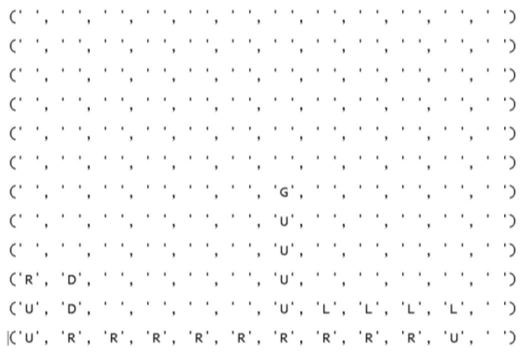


Figure 10: A different type of maze: Emin maze. Two different solutions are provided in red and blue.



```
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', 'G', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', 'U', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', 'U', ' ', ' ', ' ', ' ', ' ')
('R', 'D', ' ', ' ', ' ', ' ', 'U', ' ', ' ', ' ', ' ', ' ')
('U', 'D', ' ', ' ', ' ', ' ', 'U', 'L', 'L', 'L', 'L', ' ')
('U', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'U', ' ')
```

Figure 11: Solution of the Emin_maze provided by my implementation. Same as the red route above, 3 step jumps are not shown in this figure, although they are implemented by the robot)

Improvement

If this problem was in continuous domain, we would need additional functionality in our robot for an efficient implementation. For example sharp turns would waste us time because they would be slower, and they would be longer than more direct turns. We could apply smoothing in order to avoid sharp turns and shorten the distance travelled. This is explained well in “Artificial Intelligence for Robotics” course in Udacity. In addition, score would have to be calculated differently. We had to take into account actual time to finish, or total amount of energy expenditure.

Another very important aspect of my implementation that needs to be changed in a continuous domain would be how the robot would calculate the shortest path. Although exact same algorithms such as breadth first can be applied, we would have to divide our continuous world into artificial discrete squares before implementation. We can imagine that the size of these squares would depend on minimum distance that a robot can move at a time.

Conclusion:

Although the idea behind creating a robot that explores a maze is straightforward, the challenge in this project lies in the fact that the robot sensors were always relative. Therefore all the updating steps as well as rotation and movement calculations need to be made taking into account robots heading. I think that in future iterations the project could be made even more challenging by not specifying the location of the goal room.

In conclusion, I have implemented a robot navigating an unknown territory in order to reach a target location. My solution to this problem highlights the importance of flexibility in artificial intelligence. By creating a robot that can operate differently under different circumstances, when stuck in a dead end or during different visits to the same maze, I shown how real word problem of navigation can be solved efficiently by modeling it in a grid like world.

References:

http://www.sciencemag.org/news/2016/03/how-robots-are-becoming-critical-players-nuclear-disaster-cleanup

https://www.udacity.com/course/machine-learning-for-trading--ud501