Capstone Project Cristian Edwards

Machine Learning Nanodegree January 9, 2017

Plot and Navigate a Virtual Maze

Definition

Project Overview

Problem is about exploring a virtual maze with a robot, and then find a shortest path from the starting position to one of the centers of the maze. This is a simplification of the micro mouse competition.

This type of problem is very useful in practical applications, since finding shortest path arise many times in the reality, for example to provide the best routes for a fleet of trucks, planes, etc. A simpler example, is when we use Google Maps to calculate the best route between 2 points.

A good video to visualize micro mouse competitions can be found here.

Problem Statement

The robot runs in a maze, which is represented by a square matrix. Each cell of the matrix, can have up to four walls, the robot is only allowed to pass through cells in the direction where there are no walls.

The game play consists of two runs of the robot, the first one is aimed to “learn” about the maze, and the second one is to use the learned strategies to reach center position as fast as possible.

The game score is inversely proportional to the amount of steps taken in the 2 subsequent runs to reach the goal. However, the most significant contribution is the amount of steps in the second run. Therefore, achieving the shortest path in the second run is critical for a high score.

Metrics

As stated above, the least number of times to complete the maze, provides the maximum score.

A unit of time is considered a physical “move” of the robot, which can take none or as far as 3 physical steps in any valid direction of the maze.

In any of the 2 stages there can be a maximum of 1000 steps or moves the robot can make.

Analysis

Data Exploration

The maze can be represented as a square matrix, from 12 up to 16 rows or columns. The center of this matrix, is the goal to be reached. Any of the 4 positions in the center is considered a goal.

The robot has 3 visual sensors, aiming to the north, left and right positions. If we define the top side of the maze, to be the ‘North’ side, depending on the orientation of the robot, these sensor will aim to different poles.

Specifically we need to define every movement of the robot, so at each time we need to provide the number of movements and the rotation of the robot. The number of movements, can be either 0, 1, 2 and 3 moves. The rotation parameter can be a clockwise, counterclockwise and none rotation. Given these 2 parameters and the possibilities at any given time we can take up to 3 moves in any direction as long as this are valid moves. By valid moves, we mean not to hit a wall.

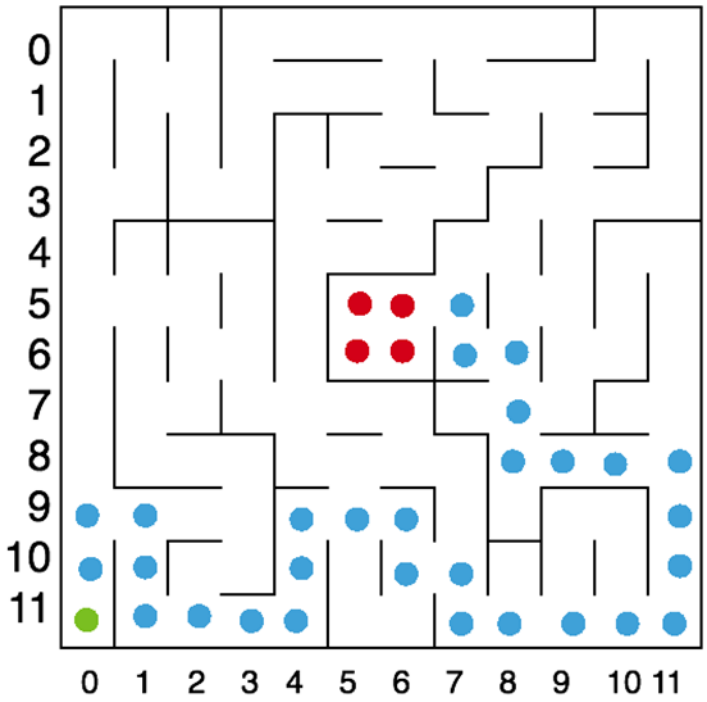
Exploratory Visualization

Maze 1

As we can see in the first maze, it’s of dimension 12. The row and column indexes used are by nature arbitrarily. We can see the initial position in green, centers in red, and the blue dots are the physical steps for a shortest path.

At the initial position, the robot is always facing north, given that the left and right walls are closed, and there are 11 free cells in the front position, the sensor from the robot will output the following list: [0, 11, 0].

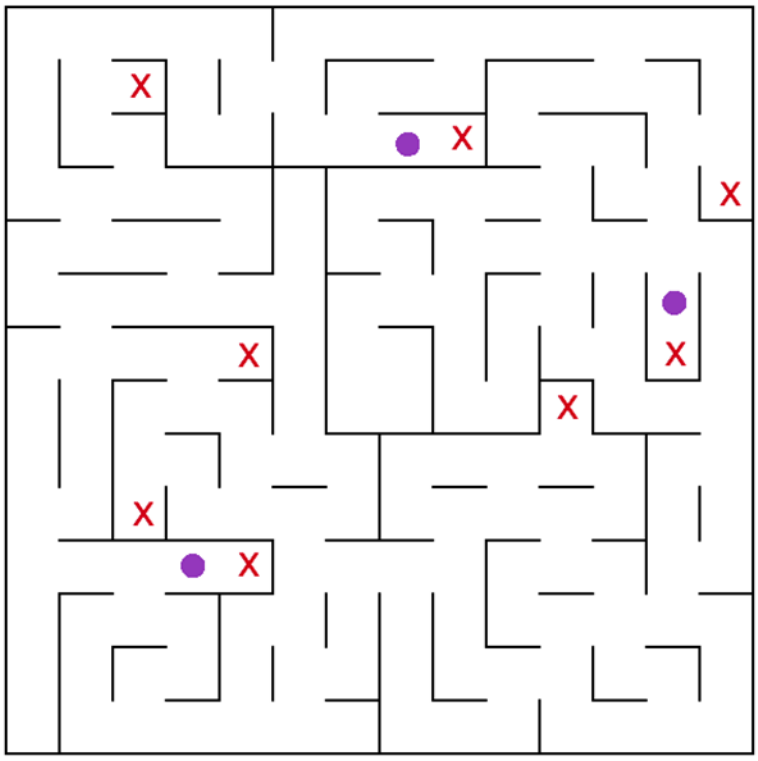
As you can see in the picture it takes 30 single steps to reach the goal area. However we know that the robot can move up to 3 cells per time, so if we count the number of moves the robot has to make, we calculate 17 moves.



Maze 2

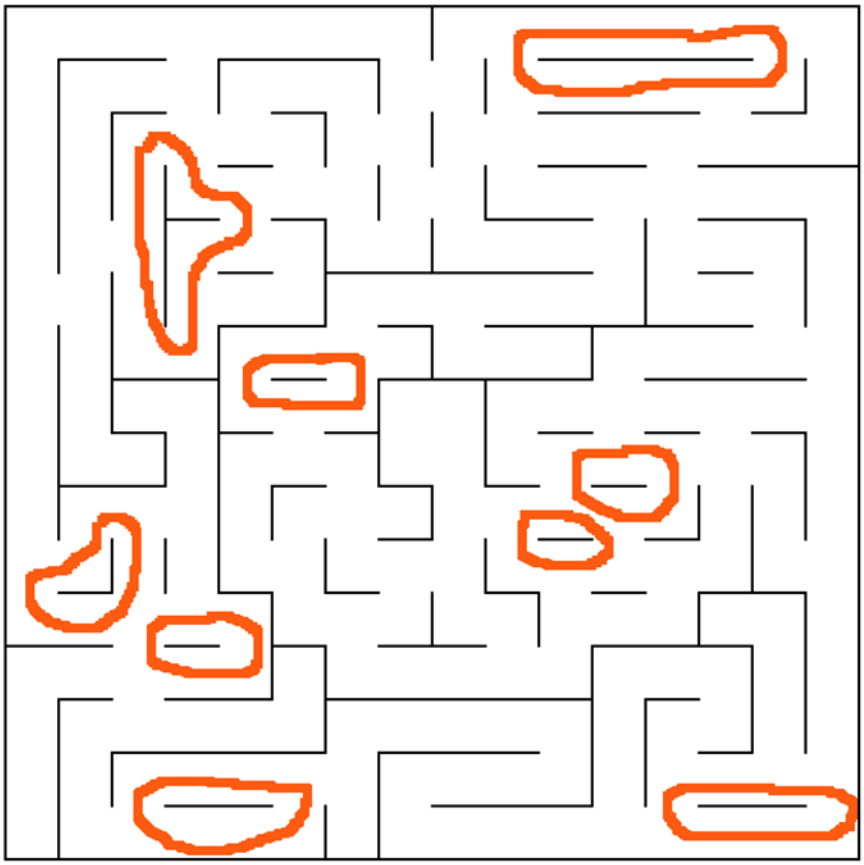
Maze 2 has a dimension of 14. In the pictures dead ends, are marked by an X, and cells adjacent to this dead ends are marked with a purple dot.

These dead ends, impose a difficulty since exploring them won´t add value to the shortest path algorithm. Since shortest path will never pass thorough them.



Maze 3

Maze 3 has a dimension of 16. In this maze cycles are depicted. Cycles can be a pitfall, since random exploration could fall into a never ending loop. To avoid cycles in the first exploration phase, we add a visit matrix, in order to prefer cells which have fewer visit times.



Algorithms and Techniques

The idea is to explore the entire maze in the first run, so in the second run we can compute Dijkstra´s algorithm to compute a shortest path.

To force the robot to explore the entire maze as soon as possible, a visit matrix is introduced, where it counts the number of visit a given cell or node has been visited. Initially is set to 0 in all values except in the initial position where it's set to 1. This matrix will discourage the robot to enter in loops, since it will be choosing always the cell with the least visits.

Fortunately with the aid of the visit matrix, the robot is able to explore the maze entirely without exceeding the imposed limit of 1000 robot movements.

For the second run, we assume that the maze is entirely discovered. Once the maze is discovered, the maze can be modeled as a graph, where the nodes are the cells and the edges are the valid moves the robot can make. Computing a shortest path in a graph is a thoroughly studied subject in computer science.

Dijkstra's algorithm is probably the most well-known algorithm to find shortest paths, so we will use this. The source node will be the initial position, hence Dijkstra's will provide the minimum number of moves to every cell including the target cells that are in the center. From these 4 target cells we select of the ones that have the minimum amount of steps.

Dijkstra´s algorithm

Let the node at which we are starting be called the **initial node** (in our case is the bottom left cell). Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some large initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

After having the minimum distances from the source to the center target, we can recursively build a path back.

Benchmark

Since we know that Dijkstra's algorithm if correctly implemented gives a shortest path, there is no comparison, since we can be sure that we have the global minimum amount of moves.

In the first run, situation is different since there is no clear and well known way to do this. To provide a point of comparison, the robot will choose a random move each time.

Methodology

Data Preprocessing

At the beginning we cannot pre-process data, since we have no initial data. We will be collecting the data on each step of the exploration.

Implementation

In order to make logic clearer, we define several auxiliary or utility methods described below.

Auxiliary methods

SHIFT is a list of orientations, e.g. going north means to subtract a row index, and leave the column index unchanged.

```
SHIFT = [[-1, 0],  # go north
         [0, 1],  # go east
         [1, 0],  # go south
         [0, -1]]  # go west
```

On the first exploration, we build the maze. The maze is represented by a square matrix of dimension equal to the maze dimensions that are given as an input. Each cell is represented as a 'word', where the cell: '0000b' represents a cell with 4 walls, and cell '1111b' represents a cell fully opened. Bit 1 means there is no wall, bit 0 means the wall is closed. From lexicographic left to write, the word represent left, bottom, right, up.

Initially, we set all cells of the maze to be closed (each cell has it's 4 walls), except for the initial up and the subsequent bottom north cell, since we know that wall must be clear in order to the robot to perform the initial move.

```
def build_maze(n):
    maze = [[0] * n for _ in range(n)]
    open_up_wall(maze, n - 1, 0)
    open_bottom_wall(maze, n - 2, 0)
    return maze
```

Since we start with the maze as 'closed' as possible, all we have to do is open walls on each step of the robot exploration, in order to achieve the representation of the actual maze. We use python bitwise operations, to set the correct bit of each wall, e.g. to open the bottom wall of a given cell, we set the bit in position 2 to 1.

```
def open_up_wall(maze, i, j):
    maze[i][j] |= 1 << 0

def open_right_wall(maze, i, j):
    maze[i][j] |= 1 << 1

def open_bottom_wall(maze, i, j):
    maze[i][j] |= 1 << 2

def open_left_wall(maze, i, j):
    maze[i][j] |= 1 << 3
```

Given an arbitrarily cell, we define the following method in order to state if it's valid to move in one step in an arbitrarily direction. Basically this method checks if the wall is open in the direction we want to move.

```
def can_move_one(maze, i, j, shift):
    n = len(maze[0])
    if not 0 <= shift < len(SHIFT):
        return False
    if not in_range(i, j, n):
        return False
    if not (0 <= i + SHIFT[shift][0] < n and 0 <= j + SHIFT[shift][1] < n):
        return False
    pos = 1 << shift
    return maze[i][j] & pos != 0
```

The update maze method is the heart of the exploration part, each time the robot makes a move, it shows the sensor information. We keep track of the heading and the location of the robot. With these 3 parameters, we have enough information to correctly free the appropriate walls of the maze.

Whenever we have a free wall, we should update at least 2 cells. For example if the robot is on the first cell, and there is one free cell in front of it, we should update the first cell top wall, and the following cell bottom wall.

For each orientation (north, east, south, west), we make a loop for the amount of free cells the robot's sensor see.

```
def update_maze(maze, heading, sensors, i, j):
    n = len(maze)
    north, east, south, west = 0, 0, 0, 0
    left, front, right = sensors[0], sensors[1], sensors[2]
    if heading == 'up':
        west, north, east = left, front, right
    elif heading == 'right':
        north, east, south = left, front, right
    elif heading == 'down':
        east, south, west = left, front, right
    elif heading == 'left':
        south, west, north = left, front, right
    if north > 0:
        open_up_wall(maze, i, j)
        for offset in range(1, north):
            i_ = i - offset
            if in_range(i_, j, n):
                open_bottom_wall(maze, i_, j)
                open_up_wall(maze, i_, j)
        open_bottom_wall(maze, i - north, j)
    if east > 0:
        open_right_wall(maze, i, j)
        for offset in range(1, east):
            j_ = j + offset
            if in_range(i, j_, n):
                open_left_wall(maze, i, j_)
                open_right_wall(maze, i, j_)
        open_left_wall(maze, i, j + east)
    if south > 0:
        open_bottom_wall(maze, i, j)
        for offset in range(1, south):
            i_ = i + offset
            if in_range(i_, j, n):
                open_up_wall(maze, i_, j)
                open_bottom_wall(maze, i_, j)
        open_up_wall(maze, i + south, j)
    if west > 0:
        open_left_wall(maze, i, j)
        for offset in range(1, west):
            j_ = j - offset
```

```
            if in_range(i, j_, n):
                open_left_wall(maze, i, j_)
        open_right_wall(maze, i, j - west)
```

Since in every exploration, we free some walls of the maze, in every location we can calculate the possible positions the robot can be in after its next move. The robot can move in every direction up to 3 cells. We can use the auxiliary method can_move_one, that returns true if and only if the robot can move from a given cell to a given direction. If we call this method recursively, for every direction, we can calculate the total amount of valid cells the robot can visit.

In addition, since we want to visit the least visited cells first, we form a tuple: amount of times visited and the location of the cell. So if we sort it, we will get the least visited valid cells first.

```
def possible_moves(maze, visit, i, j):
    options = []
    visit_m = False
    if visit:
        visit_m = True
    for shift in range(len(SHIFT)):
        off_i = SHIFT[shift][0]
        off_j = SHIFT[shift][1]
        if can_move_one(maze, i, j, shift):
            i_ = i + off_i
            j_ = j + off_j
            if visit_m:
                options.append((visit[i_][j_], i_, j_))
            else:
                options.append((i_, j_))
            if can_move_one(maze, i_, j_, shift):
                i_ += off_i
                j_ += off_j
                if visit_m:
                    options.append((visit[i_][j_], i_, j_))
                else:
                    options.append((i_, j_))
                if can_move_one(maze, i_, j_, shift):
                    i_ += off_i
                    j_ += off_j
                    if visit_m:
                        options.append((visit[i_][j_], i_, j_))
                    else:
                        options.append((i_, j_))
    if visit_m:
        options.sort()
    return options
```

After arbitrarily selecting one of least visited valid cells, we need to calculate the rotation and movement needed for the next moves. The robot doesn't understand cell locations, robot only wants to know if it should rotate or steer and if its wheels should spin forward or backwards. Since we keep track of the cell we are in, and now we know to which cell we want to go and we also keep track of the current heading of the robot, we can calculate rotation and movement.

For example if the robot it's facing north, and it must move 3 cells to the right, it should rotate 90 degrees clockwise, and advance forward/positively 3 times.

The following method does exactly what's described for every possible case.

```
def robot_moves(heading, i, j, i_, j_):
    if i != i_ and j != j_:
        print 'robot can move in one way only!'
    v_d = abs(i - i_)
    h_d = abs(j - j_)
    if v_d > 3 or h_d > 3:
        print 'robot can move maximum 3 spaces'
    if i != i_:
        if i > i_:
            inv = 1  # north
        else:
            inv = -1  # south
        if heading == 'up':
            return 0, inv * v_d
        if heading == 'right':
            return -90, inv * v_d
        if heading == 'down':
            return 0, inv * -v_d
        if heading == 'left':
            return 90, inv * v_d
    if j != j_:
        if j < j_:
            inv = 1  # east/right
        else:
            inv = -1
        if heading == 'up':
            return 90, inv * h_d
        if heading == 'right':
            return 0, inv * h_d
        if heading == 'down':
            return -90, inv * h_d
        if heading == 'left':
            return 0, inv * -h_d
    return 0, 0
```

In order to keep track of the internal current state of the heading of the robot, we make a method to change the heading based on it's actual heading and the future rotation. For example, if we are facing east and we have a future rotation of 90 degrees, we should update our internal heading to south or down.

```
def change_rotation(heading, rotation):
    if rotation == 90:
        if heading == 'up':
            return 'right'
        elif heading == 'right':
            return 'down'
        elif heading == 'down':
            return 'left'
        elif heading == 'left':
            return 'up'
    elif rotation == -90:
        if heading == 'up':
            return 'left'
        elif heading == 'right':
            return 'up'
        elif heading == 'down':
            return 'right'
        elif heading == 'left':
            return 'down'
    return heading
```

For the second run, we should have the complete maze discovered, therefore we can apply Dijkstra´s.

This method will provide 2 matrices, the first matrix comprises the minimal distances from the initial position to every other cell in the maze. The second matrix tells the ‘parent’ of every cell, this is useful to build the shortest path, since the other matrix only tells the minimum number of moves to get to any cell.

```
def dijkstra(maze):
    n = len(maze)
    dist_m = [[LARGE_INT] * n for _ in range(n)]
    prev_m = [[None] * n for _ in range(n)]
    visit_m = [[False] * n for _ in range(n)]
    to_visit = n * n - 1  # source node is visited
    dist_m[n - 1][0] = 0
    while to_visit > 0:
        i, j = get_coord_min_matrix(dist_m, visit_m)
        visit_m[i][j] = True
        to_visit -= 1
        neighbors = possible_moves(maze, None, i, j)
        for v in neighbors:
            i_, j_ = v
            if not visit_m[i_][j_]:
                alt = dist_m[i][j] + 1
                if alt < dist_m[i_][j_]:
                    dist_m[i_][j_] = alt
                    prev_m[i_][j_] = (i, j)
    return dist_m, prev_m
```

The following method, selects one of the centers with minimal distance, then builds up the 'path' from the selected center to the initial position. The path will be a list of coordinates, which the initial item will be the initial position and the final item the location of the selected center cell.

```
def shortest_path(maze):
    n = len(maze)
    dist_m, prev_m = dijkstra(maze)
    centers_dist = []
    for r in range(-1, 1):
        for c in range(-1, 1):
            i, j = n / 2 + r, n / 2 + c
            centers_dist.append((dist_m[i][j], i, j))
    centers_dist.sort()
    steps, i, j = centers_dist[0]  # i,j from optimal center
    path = [(i, j)]
    for _ in range(steps - 1):
        i, j = prev_m[i][j]
        path.append((i, j))
    path.reverse()
    return path
```

Main Methods

First Run

In the first run, we update or free walls from the maze based on the actual location, heading and sensors.

Then we calculate every possible move given the updated maze, the visit matrix and the cell coordinates. We randomly pick the the next cell to visit. If the cell hasn't been visited we have one cell less to visit. If there are no cells left to be visit then we stop the first run, then we calculate the shortest path or set of moves for the next run.

We update the visit matrix by adding one to the cell to be visit. We calculate the rotation and movement of the robot, then we update our internal heading. We also keep track of the amount of moves or times we have taken.

```
i, j = self.location[0], self.location[1]
u.update_maze(self.maze, self.heading, sensors, i, j)
visit_count, i_, j_ = random.choice(u.possible_moves(self.maze, self.visit, i, j))
self.location = [i_, j_]
if self.visit[i_][j_] == 0:
    self.to_discover -= 1
    if self.to_discover == 0:
        print 'Time took to explore: ', self.times
```

```
            self.first_exploration = False
            self.times = 0
            self.optimal_path = u.shortest_path(self.maze)
            self.heading = 'up'
            self.location = [self.n - 1, 0]
            return 'Reset', 'Reset'
self.visit[i_][j_] += 1
rotation, movement = u.robot_moves(self.heading, i, j, i_, j_)
self.heading = u.change_rotation(self.heading, rotation)
self.times += 1
```

Second Run

Second run is much easier, since we have calculated the coordinates of the optimal moves. In each step, we calculate the rotation and movement based on the shortest_path list.

We also keep track of the times and the current heading of the robot.

```
i, j = self.location[0], self.location[1]
i_, j_ = self.optimal_path[self.times]
self.location = [i_, j_]
rotation, movement = u.robot_moves(self.heading, i, j, i_, j_)
self.heading = u.change_rotation(self.heading, rotation)
self.times += 1
return rotation, movement
```

Refinement

The only refinement that we make, is on the first run. When calculating all the possible moves, instead of choosing the next move randomly, we choose one of the moves with the least visited times.

This is done using a visit matrix, which has the same dimensions as the maze matrix. It´s initialize with all the cells set to 0, except the one with the initial position. Each time the robot visit a cell, the cell values is incremented by 1.

When choosing the possible moves, we only prefer cells that have been visited the fewest.

Results

Model Evaluation and Validation

Maze 1

When random exploration is performed in this maze, almost every attempt we exceed the maximum of 1000 steps.

For example, in this run this below is a view of the visit matrix, as we can see there are many cells that are not visited by the robot. As it can be seen, it falls in cycle repeating several time the same cells.

	0	1	2	3	4	5	6	7	8	9	10	11
0	14	22	29	16	7	8	14	7	5	7	0	0
1	11	24	25	20	8	6	9	5	6	6	3	0
2	15	22	30	13	7	3	3	2	5	0	0	0
3	27	25	23	12	3	1	0	0	0	2	0	0
4	19	10	6	3	7	0	0	0	0	3	0	0
5	25	15	13	2	6	0	0	0	0	2	0	0
6	23	8	10	5	4	0	0	0	0	0	0	0
7	23	17	15	5	6	0	3	0	0	0	0	0
8	26	10	9	15	5	1	2	6	0	0	0	0
9	41	21	12	19	3	2	4	5	0	4	2	0
10	20	6	1	5	1	0	5	10	3	1	1	0
11	15	15	8	6	10	1	3	8	7	6	5	2

On the other hand, when we take account the amount of visits of each cell, the visit matrix looks like this:

	0	1	2	3	4	5	6	7	8	9	10	11
0	5	5	5	6	6	6	6	6	5	5	4	4
1	4	5	5	6	5	5	5	5	4	4	4	4
2	5	4	5	5	3	4	4	3	3	5	3	4
3	4	4	5	5	3	3	3	3	5	4	4	4
4	4	4	5	5	3	2	2	4	4	4	4	4
5	5	4	5	5	3	4	4	4	4	4	3	3
6	4	4	5	5	3	3	3	3	4	4	4	3
7	4	4	4	4	3	3	3	3	4	3	1	2
8	4	4	3	4	3	3	3	2	3	2	2	2
9	4	4	3	3	3	2	2	2	3	2	2	2
10	3	3	3	3	2	2	2	2	1	1	1	2
11	3	3	3	2	2	2	2	1	2	1	1	2

As we can see from the matrix above, each cell is visited, and the number of cells visited each time is well kept under 6 times. To discover the maze, the robot took 504 steps.

Since each cell is visited once, it means that the robot correctly explored the maze, leading us to the following maze, which represents exactly the same the test_maze_01.txt.

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

Since we have the complete maze structure, we can build a graph representation then run Dijkstra's algorithm, to obtain a path length of 17 moves. The list of moves of the shortest path is represented in the list below:

[(11,0), (9, 0), (9, 1), (11, 1), (11, 4), (9, 4), (9, 5), (8, 5), (8, 7), (11, 7), (11, 8), (11, 11), (8, 11), (8, 8), (6, 8), (6, 7), (5, 7), (5, 5)]

The task is complete and the score is 33.833

Maze 2

Exactly like the maze below, random exploration exceeds allotted time, resulting in a matrix with heavily amount of unvisited cells:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	1	1	0	0	0	1	0	0	1	0	1
1	1	0	0	0	3	2	8	6	6	1	1	2	5	0
2	0	1	2	1	1	3	8	3	6	1	1	0	3	1
3	1	1	2	0	1	4	0	0	0	0	0	0	1	1
4	1	1	1	2	0	7	0	0	3	1	4	1	5	6
5	5	9	5	4	2	9	0	0	0	0	2	0	1	3
6	13	13	7	7	5	9	0	0	0	0	2	1	2	7
7	13	8	8	5	12	7	0	0	0	0	1	1	1	7
8	12	8	5	5	18	8	10	4	3	6	4	3	9	10
9	25	8	2	17	18	24	13	7	7	8	5	4	6	2
10	38	29	33	28	19	12	2	4	3	1	1	2	3	3
11	19	17	13	9	8	17	2	1	0	2	1	5	3	4
12	13	7	6	5	5	18	7	1	3	1	2	1	1	1
13	12	10	15	13	24	25	12	1	0	0	2	0	1	1

On the contrary, when it explores by using the criteria of the least visit cell matrix, it takes 292 moves to visit every cell at least once.

We run our shortest path algorithm, to obtain the minimum amount of moves from the initial position to the center (6, 6) which is 22, and the exact coordinates to visit are the following:

[(13,0), (10, 0), (10, 2), (11, 2), (11, 1), (13, 1), (13, 2), (13, 5), (10, 5), (10, 8), (12, 8), (12, 10), (11, 10), (11, 12), (8, 12), (8, 13), (5, 13), (4, 13), (4, 10), (4, 8), (5, 8), (5, 6), (6, 6)]

The task is complete and the score is 41.733

Maze 3

Like in examples before, we run random exploration and it misses more than half of the cells, as seen below:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	19	8	11	22	7	10	6	3	0	0	0	0	0	0	0	0
1	19	19	11	15	3	0	4	0	2	0	0	0	0	0	0	0
2	17	11	18	12	6	5	3	2	1	0	0	0	0	0	0	0
3	25	15	13	20	5	5	12	1	2	0	2	5	1	5	1	1
4	15	17	20	0	1	2	2	1	9	5	9	9	1	0	0	0
5	15	12	19	2	0	3	6	6	20	8	6	13	3	3	2	0
6	9	11	20	3	0	0	0	0	19	15	16	2	4	2	3	2
7	12	7	1	4	0	0	1	0	0	0	0	2	1	1	1	0
8	12	7	1	3	0	0	0	0	0	0	0	0	0	0	0	0
9	16	8	13	4	0	0	0	0	0	0	0	0	0	0	0	0
10	13	5	17	2	0	0	0	0	0	0	0	0	0	0	0	0
11	18	10	24	9	10	0	0	0	0	0	0	0	0	0	0	0
12	12	7	13	2	7	1	0	0	0	0	0	0	0	0	0	0
13	4	13	28	11	13	11	0	0	0	0	0	0	0	0	0	0
14	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	6	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0

However when we take account the visit matrix, the algorithm takes 928 moves to discover every cell of the maze.

Running shortest path, we obtain a number of optimal moves of 25, and the path is given by:

[(15,0), (12, 0), (12, 2), (11, 2), (11, 0), (8, 0), (5, 0), (2, 0), (0, 0), (0, 1), (0, 4), (0, 7), (1, 7), (1, 8), (2, 8), (2, 9), (2, 12), (4, 12), (4, 14), (6, 14), (6, 11), (8, 11), (8, 10), (9, 10), (9, 8), (7, 8)]

The task is complete and the score is 55.967

Justification

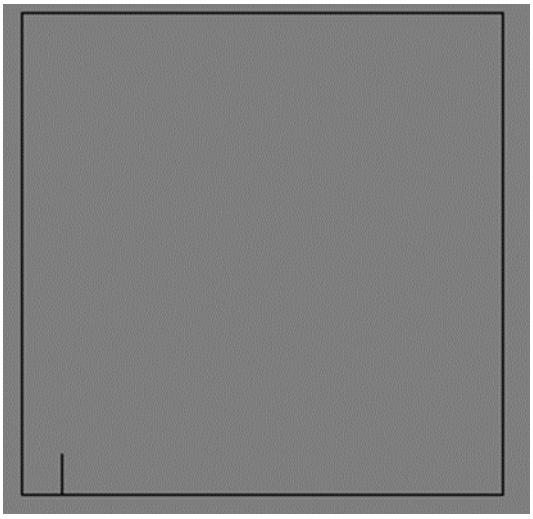
After ending the first run, and discovering the complete maze, we can easily compare the initial maze text file with the one that we discovered. Not surprisingly, these mazes are exactly the same.

Since we are able to actually obtain the initial maze, running a shortest path algorithm like Dijkstra's will certainly deliver a shortest path.

Conclusion

Free-Form Visualization

I try to provide the simplest maze I could imagine, every move would be possible, except the first one that should move only up. The maze has 12 dimension.



If we run, the random exploration, sometimes it will actually discover the complete maze. For example, in one test it discovered the maze in 984 moves. The optimal moves to reach the center is 4, and the optimal shortest path coordinate list is:

[(11, 0), (8, 0), (5, 0), (5, 2), (5, 5)]

The scored reached is 36.833.

However, if we run this again considering the visit matrix, we obtain an optimum with the visit matrix:

	0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1	1	1
9	1	1	1	1	1	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1	1	1	1	1	1
11	1	1	1	1	1	1	1	1	1	1	1	1

Each cell is visited exactly once, which is the best case for our algorithm, achieving a surprisingly high score.

The scored reached is 8.767.

Reflection

The most challenging part surprisingly comes in the first part, because the second part of finding a shortest path has a particular well known solution. It's actually not necessary to visit every node or cell of the maze to come up with a shortest path.

The implemented algorithm, does solve the problem. Nevertheless further optimization can be applied to the first part. Detecting dead ends, should improve the total score of the problem, since there is no need to discover those cells.

I find the optimization of the first path particularly challenge and interesting, since this problem should be reduced to find a shortest path at the first exploration. The person who actually achieves this, would have solve this problem optimally, if it can be achieved.

Improvement

Everything in this project is assumed to be discrete and perfect, in the real world of course, this isn’t the situation.

The actual code just assumes that when we tell the robot to go to a given cell, it will actually get there. This is not true in the reality, because a physical difficulty may arise, like for example slipping and colliding into a wall.

To confront this difficulties we should always be receiving the state of the robot, checking if giving an instruction actually results in that instruction.

Nevertheless, I don’t think there are continuous mazes that cannot be solved in discrete domains.