Capstone Project
Machine Learning Engineer Nanodegree

Plot and Navigate a Virtual Maze

Definition

Project Overview

Robotics is very important nowadays and brings technologies which can deal with automation machines that can take the place of humans in dangerous environments or manufacturing processes, or resemble humans in appearance, behavior and cognition. One of the main challenges in robotics is finding the optimal path through in the environment they operate.
In this project a robot mouse is taught to find the path in the maze on its own. The project was inspired by the micromouse competition.

Problem Statement

A robot mouse has a task to plot a path from a corner of the maze to its center. The robot may make multiple runs in a given maze. The first run should be used not only to find the center, but to explore as much maze as possible and figure out the best paths to the center. In subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned.
The problem lies in the machine learning area as it operates within the initially unknown world which needs to be explored. This prevents the classical search algorithms to perform well as it's required to do some initial exploration beforehands.
The role of Machine Learning in autonomous robots is described in the following papers: Reinforcement Learning for Autonomous Robot Navigation (see https://ieeexplore.ieee.org/document/833418/ for more details); Real-world reinforcement learning for autonomous humanoid robot docking (see https://www.sciencedirect.com/science/article/pii/S0921889012000814 for more details).

Scoring

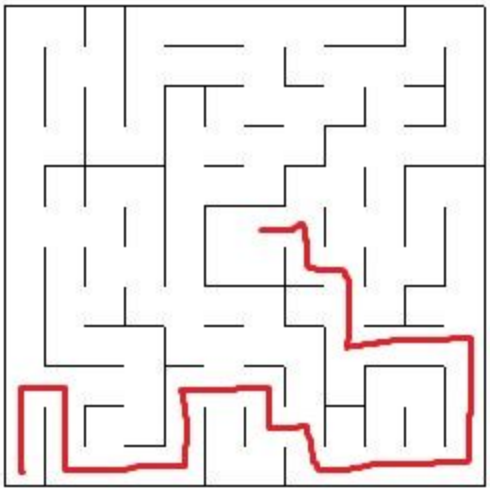
The sore for the robot-mouse is equal to the number of time steps required to execute the second run plus one thirtieth of time steps to execute the first run. A maximum of one thousand time steps is allowed to complete both runs for a single maze.

Analysis

Data Exploration and Visualization
Here are the descriptions of the robot and an environment.

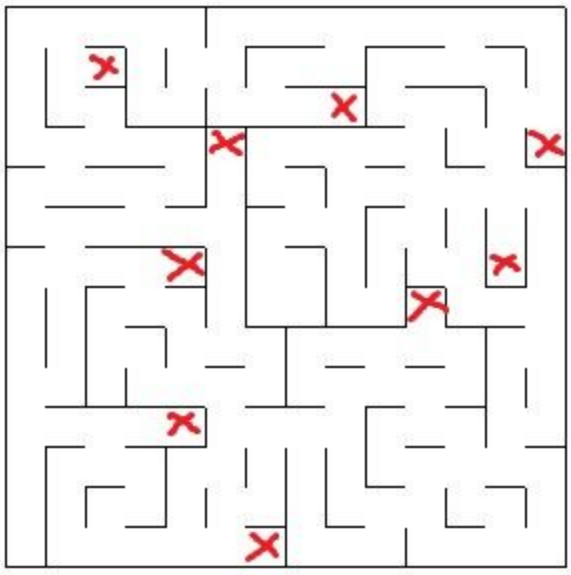
The maze exists on an n x n grid of squares, n is even. The minimum value of n is twelve, the maximum - sixteen. Along the outside perimeter of the grid, and on the edges connecting some of the internal squares, are the walls that block all movement. The robot will start in the square in the bottom-left corner of the grid, facing upwards. The starting wall will always have a wall on its right side and an opening on its top side. In the center of the grid is the goal room consisting of a 2 x 2 square. The robot must come here from its starting square as fast as possible.
Mazes are provided to the system via text file. On the first line of the text file is a number describing the number of squares on each dimension of the maze n. On the following n lines, there will be n comma-delimited numbers describing which edges of the square are open to movement. Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom (0*1 + 1*2 + 0*4 + 1*8 = 10). Due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom-left) corner of the maze.
I will be using the 3 mazes provided with the files in the starter code.

Maze 01 (12x12):
The start location is (0, 0). At the start location, the robot is facing north and it'll perceive the following sensory information: (0, 11, 0).



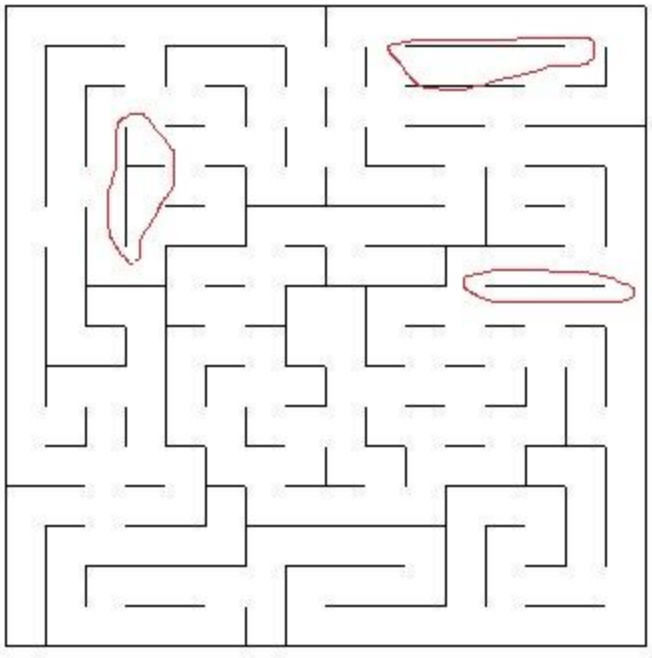
The optimal path from the start location (bottom left corner) is shown as a red line. The robot needs 30 single steps to reach to the goal. Considering that the robot is allowed to take up to 3 steps at a time, the minimum time steps required to reach the goal is 17.
Based on the path shown we can come to a conclusion that robot should prefer the straight paths with minimum turns in order to achieve the goal in the minimum amount of steps.

Maze 02 (14x14):



The maze contains the dead ends which are to be handled by the agent.

Maze 03 (16x16):



The maze contains the areas where the agent might be looping around. The exploration logic has to provide some mechanism to actually avoid looping for too long.

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

Algorithms and Techniques

To achieve the goal robot needs to do the following:

- Explore the maze.
- Find the optimal path.

The first step to do is to explore the maze in the best possible way. The output from the exploration step will be a graph which then can be fed to the search algorithm to do its "magic". Here's the list of the techniques I tried to explore:

- Random move
- Random move with dead end detection
- Counting the number of visits to expand the less visited area

Every exploration method I'm about to describe is doing one thing in common - it checks whether the agent can actually go to the specified location, based on the (rotation, movement) value produced. If it cannot go there, it'll just skip this iteration and will try to generate a new combination of (rotation, movement). This is needed to make the explorer update its belief state.

Random move exploration will generate the rotation/movement values at random with no consideration of the sensory information.

Random move with dead end detection will be working its way around dead ends. In current implementation it's done by returning the sequence (90, 0), which means that the agent will need to turn 90 degrees to the right and try to go from there. This works effectively to return on the path after reaching the dead end.

Counting the number of visits to expand the less visited area encloses having additional part of belief state where the values are the number of times the cell been visited. This helps to explore less explored areas first. This agent also has a chance to do the exploration at random with the probability of 0.2.

Every time, the robot perceives, it updates its belief state by putting a mask on the maze grid.

The maze is considered to be explored if there's a goal reached and % cells of the maze are explored.

The second bullet implies that there's already a known state-action space where to apply the algorithm. If we have it known, we can use any of the following algorithms: BFS, A* or Dijkstra

algorithm. All of them guarantee to find the shortest path. There're differences though described below:

- BFS. Gives a guarantee to find the shortest path between 2 points on the graph given enough memory. The implementation is pretty simple. But there's a serious drawback with the memory usage as it requires to keep the whole frontier in memory which might seriously affect the performance and even whether we may find an answer to our question. This one will be used as the benchmark solution.
- A* algorithm. Gives a guarantee to find the shortest path between 2 points on the graph given enough memory. The implementation is a bit more complex than the BFS as it requires the usage of priority queue data structure. Having this in mind the element which needs to be put there requires to have the comparison operators implemented. In our case the comparison will be done by using the path cost and heuristics. A* algorithm wouldn't be very beneficial comparing to BFS in terms of speed/memory consumption if one doesn't provide a valuable heuristics which is the key to avoid looking at non-relevant paths which gives some runtime and memory efficiency benefits.
- Dijkstra. Gives a guarantee to find the shortest paths from a given point on the graph to any other points given enough memory. The implementation requires to use some of the advanced data structures, e.g. IndexedMinPriorityQueue. Based on its purpose (to find the shortest paths from starting point to all other points on the graph), it will require to do a traverse through the whole graph, which might not be that beneficial if we are talking about finding the path from point A to B (from both memory and runtime perspective). However, if there will be a need in finding paths to multiple points, then this would be the best possible option.

Speaking of all the 3 algorithms, they are NP-hard as they all depend on the size of the state-action space, which means, we might not be able to get an answer to the question we ask. In terms of our task (find the path on the maze), the state-action space is not that large (because of the simplification of the problem, e.g. we have it discrete).
All these 3 algorithms can provide the optimal path to the goal only if the maze if fully explored. I decided not to proceed with Dijkstra algorithm because of it doing the redundant work during finding the optimal paths.

The underlying data structure to build the graph is implemented in the following way:

```
class Node:
    """
    The data structure for bookkeeping during the path search.
    """
    def __init__(self, parent, edge, cost=1, goal=None):
        """
        Ctor.
        :param parent: the parent Node.
```

```
        :param edge: edge on the Node.
        :param cost: cost to choose the Node.
        :param goal: the goal.
        """
        self.parent = parent
        self.edge = edge
        self.cost = cost
        self.goal = goal

    def get_cost(self):
        """
        Get the cost, based on the initial cost + the heuristics (squared distance
    to the goal).
        :return: the cost.
        """
        # we'll use the following heuristics: do penalize the agent for a turn.
        if self.parent is not None and (self.parent.edge.direction !=
    self.edge.direction):
            return self.cost + 1

        return self.cost


def __lt__(self, other):
    """
    The comparison operator (<).
    :param other: other node.
    :return: True if current node's cost is lower than the other node's cost.
    """
    return self.get_cost() < other.get_cost()
```

Benchmark

The performance of the agent will be compared to the one using BFS algorithm.

Methodology

Data Pre-processing

The maze specification and robot's sensory data are available and accurate, so there is no need in data pre-processing.

Implementation

All the implementation is done with python 3.6 (the test script was updated accordingly). Please refer to readme.md for more details on how to run the test script.

Supporting classes

Robot_utils.py

First of all I've implemented the wrapper for sensory information which helps with exploration stuff:

```
class SensorInterpreter:
    """
    Wrapper for the sensory information.
    """
    def __init__(self, sensors):
        """
        Ctor.
        :param sensors: sensory information in the given direction.
        """
        self.sensors = sensors

    def distance(self, rotation):
        """
        Distance to the walls in the given direction (based on the rotation).
        :param rotation: Rotation value (-90, 0, 90).
        :return: Distance to the walls in the given direction (based on the rotation).
        """
        return self.sensors[rotation_idx_dict[rotation]]

    def is_dead_end(self):
        """
        Checks if the agent reached the dead end.
        :return: True if the agent reached the dead end, False otherwise.
        """
        return max(self.sensors) == 0

    def is_one_way(self):
        """
        Checks if the agent can go one way.
        :return: True if the agent can go one way, False otherwise.
```

```
        """
        return self.sensors[0] == 0 and self.sensors[1] > 0 and self.sensors[2] == 0

    def can_go(self, rotation, movement):
        """
        Apply rotation and see if the agent can go in the given direction.
        :param rotation: Rotation value (-90, 0, 90).
        :param movement: Number of steps to go (the range is: [0, 3]).
        :return: True if the agent can move in the specified direction, False otherwise.
        """
        return self.distance(rotation) >= movement

    def get_perceived_cell_mask(self, direction):
        """
        Build the mask to apply to a cell, based on the information perceived.
        :param direction: heading direction.
        :return: the mask to apply to a cell, based on the information perceived.
        """

        directions = ['u', 'r', 'd', 'l']
        direction_idx_dict = {directions[i]: i for i in range(len(directions))}

        dir_idx = direction_idx_dict[direction]
        front = direction

        left_idx = dir_idx - 1
        if left_idx < 0:
            left_idx = len(directions) - 1
        left = directions[left_idx]

        right_idx = dir_idx + 1
        if right_idx >= len(directions):
            right_idx = 0
        right = directions[right_idx]

        mask = 0
        if self.sensors[0] == 0:
            mask = mask | dir_int_mask[left]

        if self.sensors[1] == 0:
            mask = mask | dir_int_mask[front]

        if self.sensors[2] == 0:
            mask = mask | dir_int_mask[right]

        return mask

    def __str__(self):
        """
        The string representation of the sensors (for debugging).
```

```
        :return: the string representation of the sensors.
        """
        return str(self.sensors)
```

Then I built the infrastructure for graph search algorithms (classes Graph, Edge and GraphSearch).
In order to do the grid -> graph conversion I implemented the following method:

```
def robot_position_2_graph_vertex(self):
    """
    Converts the robot's location to a vertex on the graph.
    :return: a vertex on the graph corresponding to robot's location.
    """
    return int(int(self.robot_pos['location'][1]*self.maze_dim) + self.robot_pos['location'][0])
```

Following are the implementations of Edge, Graph and GraphSearch.

```
class Edge:
    """
    Represents an edge for the graph data structure, simply the connection between 2 vertices.
    """
    def __init__(self, direction, u, v, w, contains_goal):
        """
        Ctor.
        :param direction: heading direction.
        :param u: start vertex.
        :param v: end vertex.
        :param w: weight.
        :param contains_goal: is there a goal on any side.
        """
        self.direction = direction
        self.w = w
        self.u = u
        self.v = v
        self.contains_goal = contains_goal

    def either(self):
        """
        Get a single vertex on the edge.
        :return: a single vertex on the edge.
        """
        return self.u

    def other(self, u):
        """
        Gets the vertex other than an input.
        :param u: the vertex on the edge.
        :return: the vertex other than an input.
        """
        if u == self.u:
```

```
            return self.v

        return self.u

    def weight(self):
        """
        Gets the weight of the edge.
        :return: the weight of the edge.
        """
        return self.w

    def direction(self):
        """
        Get the heading direction of the edge.
        :return: the heading direction of the edge.
        """
        return self.direction

class Graph:
    """
    Represents the directed graph data structure.
    """
    def __init__(self, V):
        """
        Ctor.
        :param V: number of vertices on the graph.
        """
        self.V = V
        self.edges = np.empty(V, dtype=list)
        self.edges_count = 0

    def is_connected(self, u, v):
        """
        Checks if 2 vertices are connected in any direction.
        :param u: the first vertex.
        :param v: the second vertex.
        :return: True if 2 vertices are connected, False otherwise.
        """
        if self.edges[u] is None or self.edges[v] is None:
            return False

        for e in self.edges[u]:
            if e.either() == v or e.other(u) == v:
                return True

        return False

    def connect(self, u, v, w, direction, contains_goal):
        """
        Connect 2 vertices if they are not connected.
```

```
        :param u: first vertex.
        :param v: second vertex.
        :param w: the weight of the edge.
        :param direction: the heading direction.
        :param contains_goal: whether or not the edge contains a goal.
        """
        if not self.is_connected(u,w):
            edge_v = Edge(direction, u, v, w, contains_goal)
            edge_u = Edge(dir_reverse[direction], v, u, w, contains_goal)

            if self.edges[v] is None:
                self.edges[v] = []

            if self.edges[u] is None:
                self.edges[u] = []

            self.edges[v].append(edge_v)
            self.edges[u].append(edge_u)

            self.edges_count += 2
```

This class gives the ability to actually find a path + convert it to the sequence of actions (rotation, movement). All we need to do to search effectively is to write an implementation of the graph traversal to do the actual search based on the algorithm we choose.

```
class GraphSearch:
    """
    A base class for the search algorithms on the graph.
    """
    def __init__(self, graph, starting_point=0):
        """
        Ctor.
        :param graph: Graph ready to be explored.
        :param starting_point: The starting point on the graph.
        """
        self.graph = graph
        self.starting_point = starting_point
        self.visited = [False for i in range(graph.V)]
        self.path = []

    def search(self):
        """
        Search on the graph.
        :return:
        """
        node = self.build_node()
        self.path = self.convert_node_to_path(node)

        # print the debug info
```

```
        if len(self.path) > 0:
            print("the cost of the path is: {}".format(node.cost))
            print("the number of moves is: {}".format(len(self.path)))

    def build_node(self):
        """
        Builds the goal node.
        :return: the goal node if the goal is found, Nothing otherwise.
        """
        raise NotImplemented

    @staticmethod
    def convert_directions_to_rotation(previous_direction, current_direction):
        """
        Builds the rotation value based on the 2 directions.
        :param previous_direction: previous direction.
        :param current_direction: current direction.
        :return: the rotation value in degrees: [-90, 0, 90].
        """
        if previous_direction == current_direction:
            return 0

        directions = dir_sensors[previous_direction]

        if current_direction == directions[0]:
            return 90
        elif current_direction == directions[2]:
            return -90

        return 180

    def convert_node_to_path(self, node):
        """
        Converts the goal node to the path ((rotation, movement) sequence used by the agent).
        :param node: the goal node.
        :return: the list of the (rotation, movement) tuples to be used by the agent.
        """
        path = []

        if node is not None:
            previous_direction = node.edge.direction

            while node is not None:
                current_direction = node.edge.direction

                if len(path) == 0:
                    object_to_add = [0, 1]
                    path.insert(0, object_to_add)
                else:
                    if current_direction == previous_direction:
```

```
                if path[0][1] < 3:
                    path[0][1] += 1
                else:
                    rotation = path[0][0]
                    movement = 1
                    path[0][0] = 0
                    object_to_add = [rotation, movement]
                    path.insert(0, object_to_add)
            else:
                rotation = self.convert_directions_to_rotation(previous_direction, current_direction)
                path[0][0] = rotation
                rotation = 0
                movement = 1
                obj_to_add = [rotation, movement]
                path.insert(0, obj_to_add)

        previous_direction = current_direction
        node = node.parent

    return path
```

All the perceived information is stored in the class MazePerceived.

```
class MazePerceived:
    """
    The representation of the maze perceived by the robot.
    """
    def __init__(self, dim):
        """
        Ctor.
        :param dim: the dimension of the maze.
        """
        self.shape = (dim, dim)
        self.explored_space = np.zeros(self.shape, int)
        self.explored = np.zeros(self.shape, int)
        self.explored_cnt = 0

    def update_cell(self, robot_pos, sensor):
        """
        Updates the cell with the sensory information.
        :param robot_pos: robot position and heading.
        :param sensor: sensory information (distance to the walls).
        :return: no return value.
        """
        if self.explored[robot_pos['location'][0]][robot_pos['location'][1]] != 1:
            mask = 0
            sensor_heading = dir_sensors[robot_pos['heading']]

            for i in range(len(sensor)):
```

```
            if sensor[i] == 0:
                mask |= dir_int_mask[sensor_heading[i]]

        self.explored_space[robot_pos['location'][0]][robot_pos['location'][1]] |= mask
        self.explored[robot_pos['location'][0]][robot_pos['location'][1]] = 1
        self.explored_cnt += 1

    def get_cell(self, position):
        """
        Gets the value of the cell.
        :param position: Position on the grid.
        :return: the value of the cell.
        """
        return self.explored_space[position[0]][position[1]]

    def check_cell_explored(self, position):
        """
        Checks if the current cell is explored (to be used with the search algorithms).
        :param position: Position on the grid.
        :return: True if the current cell is explored, False otherwise.
        """
        return self.explored[position[0]][position[1]] == 1

    def is_explored(self):
        """
        Checks if the maze can be marked as explored.
        :return: True if the maze is explored, False otherwise.
        """
        goal_bounds = [int(self.shape[0] / 2) - 1, int(self.shape[0] / 2)]

        if self.explored_cnt < 5*(self.shape[0] ** 2)/6:
            return False

        return self.check_cell_explored([goal_bounds[0], goal_bounds[0]]) \
               or self.check_cell_explored([goal_bounds[0], goal_bounds[1]]) \
               or self.check_cell_explored([goal_bounds[1], goal_bounds[0]]) \
               or self.check_cell_explored([goal_bounds[1], goal_bounds[1]])

    def is_permissible(self, cell, direction):
        """
        Returns a boolean designating whether or not a cell is passable in the
        given direction.
        :param cell: Cell is input as a list.
        :param direction: Directions may be
        input as single letter 'u', 'r', 'd', 'l', or complete words 'up',
        'right', 'down', 'left'.
        :return: True if a cell is passable in the given direction, False otherwise.
        """
        try:
            return (self.explored_space[tuple(cell)] & dir_int_mask[direction] != 0)
```

```
except:
    print ('Invalid direction provided!')
```

Refinement

To be able to explore the maze I used 2 exploration classes: random move with walls and dead end detection and random move with visit counter. It turned out that it's possible to explore the first maze (12x12) with both having a slight difference in performance.
The second and the third turned out to be a tough ones for random move with walls and dead end detection class as the success rate is very low (33% for 2nd and 20% for the 3rd), considering that the random move with visit counter has the rate as (90% for 2nd and 80% for the 3rd).

The number of exploration steps to explore % of the maze cells are shown in the following table.

Maze number / Explotation method	**Random move with walls and dead end detection (number of moves + success rate)**	**Random move with visit counter (number of moves + success rate)**
01	578 (success rate is 100%)	565 (success rate is 100%)
02	311 (in most cases can't reach the goal; the success rate is 33%)	708 (the success rate is 90%)
03	662 (in most cases can't reach the goal; the success rate is 20%)	927 (the success rate is 80%)

Looking at this table, we can come to the conclusion that using the random move with walls and dead end detection doesn't give us the acceptable success rate. However, it was the first method of choice for me as if we are talking about maze 01, it shown a very promising results. Moving to a more complex mazes it started giving a lot of failures because of cycles and dead end zones which were not avoided after their exploration.
To help with this, I introduced the random move with visit counter, which actually lets the agent to choose within the actions letting the agent to go to less explored cells of the maze. It shows pretty acceptable success rate if we want to explore % part of the maze. In order to increase the success rate we can decrease the ration of explored cells in the maze having the goal explored, but this will lead to sub-optimal or not optimal paths found.

So, I ended up with the random move with exploration counter exploration in my final implementation because of the properties described above.

Testing Robot

I slightly modified the parameters to run the tester.py script. In addition to the maze file name if also takes another argument with the name of the algorithm you'd like to use (it accepts one of the following options: **bfs, astar, dijkstra**), so the command will look as follows:
Tester.py <maze_name> <algorithm name>

Results

Model Evaluation and Validation

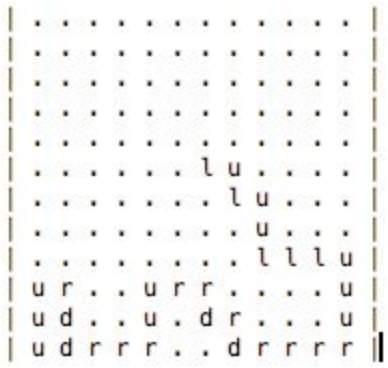
The optimal moves for the mazes are as follows:

Test maze #	Path cost	# of moves
01	30	17
02	43	22
03	49	27

The detailed sequences of (rotation, movement) are shown below for every maze.

Maze 01:
Optimal path:
the cost of the path is: 30
the number of moves is: 17
[[0, 2], [90, 1], [90, 2], [-90, 3], [-90, 2], [90, 2], [90, 1], [-90, 1], [90, 1], [-90, 1], [0, 3], [-90, 3], [-90, 3], [90, 2], [-90, 1], [90, 1], [-90, 1]]

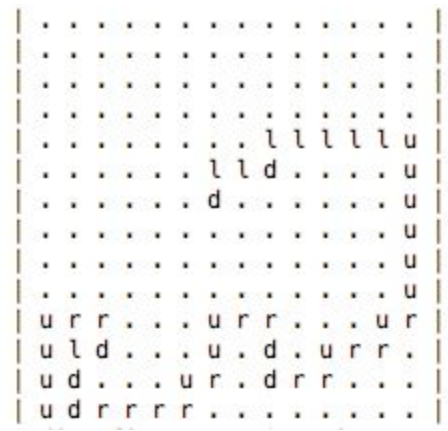
Nodes explored for A*: 141
Nodes explored for BFS: 143



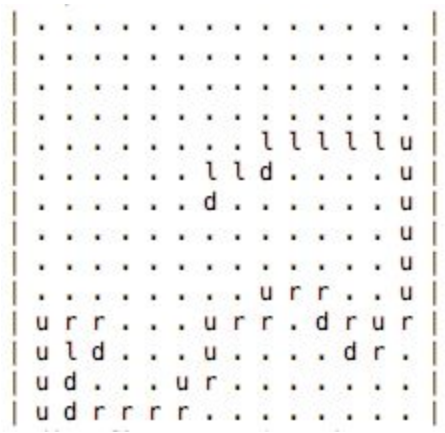
Non-optimal path can be as follows:
the cost of the path is: 34
the number of moves is: 20
[[0, 2], [0, 3], [0, 3], [0, 3], [90, 1], [90, 1], [-90, 1], [90, 2], [-90, 1], [-90, 3], [90, 1], [0, 3], [90, 1], [-90, 2], [90, 2], [90, 1], [-90, 1], [90, 1], [-90, 1], [90, 1]]

Maze 02:
Optimal path:
the cost of the path is: 43
the number of moves is: 24
[[0, 3], [90, 2], [90, 1], [90, 1], [-90, 2], [-90, 1], [0, 3], [-90, 1], [90, 1], [-90, 2], [90, 2], [90, 2], [-90, 2], [-90, 1], [90, 2], [-90, 1], [90, 1], [-90, 3], [0, 3], [-90, 2], [0, 3], [-90, 1], [90, 2], [-90, 1]]

Nodes explored for A*: 191
Nodes explored for BFS: 195



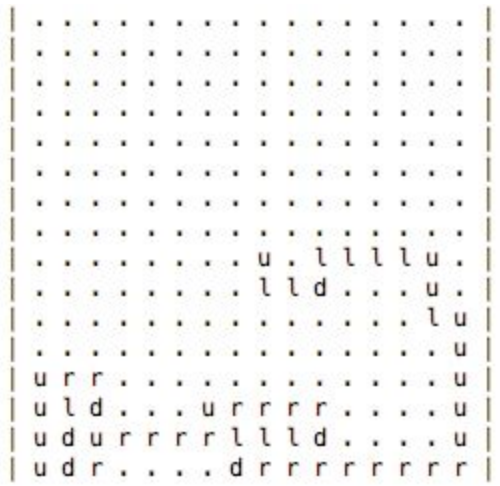
The sub-optimal path is:
the cost of the path is: 43
the number of moves is: 26
[[0, 3], [90, 2], [90, 1], [90, 1], [-90, 2], [-90, 1], [0, 3], [-90, 1], [90, 1], [-90, 2], [90, 2], [-90, 1], [90, 2], [90, 1], [-90, 1], [90, 1], [-90, 1], [-90, 1], [90, 1], [-90, 3], [0, 3], [-90, 2], [0, 3], [-90, 1], [90, 2], [-90, 1]]



Maze 03:
Optimal path
the cost of the path is: 49
the number of moves is: 26
[[0, 3], [90, 2], [-90, 1], [-90, 2], [90, 2], [0, 3], [0, 3], [0, 3], [90, 1], [0, 3], [0, 3], [90, 1], [-90, 1], [90, 1], [-90, 1], [90, 1], [-90, 3], [90, 1], [-90, 2], [90, 2], [90, 3], [-90, 2], [90, 1], [-90, 1], [90, 2], [90, 1]]

Nodes explored for A*: 249
Nodes explored for BFS: 255

Sub-optimal path
the cost of the path is: 49
the number of moves is: 27
[[0, 3], [90, 2], [90, 1], [90, 1], [-90, 2], [-90, 1], [-90, 1], [90, 1], [0, 3], [-90, 1], [90, 1], [0, 3], [90, 1], [90, 3], [-90, 1], [-90, 2], [0, 3], [0, 3], [-90, 2], [0, 3], [-90, 1], [90, 2], [-90, 2], [-90, 1], [90, 1], [0, 3], [90, 1]]



The numbers for optimal paths can be achieved by either A*, BFS or Dijkstra algorithm if the robot explored the whole maze in the first run. I was able to achieve them if the agent perceives at least % of the maze.
If the exploration has been done poorly (even having the goal), the results might be different as the graph build might not contain the optimal path to the goal. The non-optimal moves are shown to give a point. If only % of the maze is explored, one might end up with the sub-optimal moves shown. However, it makes the exploration much faster.

A* gives a better visited nodes count comparing to BFS or Dijkstra as it doesn't need to reach out to all nodes when searching for a goal (considering we have a good heuristics). The comparison on the explored nodes is shown above. Even though the difference is very small, it just means that the graph is too small to give a better picture. If there would be a dense graph, the difference might be very large.

One of the main problems was the exploration logic as it's the most complex and important part of the agent (considering that the search algorithms are well-known as well as their properties and pros/cons, which you can find in the *Algorithms and Techniques* part). I ended up with the counting agent with random moves which is described in details in the *Algorithms and Techniques* part.
All the exploration agents do explore at least % of the maze cells or they fail otherwise. This value is set in order to be able to find the optimal path on almost every run.
However, this also means that there might be some fail rate for every exploration agent available as the coverage rate might be much less than %. Setting the exploration value to a smaller value

may help to reduce failures, but the agent might end up finding non-optimal path which may lead to a smaller score at the end of the run.

Conclusion

When building up the agent for that micromouse the initial challenge was to divide the agent's actions into 2 parts and deal with those separately. The parts are: exploration and finding the optimal path.

Exploration. Building the exploration part was very challenging as the robot needs to explore the biggest part of the maze in the least possible time. I explored blind random move, random move with dead end detection and random move with counting the visits. Both random move with walls and dead end detection and random move with counting the visits can be used for the exploration. However, the random move with counting the visits shows a better performance on every maze in terms of success rate on every maze explored (please see the refinement section for more details).

When doing the implementation and further testing, I've come up with several classes dealing with exploration. They all extend the common class, Exploration, which actually returns the exploration step and keeps track of robot's position, explored space, etc.

The exploration class became the internal part of the Robot which actually uses it as the layer for doing the exploration job, then it builds the graph, based on the explored space and feeds it into the graph search algorithm which gives the path as an output. Robot has the logic to map every grid cell to a vertex on the graph.

Every class which extends Robot implements does a simple thing - it applies the path finding algorithm of choice to actually find that path. There're 2 active classes available: RobotBFS and RobotAStar. Both allow to find the optimal path given the maze fully explored. However, I ended up with suggestion to use A* in favor of BFS because of the smaller number of nodes explored during the search. I know, that with the simplified state-action space we have in our case, it doesn't matter which algorithm to use, just because the performance improvement (both runtime and memory) aren't that large, but if we move to a much bigger maze or to continuous space it will get more important.

Having the agent use the random move with counting the visits for exploration and A* for finding the optimal paths gives a pretty good results in finding the optimal and sub-optimal paths on the maze which concludes that the agent does its job given the current maze specification.

Speaking of real micro-mouse competition, the current implementation cannot be used there without refinement. The current task lies in the discrete domain which simplifies things a lot. The implementation needs to be adjusted to the continuous state and action domain which introduces much more complexity.

Moving to continuous domain raises lots of different problems - how to treat the sensory information, considering it might have some errors in there, how to make sure the robot sticks to the needed trajectory, keeps the speed at the needed level (can use PID controller there), etc which is really challenging. All this appears to be more complex than just finding optimal path

using any of the algorithms I described. There're a lot of challenges in building the robot, putting the right sensors (which might rotate to give 360 degrees view for better exploration experience), making sure, it uses proper motors and that it's as light as possible to achieve the best performance.