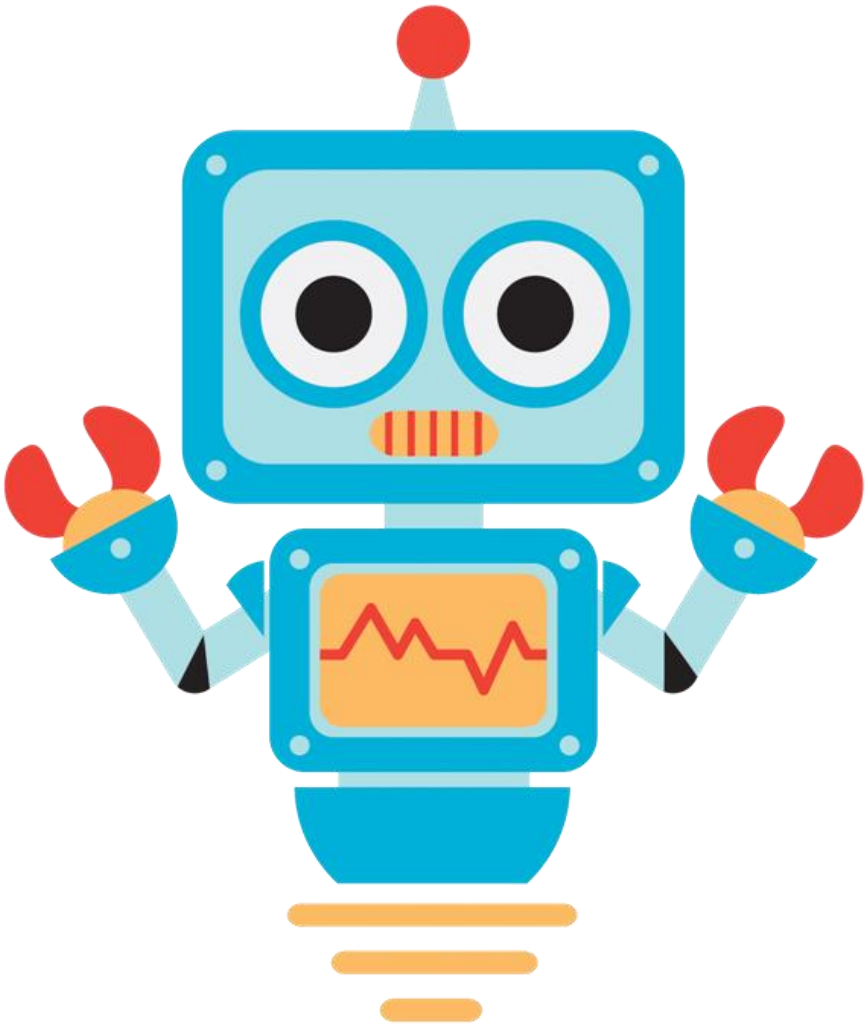
Robot Motion Planning Capstone Project

Plot and Navigate a Virtual Maze

Qingyuan Ji

Udacity Machine Learning Nanodegree

November 18, 2017



Definition

Project Overview

This project is inspired by and based on the annual competition event – the Micromouse, where a small robot mouse is asked to navigate small sized maze (https://en.wikipedia.org/wiki/Micromouse). The maze is a grid cell system normally in the size of 12×12, 14×14, or 16×16 cells. Walls exists in the maze, to decide the permitted move into a cell in different directions. There is a goal area in the centre of the maze. The robot will be asked to starting at a certain cell in the maze, and try to explore the unknown maze as much as possible and figure out the fastest way to reach the goal from the starting cell.

Problem Statement

The maze a grid system in the size of 12×12, 14×14, or 16×16. For each maze, the robot will be given two runs to solve it. In each run, the robot always starts from the bottom-left most cell. In the first run, the robot knows nothing about the maze (except for its size), and should try to explore the maze as much as possible and also find the goal area. Then it should be able to somehow map the maze based on its 1st run. After that the robot will start its 2nd run to reach the goal area as soon as possible.

The robot is equipped with 3 sensors on the forward, left and right side of it to know how far away the nearest wall is in these 3 directions. To clarify the allowed action of robot, we define "step" here. A "step" consists of a rotation and a movement. A rotation can be clockwise 90 degree, counter clockwise 90 degree, or no rotation. A movement can be 0, 1, 2, or 3 units its forward direction (a unit is the distance to move into the adjacent cell provided that no walls blocking). So for example, a valid step can be "robot turns 90 degree clockwise, and make a movement for 2".

Metrics

As mentioned earlier, the robot has 2 runs to beat a maze. Moreover, we regulate that the robot is given totally 1000 steps for its 1st and 2nd run. That means if the robot spend more than 1000 steps in 2 runs, it is considered to be a failed performance.

Furthermore, if the robot does complete 2 runs within 1000 steps, and reach the

goal area in the end of the 2nd run, we want to judge how "fast" it is in order to solve this maze. The metrics we will use here will be based on the steps it used in the 1st and 2nd run. In particular, the metrics score is defined to be the 2nd run steps plus 1/30 of the 1st run steps. For example, if the robot spend 900 steps in the 1st run and 30 steps in the 2nd run, then it has a score of 900 * 1/30 + 30 = 60.

Therefore, it seems to be a good strategy for the robot to explore the maze as much as possible in the 1st run, even that will spend lots of steps. If the exploration process pays off (for robot to effectively figure out a shortest path in the 2nd run), it is considered to be a worthy exploration.

Analysis

Data Exploration

Each maze is represented as a text file. For example, the test_maze_01.txt file we used in this project looks like the following:

```
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
```

In here, the first line indicates the dimension of the maze, so we know our test_maze_01 is a 12×12 maze. From the second line to the end of the file, each number here represents the permissibility of each cell within in the maze. The number is always between 1 and 15 (including 1 and 15) and actually indicates presence of walls at the four directions (up, right, bottom and left) of this cell.

The idea is that we used 0, 0, 0, and 0 to represent the presence of walls at up, right, bottom and left of a cell, and 1, 2, 4, and 8 to represents the absence of walls for a given cell. If we add all the four numbers together, that is the number for the corresponding cell. For example, in the figure 1, there are 9 adjacent cells, which

actually form the left-bottom most corner for the test_maze_01. The blue lines indicate the wall and the dashed lines no wall. We can see that the cell on the top-right corner has the value of 10. Because that cell has walls on the top and bottom, but no walls on the left and right, so its value is 0 (top) + 2 (right) + 0 (bottom) + 8 (left) = 10.

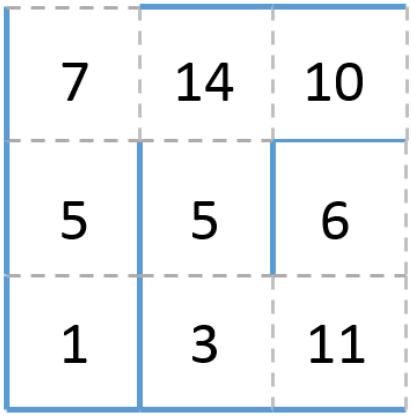


Figure 1. Explanation of the cell value.

Note that, due to array indexing issue, the number for each in the text file, doesn't look exactly the same as it is in the maze visualization result. We will be able to visualize the test_maze_01 as a whole in the next section.

Exploratory Visualization

The test_maze_01 is showed in figure 2, where the blue cell indicates where the robot starts its 1st and 2nd run and the red area in the centre indicates the goal area. To make it easier to locate cell and for constructing algorithm in this project, we add a 2D coordinate system into the maze. The starting point can be referenced by the cell (0, 0). The goal area can be regarded as a combination of four cells, cell (5, 5), cell (5, 6), cell (6, 5) and cell (6, 6). Moreover, at the beginning of the 1st and 2nd run, the robot is facing the up direction, or in other words, the y axis positive direction.

Moreover, at any cell in the maze, the sensors on the robot will tell the distance of the nearest walls to it at three directions (left hand side, forward, and right hand side), depending on the heading of the robot. For example, when at the starting cell (0, 0), and heading up (y-axis positive direction), the robot knows that on its left hand side, the wall distance is 0, on its right hand side, and the wall distance is also 0, and on its forward, the wall distance is 11.

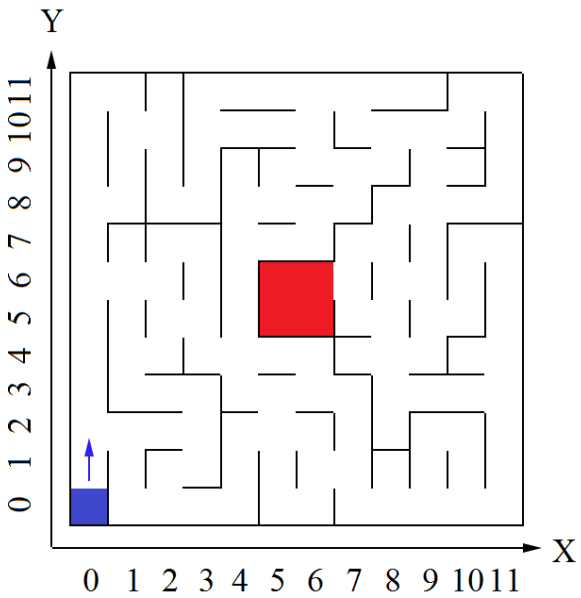


Figure 2. Configuration of test_maze_01.

From figure 2, we can know that this is a maze of several loops and dead ends. For example, if we travel via the cells (1, 0), (1, 1), (1, 2), (2, 2), (3, 2), (2, 1), (2, 0) and back to (1, 0), we actually travelled around a small loop that exists very close to the bottom-left of the maze. On the other hand, if we travel into the cell (8, 1), that is a dead end for us. In my opinion, the loops and dead ends here are the real challenge for our robot. Because if they are not carefully dealt with, our robot will be trapped into the loops or dead ends again and again, spending many steps but just couldn't get out of them, and ultimately failing to explore other parts of the maze. This maze has some but not many loops and dead ends, so it should be a good starting point to test our robot.

Based on the configuration of test_maze_01, it is also possible to draw a shortest path (spends fewest steps) for the robot, which is showed in figure 3. This is a path consisting of 17 steps, and can be broken down into a sequential of steps showed in table 1. In table 1 we use -90 degree to indicate a counter clockwise rotation 90 degree, and +90 degree to indicate a clock wise rotation 90 degree and 0 degree to indicate no rotation.

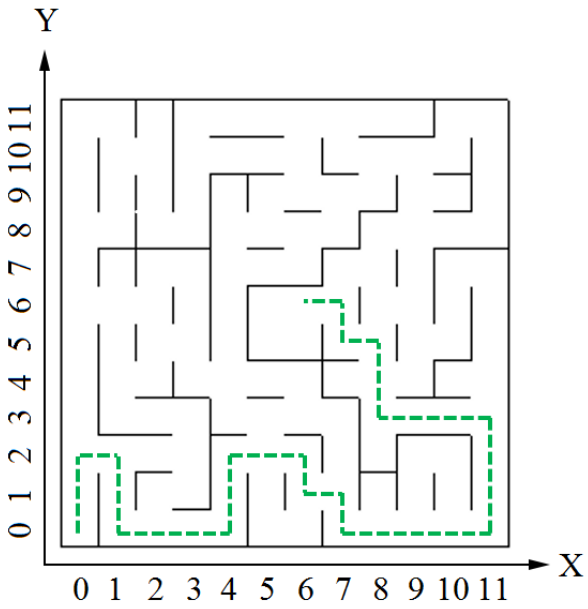


Figure 3. Shortest path to solve test_maze_01.

Current Position	Total Steps	The Next Step
(0, 0) - Start	0	0 degree, 2 units
(0, 2)	1	+90 degree, 1 unit
(1, 2)	2	+90 degree, 2 units
(1, 0)	3	-90 degree, 3 units
(4, 0)	4	-90 degree, 2 units
(4, 2)	5	+90 degree, 2 units
(6, 2)	6	+90 degree, 1 unit
(6, 1)	6	-90 degree, 1 unit
(7, 1)	7	+90 degree, 1 unit
(7, 0)	8	-90 degree, 3 units
(10, 0)	10	0 degree, 1 unit
(11, 0)	11	-90 degree, 3 units
(11, 3)	12	-90 degree, 3 units
(8, 3)	13	+ 90 degree, 2 units
(8, 5)	14	-90 degree, 1 unit
(7, 5)	15	+90 degree, 1 unit
(7, 6)	16	-90 degree, 1 unit
(6, 6) – Goal	17	None

Table 1. Break down of shortest path showed in figure 3 for test_maze_01.

Algorithms and Techniques

The aim of this project is to develop an algorithm to allow robot effectively explore the maze in the 1st run and then try to reach the goal area as fast as possible in the

2nd run. Therefore, different algorithm are required to deal with these two run separately.

For the 1st run, I mainly used a random action algorithm to pick up a random action from all the allowed actions (based on robot sensors, position and robot heading) for the next step. Moreover, the robot has a preference that, this random action had better send him to an unvisited cell. Figure 4 shows what I mean here.

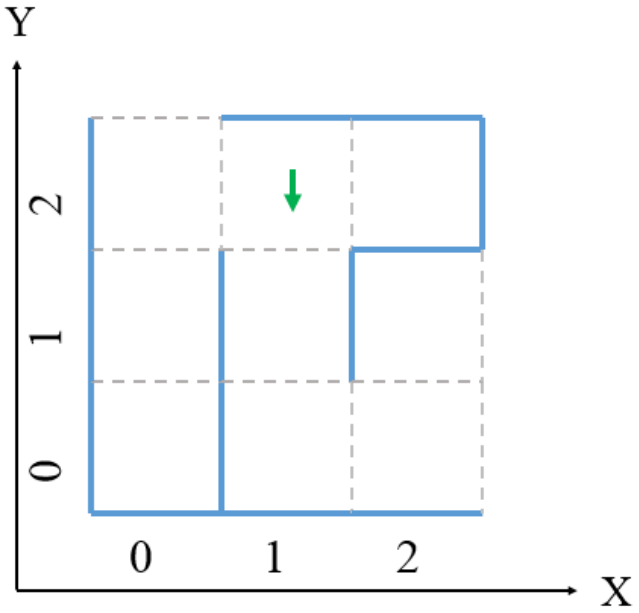


Figure 4. Example to show random action algorithm.

In figure 4, let's suppose the robot is now in the cell (1, 2) and it is facing bottom (or the y-axis negative direction). Please note, here we don't consider reliability of moving backwards to just simplify the problem. Now the sensors of robot will tell that on the left hand side the wall distance is 1, on the forward side the wall distance is 2 and on the right hand side the wall distance is 1. Since the robot is limited to make a movement of 3 units at most, then the robot has all the following actions:

Action 1 – Make no rotation, and move for 1 unit into cell (1, 1)
Action 2 – Make no rotation, and move for 2 units into cell (1, 0)
Action 3 – Turn counter clockwise 90 degree, and move for 1 unit into cell (2, 2)
Action 4 – Turn clockwise 90 degree, and move for 1 unit into cell (0, 2)

Obviously, these 4 actions are the only allowed actions the robot should consider for the next step. Moreover, during the 1st run (exploration run), the robot will always keep track of which cell it has visited and which are not. Therefore, in this case, let's suppose the cell (1, 1) and cell (2, 2) have been visited and cell (1, 0) and cell (0, 2) are not. Therefore, the robot picks up a random action that will send it to cell (1, 0) or (0, 2), so it will choose either action 2 or action 4 here.

In some cases, if no allowed actions can send the robot to an unvisited cell (very likely because the robot has explored a majority of the maze already), then the robot simply picks up a random action that is allowed. Also, in very few cases, the robot doesn't have any allowed action, and the reason is that the robot gets stuck in a dead end (or in other words, sensors tell him that on the forward, left hand and right hand side there are all immediate walls), so in that situation, we let the robot to turn 90 degree clockwise and make a movement for 0 unit for the next step, to let him get out of the dead end.

Using this algorithm, the robot will try to visit as much as possible cells in the first run and record the cell value of each cell it visited. There is also additional constraints in the 1st run, which is, if the robot has explored all the cells in the maze including the goal area, it should terminate the 1st run. That makes sense, because the robot should be able map the maze now (or more preciously speaking the, location of every wall) and there is no need waste steps in the 1st run. **On the other hand, if robot has reached the goal area but not visited every cell yet, we let the robot to terminate its 1st run, if it already spends 900 steps.** The reason to set up this threshold is that we need to still make sure we have remaining steps for the 2nd run. Even if the robot hasn't explored every cell yet, it should have explored a majority of them, so that should be enough to map the maze.

During the 1st run, the value of each cell visited is stored in a grid which has the same size of the maze, and we call it valueGrid. For example, below can be the valueGrid for the test_maze_01 for a certain run:

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

Figure 5. The valueGrid for the test_maze_01 (example)

Ideally, if our robot is able to visit all the cells in the maze, then this valueGrid we get should have exactly the same value stored in the test_maze_01.txt file.

After generating the valueGrid, we will use heuristic approach to construct a

heuristic grid, called heuGrid which is also the same size of the maze. This approach is inspired by the Udacity course of Artificial Intelligence for Robotics (CS373) (https://www.udacity.com/course/artificial-intelligence-for-robotics--cs373). Each number in the heuGrid is the shortest distance from the corresponding cell to the goal area. Figure 6 shows the heuGrid based on the valueGrid in figure 5. To calculate the heuGrid, we need to start from the goal area, and assign heuristic value of the goal areas (totally four cells) to be 0. Then starts from them, find all the adjacent cells (as long as no walls blocking) and assign heuristic values for these cells 1. It is like a propagation process that starts from the goal area, until it spreads to every cell in the maze. We can see the starting point is the nearest cell to the goal, since it has the largest heuristic value (30).

[23, 22, 21, 14, 13, 12, 11, 10, 11, 12, 9, 10]
[24, 21, 20, 15, 14, 13, 12, 9, 8, 7, 8, 9]
[25, 22, 19, 16, 15, 12, 11, 10, 9, 6, 7, 8]
[24, 23, 18, 17, 14, 13, 12, 11, 4, 5, 6, 7]
[25, 24, 23, 22, 15, 14, 13, 2, 3, 6, 9, 10]
[24, 23, 22, 21, 16, 0, 0, 1, 4, 5, 8, 11]
[25, 24, 21, 20, 17, 0, 0, 2, 3, 6, 7, 10]
[26, 23, 22, 19, 18, 19, 20, 5, 4, 5, 10, 9]
[27, 24, 25, 26, 19, 20, 19, 18, 5, 6, 7, 8]
[28, 27, 28, 27, 20, 19, 18, 17, 6, 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]

Figure 6. The heuGrid calculated based on figure 5.

After that, we will use dynamic programming to create a policy grid (policyGrid) that is still the same size of maze. It is actually a global policy so that no matter which cell our robot is, it know how to reach the goal area as fast as possible. Generating policyGrid based on the heuGrid is straightforward. For each cell, the policy (the suggested travel direction) is the direction through which it can move to the adjacent cell having the smallest heuristic value, provided that no walls blocking. If there are multiple visitable adjacent cells having the same smallest heuristic value, then just picks up a random direction to any of them. For example, figure 7 shows the policyGrid generated based on the figure 6. Note that the cells displayed as dash '-' are actually the goal area.

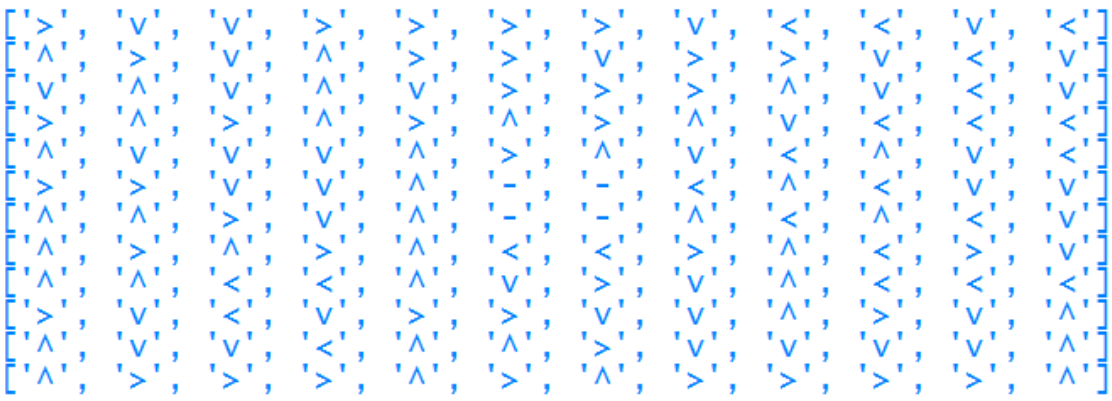


Figure 7. The policyGrid generated based on figure 6.

Pretty much all preparations are done for the 2nd run at this moment, and the robot should be able to start the 2nd run.

For the 2nd run, the robot is asked to follow the policyGrid generated above, therefore from its starting point it should be able to navigate to the goal area. This is mostly correct, but each step the robot will only make only a movement of 1 unit, which is not a very efficient way. According to the project definition, our robot is allowed each time, to take a step of movement of 3 units at most, and therefore this should be taken into consideration to save the robot some time. The solution for this is also straightforward. The robot will be instructed in the 2nd run, that following the policyGrid, if there are two or three arrows that is point to the same direction, it can take a single step of two or three movements. Therefore, following the policyGrid showed in figure 7, it will take as few as 17 steps totally for the robot to reach the goal area.

Benchmark

The benchmark for this project, in my opinion should be an expected spending steps for the robot in the 1st and 2nd run. Our metrics score really puts a quite small weight on the running steps in the 1st run, so intuitively I would argue it is a sensible approach that the robot try to make a thorough exploratory 1st run, even if it costs a lot of steps. The larger the maze is, the most steps needed to make such thorough exploratory run. The larger the maze is, the more steps needed to make a thorough exploratory run. Therefore I would say the benchmark for each maze can be something like this:

Maze 1 (12×12): 500 – 600 steps in 1st run, 20 - 30 steps in 2nd run, score is 37 – 50.
Maze 2 (14×14): 600 – 700 steps in 1st run, 25 - 35 steps in 2nd run, score is 45 – 58.
Maze 3 (16×16): 800 – 900 steps in 1st run, 30 – 40 steps in 2nd run, score is 56 – 70.

Methodology

Data Processing

This project is a bit special that it provides the accurate modelling environment, the mazes, where starting points and goal areas are specified already. Therefore, no further data processing work needs to be done to solve this project.

Implementation

We mentioned previously that, during the 1st run, the robot will keep track of which cells it has visited and which cells are not. This is implemented by assigning an object variable visitedGrid to the robot object. Every cell in the visitedGrid is set to o indicate not visited initially.

```
# This 2D array is used to identify if a cell is visited
# The value 0 means unvisited and 1 means visited, initially everycell is unvisited  # NOQA
self.visitedGrid = [[0 for row in range(self.maze_dim)] for col in range(self.maze_dim)]  # NOQA
```

In the 1st run, we define the calculateAllowedActions () function and calculatePreferedActions () function to help the robot make decision for the next step.

```
def calculateAllowedActions(self, location, heading, sensors):
    '''
    This function will calculate allowed actions for the robot,
    at a given location, heading and sensors information.

    In particular, this function will return a list, where each element
    in the list is a 2-elements tuple to indicate the direction and
    movements.

    For example, if this function returns [('u', 1), ('l', 1), ('l','2')],
    that means there're 3 allowed actions for the robot at this stage. One
    is to make 1 movement up, one is to make 1 movemenet to the left, and
    and the other one is to make 2 movements to the left. Note that the
    directions 'u', and 'l' are the global directions, and or the
    absolute directions.
    '''

    # Get the headings of 3 sensors in the global direction system
    sensors_dirs = self.dir_sensors[heading]

    # Use this array to store allowed actions
    allowed_actions = []

    for i in range(len(sensors)):
        if sensors[i] > 0:  # at least one movement on this global direction is allowed  # NOQA
            max_movement = min(3, sensors[i])  # adjuest the allowed movement to 3 at most  # NOQA
            for move in range(1, max_movement + 1):
                allowed_actions.append((sensors_dirs[i], move))

    return allowed_actions
```

```
def calculatePreferedActions(self, location, allowed_actions):
    '''
    This funtion will be passed currect robot's location and all the
    allowed actions for the robot in the next step.

    This function will calculate, among these actions, which are prefered
    actions. A prefered action is the action that will send the robot to a
    cell it hasn't visited before.

    In other words, this function is a filtering process to keep the
    actions that will send the robot to unvisited cells.
    '''
    prefered_actions = []
    x, y = location
    for action in allowed_actions:
        direction = action[0]  # for example, direction == 'u'
        move = action[1]  # for example, move == 2
        delta = self.dir_move[direction]  # for example, delta == [0, 1]
        # calculate the new coordinate based on this action
        new_x = x + move * delta[0]
        new_y = y + move * delta[1]
        if not self.visitedGrid[new_x][new_y]:  # if cell on the new_coordinate hasn't been visited  # NOQA
            prefered_actions.append(action)

    return prefered_actions
```

Then we define another grid variable for the robot object, valueGrid, to store the value of each cell it has visited for the 1st time.

```
# This 2D array is used to record the permissibility of each cell
# For example, if the value for a cell is 1, that means the cell is only open on the top,  # NOQA
# and close on the other 3 sides. Initially, value of every cell is -1 to indicate unknown  # NOQA
self.valueGrid = [[0 for row in range(self.maze_dim)] for col in range(self.maze_dim)]  # NOQA
```

The valueGrid will be updated by a function called updateCellValue.

```
def updateCellValue(self, location, heading, sensors):
    '''
    This function is used in the first (exploratory) run.
    Especially, at the beginning of each step, the robot will call it,
    if and only if it hasn't visited the current cell before.

    Depending on the robot location, heading, and sensors, the robot will
    update the value (permissibility) for this cell, so that we know
    whether or not we can get into (or out of) this cell in a certain
    direction.

    For example, if the cell value is 9, it means this cell can be entered
    from the top or left side, but not from bottom or right side.
    '''

    permitted_dirs = []  # store the permitted travel directions into this cell  # NOQA

    # The cell is always visitable, from the opposite direction of the robot's heading  # NOQA
    permitted_dirs.append(self.dir_reverse[heading])

    # Know the directions of sensors in the global direction system
    global_dirs = self.dir_sensors[heading]

    for i in range(len(sensors)):
        if sensors[i] > 0:  # means that this travel direction is allowed
            permitted_dirs.append(global_dirs[i])

    cellValue = 0
    for direction in permitted_dirs:
        cellValue += self.dir_value[direction]

    # For example, if permitted_dirs == ['u', 'r', 'd'], cellValue is 7

    # Finally update this cellValue to valueGrid
    x, y = location
    self.valueGrid[x][y] = cellValue
```

We also define a heuGrid, which is used to store the heuristic value of each cell.

```
# This 2D array is used to record the heuristic for each cell
# The heuristic is a value for shorest movement from this cell to goal
# Initially, every cell has a heuristic value of -1 to indicate unkonw
self.heuGrid = [[-1 for row in range(self.maze_dim)] for col in range(self.maze_dim)]  # NOQA
```

Based on the valueGrid, at the end of the 1st run, we use the calculateHeuGrid () function to calculate the heuristic value for each cell based on the valueGrid.

```
def calculateHeuGrid(self):
    '''
    This function is used to generate the heuristic grid
    for the maze. The heuristic grid will be later referened using dynamic
    programming to find a global policy for this maze.

    This function is only called once, when it is the last step for the 1st
    run. This function will calculate the heuristic value for each cell
    and modify the robot.heuGrid variable.
    '''
    # A list used to keep track of the active cells
    current_active_cells = []

    # We initiate the heuristic calculation by assigning 0 to the central
    # 4 cells in the maze, and push them in the list
    for x in range(self.maze_dim/2 - 1, self.maze_dim/2 + 1):
        for y in range(self.maze_dim/2 - 1, self.maze_dim/2 + 1):
            self.heuGrid[x][y] = 0
            current_active_cells.append([(x, y), 0])

    while current_active_cells:  # while the list is not empty
        active_cell = current_active_cells.pop(0)
        x, y = active_cell[0]
        temp_heuristic = active_cell[1]

        if self.valueGrid[x][y] & 1:  # the active_cell is open on the top
            if y+1 <= self.maze_dim -1 and self.valueGrid[x][y+1] > 0:
                # make sure the adjacent cell exists and does have open on the bottom
                if self.heuGrid[x][y+1] == -1 or self.heuGrid[x][y+1] > temp_heuristic + 1:
                    # check if need to update the heuristic for the adjacent cell on the top
                    self.heuGrid[x][y+1] = temp_heuristic + 1
                    current_active_cells.append([(x, y+1), temp_heuristic + 1])

        if self.valueGrid[x][y] & 2:  # the active_cell is open on the right
            if x+1 <= self.maze_dim -1 and self.valueGrid[x+1][y] > 0:
                # make sure the adjacent cell exists and does have open on the left
                if self.heuGrid[x+1][y] == -1 or self.heuGrid[x+1][y] > temp_heuristic + 1:
                    # check if need to update the heuristic for the adjacent cell on the right
                    self.heuGrid[x+1][y] = temp_heuristic + 1
                    current_active_cells.append([(x+1, y), temp_heuristic + 1])

        if self.valueGrid[x][y] & 4:  # the active_cell is open on the bottom
            if y-1 >= 0 and self.valueGrid[x][y-1] > 0:
                # make sure the adjacent cell exists and does have open on the left
                if self.heuGrid[x][y-1] == -1 or self.heuGrid[x][y-1] > temp_heuristic + 1:
                    # check if need to update the heuristic for the adjacent cell on the bottom
                    self.heuGrid[x][y-1] = temp_heuristic + 1
                    current_active_cells.append([(x, y-1), temp_heuristic + 1])

        if self.valueGrid[x][y] & 8:  # the active_cell is open on the left
            if x-1>=0 and self.valueGrid[x-1][y] > 0:
                # make sure the adjacent cell exists and does have open on the right
                if self.heuGrid[x-1][y] == -1 or self.heuGrid[x-1][y] > temp_heuristic + 1:
                    # check if need to update the heuristic for the adjacent cell on the left
                    self.heuGrid[x-1][y] = temp_heuristic + 1
                    current_active_cells.append([(x-1, y), temp_heuristic + 1])
```

Finally, we define a policyGrid used to store the policy for the robot.

```
# This 2D array is used to record the policy for each cell
# The policy of each cell is a char: 'u', 'r', 'd', 'l',
# and can be interpreted as an arrow for the robot in the cell
# to move closer to the goal area. Initially they are set to '-'
self.policyGrid = [['-' for row in range(self.maze_dim)] for col in range(self.maze_dim)]
```

We used the calculatePolicyGrid () function to calculate the policy for each cell based on the heuGrid.

```
def calculatePolicyGrid(self):
    '''
    This function will be called at the end of 1st run,
    after the robot has calculated the heuristic grid.

    The function will actually modify the self.policyGrid, so that
    it finally contains correct policy for the robot to follow.
    '''
    for x in range(self.maze_dim):
        for y in range(self.maze_dim):
            cell_value = self.valueGrid[x][y]

            # No need to calculate the policy if the cell is already in goal area
            if x not in range(self.maze_dim/2 - 1, self.maze_dim/2 + 1) or y not in range(self.maze_dim/2 -
                # allowed_dirs is used to store the allowed move on this cell, for exmaple ['u','r']
                allowed_dirs = []

                # adajacent_heuristics is used to store the heuristic values for the allowed_dirs, for examp
                adjacent_heuristics = []


                if cell_value & 1:  # cell is open on the top
                    if y+1 <= self.maze_dim - 1:
                        if self.heuGrid[x][y+1] >= 0:
                            allowed_dirs.append('u')
                            adjacent_heuristics.append(self.heuGrid[x][y+1])

                if cell_value & 2:  # cell is open on the right
                    if x+1 <= self.maze_dim - 1:
                        if self.heuGrid[x+1][y] >= 0:
                            allowed_dirs.append('r')
                            adjacent_heuristics.append(self.heuGrid[x+1][y])

                if cell_value & 4:  # cell is open on the bottom
                    if y-1 >= 0:
                        if self.heuGrid[x][y-1] >=0:
                            allowed_dirs.append('d')
                            adjacent_heuristics.append(self.heuGrid[x][y-1])

                if cell_value & 8:  # cell is open on the left
                    if x-1 >= 0:
                        if self.heuGrid[x-1][y] >=0:
                            allowed_dirs.append('l')
                            adjacent_heuristics.append(self.heuGrid[x-1][y])

                if len(adjacent_heuristics) == 0:
                    print("x,y", x,y, "cellValue", cell_value)
                else:
                    min_heuristic = min(adjacent_heuristics)
                    indices = [i for i, val in enumerate(adjacent_heuristics) if val == min_heuristic]
                    index = random.choice(indices)
                    self.policyGrid[x][y] = allowed_dirs[index]
```

Refinement

In our algorithm, the value for each cell is only known if the cell has been visited. Therefore, if a cell has never been visited by the robot, then its value on the valueGrid remains 0 (initial value). Moreover, this cell will not have heuristic value in the heuGrid, and thus doesn't gain a policy for it.

However, in many situations, an unvisited cell can still be calculated its value, as long as all the four adjacent cells have been visited and therefore have values. For example, figure 8 shows an explanation for this.

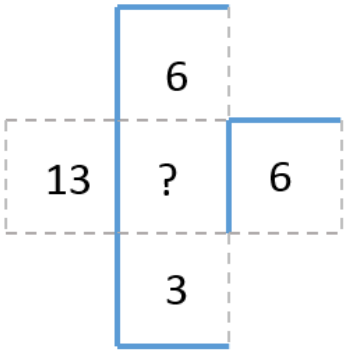


Figure 8. Calculate value of unvisited cell using its neighbours.

In figure 8, we have a cell (with a question mark) that is never visited by the robot. But all of its neighbours have been visited and have been calculated values. Then it is possible to get the value for the unvisited cell. The idea is that, we know the value for the cell on the top side (the top cell with value 6) and therefore knows the top cell has opening at its bottom, and that means our unvisited cell must have opening at its top. We can repeat this using all other neighbouring cells and in the end we know the value for this unvisited cell is actually 5.

Therefore, in our code, we implement a function called fixMissingCellValue () to calculate the value for an unvisited cell, if all its neighbours have been visited. Since it is quite long, please refer this function in the robot.py file.

Results

Model Evaluation and Validation

Since we have random operation in our algorithms, therefore it sounds a good idea to run our algorithm for each model several times to calculate the average performance of the model. We have three mazes here, each of different size, and we will test our model 10 times for each maze.

First of all, we use the smallest maze (12×12), which we actually saw earlier. Figure 9 shows this maze and the shortest path containing 17 steps. The performance of each test (10 totally) for this maze is showed in table 2.

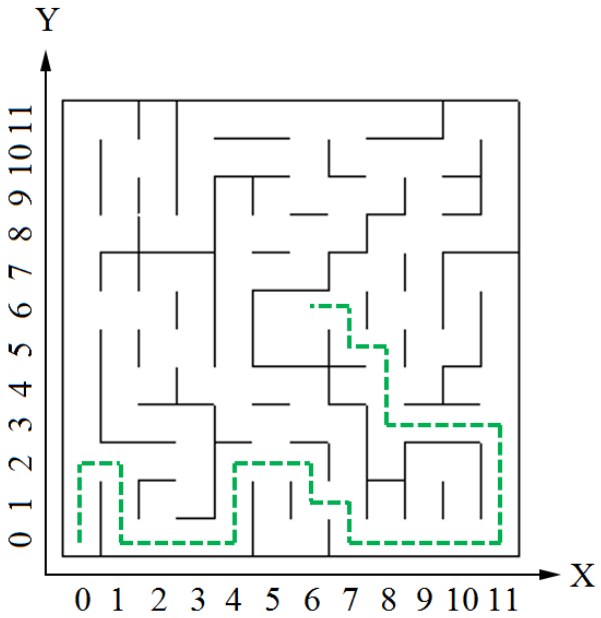


Figure 9. The smallest maze (12×12) with its shortest path in green (17 steps).

Test	Visited Cells	1st Run steps	2nd Run steps	Final Score
1	144 (100%)	501	20	36.7
2	143 (99%)	900	17	47
3	143 (99%)	900	17	47
4	144 (100%)	476	17	32.8
5	144 (100%)	728	17	41.3
6	144 (100%)	771	17	42.7
7	144 (100%)	805	21	47.8
8	144 (100%)	741	21	45.7
9	143 (100%)	900	17	47
10	144 (100%)	589	17	36.6

Table 2. 10 tests performances of our model in the smallest maze (12×12).

From table 2, we see that the robot is quite good to visit almost every cell in all of the test. Sometimes it spend only 476 steps to visit all the cells, but sometimes it need to terminate its 1st run for 900 steps without visiting them all. Besides, even if the robot has explored all the cells, the 2nd run steps still is not fixed. This is understandable, because when we generates our policyGrid, if a robot can travel from one cell to multiple adjacent cells with same smallest heuristic value, the policy for the that cell is randomly decided. The score for this maze is around 35 – 45.

Then we use the medium sized maze (14×14) to test our model. The maze and the shortest path (of 23 steps) is showed in figure 10. Table 3 shows the performance of 10 tests of our model in this maze.

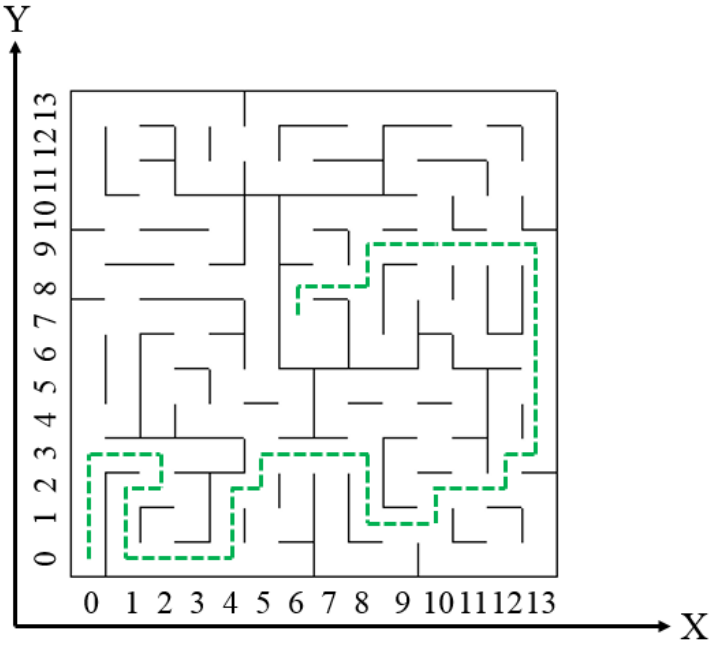


Figure 10. The medium sized maze (14×14) and shortest path in green (23 steps).

Test	Visited Cells	1st Run Steps	2nd Run Steps	Final Score
1	194 (99%)	900	24	54
2	196 (100%)	748	26	50.9
3	195 (99%)	900	28	58
4	194 (99%)	900	29	59
5	194 (99%)	900	23	53
6	196 (100%)	929	25	56
7	194 (99%)	900	27	57
8	194 (99%)	900	26	56
9	194 (99%)	900	23	53
10	173 (88%)	900	23	53

Table 3. 10 tests performances of our model in the medium sized maze (14×14).

From table 3, we find only 30% (3 out of 10) of the tests managed to visit all the cells in the 1st run, when for the smallest maze, that ratio is 80% from table 4. That makes sense, as the size of maze increases, it would be hard to visit all the cell at least once. Moreover most test need to spend 900 steps in the 1st run, because they are actually terminated by our 900 steps threshold. It might be interesting to see the test 6 visited 100% cells in the 1st run, but spent far more than 900 steps, and I would argue, because the central area (goal area) is the last area visited by the robot, so even it has spent 900 steps, it still needs to continue exploring until reaching the goal. Also surprisingly, the last test (test 10) only visited 88% of the cell but still managed to figure out the shortest path, and it's likely these 12% cells which are unvisited have nothing to do with the shortest path. The score for this maze seems

to be between 50 and 60.

Finally, let's see how our robot deals with the hardest and biggest maze (16×16). Figure 11 shows the third test maze and its shortest path (of 25 steps). Table 4 shows the performance of 10 tests of this maze.

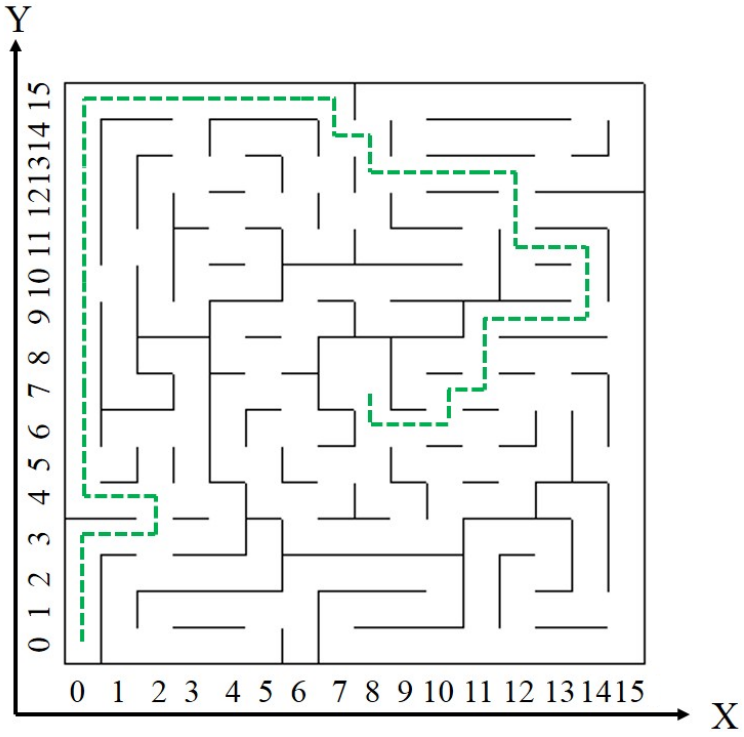


Figure 11. The large sized maze (16×16) and shortest path in green (25 steps).

Test	Visited Cells	1st Run Steps	2nd Run Steps	Final Score
1	246 (96%)	900	27	57
2	256 (100%)	660	30	52
3	223 (87%)	900	26	56
4	250 (97%)	900	27	57
5	255 (99%)	900	29	59
6	246 (96%)	900	29	59
7	238 (93%)	900	28	58
8	252 (98%)	900	25	55
9	234 (91%)	900	27	57
10	254 (99%)	900	27	57

Table 4. 10 tests performances of our model in the largest sized maze (16×16).

From table 4, we can easily see that it becomes really difficult for robot to explore all the cells, and only 10% (1 out of 10) test managed that. Moreover, the running time for the 1st run inevitably reach the 900 steps threshold, expect for the test that has visited all cells in 660 steps. The most interesting thing we can find here, is that

it seems the score for this largest maze is still between 50-60, quite similar with the result of maze 2 (14×14) even if the maze 2 is smaller? I think the reason for this is that even for solving maze 2, the robot almost always reaches the 900 steps threshold in the 1st run (because it is designed to be a perfectionist to explore all the cells possible), also the lengths of optimal shortest paths for maze 2 (23 steps) and maze 3 (25 steps) are not very different, rendering the scores for maze 2 and maze 3 to be surprisingly similar as well.

We also created the table 5 to summarize the **average** performance of our model for maze 1, 2 and 3. In the table 5, each value in the is the **average value** for percentage visited cell, 1st run steps, 2nd run steps, score, with their **standard deviations** displayed in parenthesis.

Maze	Visited Cells	1st Run Steps	2nd Run Steps	Score
Maze 1 (12×12)	99.8% (0.004)	731.1 (152.0)	18.1 (1.7)	42.5 (5.1)
Maze 2 (14×14)	98.1% (0.034)	887.7 (47.4)	25.4 (2.1)	55.0 (2.5)
Maze 3 (16×15)	95.6% (0.039)	876.0 (72.0)	27.5 (1.4)	56.7 (2.0)

Table 5. Average performance of our model for maze 1, 2, and 3.

From table 5, we can clearly see that, with the size of maze increases, the percentage of visited cell drops from almost 100% to 95.6%, with the standard deviation increasing. On the other hand, interestingly, the 1st run steps starts to increases with its standard deviation decreasing. This is understandable, because when the size of maze increases, the 900 steps threshold will be more frequently applied on our robot to terminate its 1st run, therefore the 1st run steps for maze 2 and maze 3 is almost always 900 steps. For the 2nd run steps, there is not a very significant difference between the shortest path distance and actual spending steps here. Finally, the score will increase as the size of the maze increases, with less variance. This is also caused by the fact that, robot almost needs to spend 900 steps in the 1st run for maze 2 and maze 3, and therefore, in fact, the average scores for maze 2 and maze 3 are quite similar, even if the maze 3 has 23% more cells than maze 2.

Justification

Previously, we have set up a benchmark for maze 1 (37-50), maze 2 (45-58), maze (56-70). Now here the actual average scores for these mazes are 42.5, 55.0, and 56.7, which are all actually within their expected ranges. Therefore, I would argue that our model did a successful job for solving these three mazes.

Conclusion

Free Form Visualization

Now I have come up with my customized maze, the test_maze_04, to make it a small challenge for our robot. This is still the maze of smallest size (12×12), but the configuration is a little harder now, since this maze has quite some small loops and dead ends, figure 12 shows the this customized maze with its shortest path (of 9 steps).

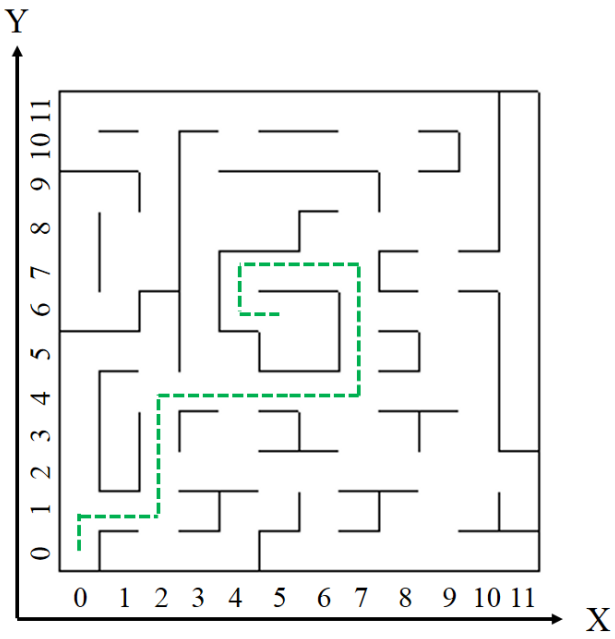


Figure 12. Our customized maze, with shortest path in green (of 9 steps).

From figure 12, we can see this maze actually has many small loops since walls are not spatially dense in this maze. Moreover, there are also many dead ends here, especially one on the right boundary of the maze, between the cell (11, 11) and (11, 4), where the only entrance to get into or out of this dead end is the cell (10, 7). This robot is also expected to solve this maze without problem especially not wasting too much on loops or dead ends in the 1st run. We still run our model on this maze 10 times and get the model performance in table 6. We also calculated the average performance at the end of the table.

Test	Visited Cells	1st Run Steps	2nd Run Steps	Score
1	144 (100%)	864	9	37.8
2	142 (98%)	900	12	42
3	144 (100%)	718	9	33.0

4	144 (100%)	560	12	33.1
5	144 (100%)	761	10	35.3
6	144 (100%)	504	12	28.8
7	144 (100%)	459	12	27.3
8	144 (100%)	879	9	38.3
9	144 (100%)	420	9	23
10	144 (100%)	643	10	31.4
avg	**99.8%**	**670**	**10.4**	**33.0**

Table 6. Model performance for our customized maze.

From table 6, we can see that first of all, 90% of tests (9 out of 10) managed to visit all the cells. Moreover, almost all 90% of tests (9 out 10) spend fewer than 900 steps in the 1st run. If we compare it to the model performance of maze 1 (of the same size), we find that for maze 1, only 70% (7 out of 10) tests finish 1st run within 900 steps. Moreover, the average steps in 1st run for maze 4 is 670, while that value is 731 for maze 1.

Based on the above comparison between maze 4 (customized) and maze 1 of the same dimension, I would argue that our robot is quite robot in solving maze with more loops and dead ends. In fact, our algorithm of "picking up a random allowed action preferably to an unvisited cell" in the 1st run, works well to avoid wasting time in the loops or dead ends. Our maze 4 has more loops and more dead ends, but that means it has fewer walls in the same, and is more easily to be explored thoroughly within the 900 steps threshold in the 1st run.

Reflection

This is actually an interesting but quite challenging project for me. To be perfectly honest, I have almost zero knowledge in robotics and artificial intelligence when I started this project. This project is special since it doesn't look seem to be solvable to any algorithms and techniques I have learned in the Udacity Machine Learning Nanodegree. However, this project is carefully curated, and I would the additional resources linking to the Udacity Robotics and AI course is awesome. This is how I have a basic understanding about how to use heuristics and dynamic programming to solve a maze for the robot.

With the above mentioned basic knowledge, when I dive into the project, it is still hard for me due to the specification of the project. The robot will be given two runs here to beat a maze. And in the first run, it starts exploring the maze without any knowledge or hints. Therefore, common shortest path algorithm cannot be applied here, since these algorithms need to be able to "see" the entire maze first. Therefore, I must somehow come up with my algorithm to let the robot explore

the maze as much as possible without wandering into loops or dead ends too much, this is the first most difficult thing for me. I am quite happy to be able to somehow sort it out.

Another difficult part in the project, is actually the allowance of "3 movements in direction at most" in a single step. Therefore, both in the 1st run and 2nd run, I need to find a way to make sure the robot "knows" this rule and will use this rule if available. Again, it was nice for me to find implementations for this rule.

Overall, it is a very worthwhile learning experience, where I need to almost start everything from scratch and look up resources when diving into a field I have never touched before.

Improvement

First of all, this is a simplified and discrete world. Therefore, in here we don't allow the robot to make a movement of, say 2.5 units, or turn clockwise 60 degrees. Also, the robot here needs to be exactly within one cell at each step, for example, it cannot be at a location between, say cell (0, 0) and (0, 1). Also, that means we had better consider the actual size of the robot, and the how thick the wall is. Moreover, the robot here turns around immediately, while in reality turning around still cost time, and sometimes it might be very cost behaviour. Therefore, our improved model can take these factors into account to deal with a real and continues world.

On the other hand, the robot can be improved to deal with situation, both in the 1st run and 2nd run, when it needs to go through a jagged path, which is explained in figure 13.

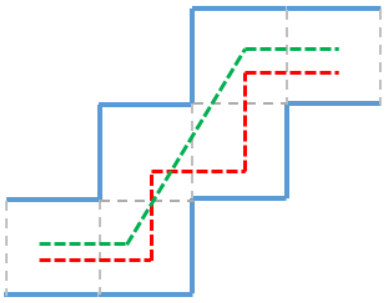


Figure 13. Issues with jagged path.

In figure 13, the red path is what our current robot will do to travel from the cell on the bottom-left to the top-right. But it might make more sense the robot follows the green path since turning around frequently is really costly. An improved model can considering implementing this.