Robot motion planning

Ilya Semenov

16 June 2017

Plot and Navigate a Virtual Maze

Capstone project report for Udacity machine learning engineer nanodegree

Project description

The project takes inspiration from Micromouse competitions, wherein a robot mouse is tasked with plotting a path from a corner of the maze to its center. The robot mouse may make two runs in a given maze. In the first run, the robot mouse tries to map out the maze to not only find the center, but also figure out the best paths to the center. During the second run, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned.

Maze solving is a well known academic problem in the robotics field, some approaches described, for example, in following papers:

- Maze Solving Algorithms for Micro Mouse
- An Efficient Algorithm for Robot Maze-Solving
- A New Shortest First Path Finding Algorithm for a Maze Solving Robot

Project requirements

Requirements

- Python 2.7.x
- Numpy library

Deliverables

Project submission includes:

- PDF report of algorithms used and approaches taken to problem (this file)
- Python scripts for testing virtual robot (**robot.py** and **graph.py**)
- README text file entitled "Plot and navigate a virtual maze" describing the submitted files

The submission is packed in a single .zip archive.

I. Problem statement

Maze definition

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 walls that block all movement. The robot will start in the square in the bottom- left corner of the grid, facing upwards. The starting square will always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the center of the grid is the goal room consisting of a 2 x 2 square; the robot must make it here from its starting square in order to register a successful run of the maze.

Robot definition

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.

Complement to Udacity robot definition

- sensor range in all directions is three cells for all three sensors
- robot recieves sensor data only when it reaches a final cell of movement sequence (for example, if robot moves two cells forward it recieves sensor data from the second cell only)

Note

- according to robot definition from above diagonal moves are forbidden

Scoring

On each maze, the robot must complete two runs. In the first run, the robot is allowed to freely roam the maze to build a map of the maze. It must enter the goal room at some point during its exploration, but is free to continue exploring the maze after finding the goal. After entering the goal room, the robot may choose to end its exploration at any time. The robot is then moved back to the starting position and orientation for its second run. The robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. A maximum of one thousand time steps are allotted to complete both runs for a single maze.

Robot's goal

Using the maze configuration data collected during explorational phase move from start position to the center of the maze as quickly as possible during the final phase.

Inputs

Maze configuration

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). Note that, due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom-left) corner of the maze.

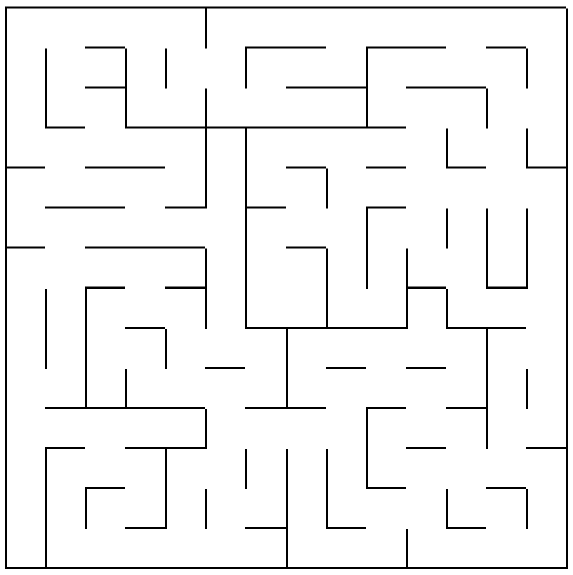

Figure 1: 14x14 Micromouse maze. One of the test mazes provided by Udacity

Robot interface

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

Starter code

Robot and maze specifications along with other support code provided by Udacity will be downloaded from the following link.

Data preprocessing

No data preprocessing required in a given project pipeline.

II. Analysis

General review

According to task description there are only three available maze sizes: 12x12, 14x14 and 16x16 cells. Regardless of the size each given maze has 4 cells goal room without inner walls in the center of the maze and the starting point is always located at the down left corner of the maze. Robot movement algorithm must be invariant to maze size.

It is assumed that the robot movement and rotation are perfect. Any valid robot movement may or may not include rotation 90 degrees clockwise or anticlockwise and than up to three cells movement phase forwards or backwards. Any valid robot movement with or without rotation takes exactly one time-step.

Robot interface implementation provided by Udacity prevents incorrect moves, that are, for example, moves that cause the robot to hit the wall. Since the robot interface does not provide coordinates inside the maze (on each step we have sensors info only) the algorithm should correctly maintain robot's position inside the maze and avoid unacceptable moves propositions since it would cause synchronization loss between actual robot position inside the maze and robot position that used by the algorithm internally. Synchronization loss between real and internal positions would almost definitely cause the robot to fail in final phase while trying to reach target cells.

Let consider the simplest 12x12 maze given by Udacity (Figure 2). Optimal path outlined with green color and takes 17 time-steps. To be more precise the path for this maze is as follows (in each tuple first letter denotes rotation and second number denotes No of cells to move):

(F,2), (R,1), (R,2), (L,3), (L,2), (R,2), (R,1), (L,1), (R,1), (L,3), (F,1), (L,3), (L,3), (R,2), (L,1), (R,1), (L,1).

"F" stands for forward, "L" stands for left turn and "R" stands for right turn accordingly.

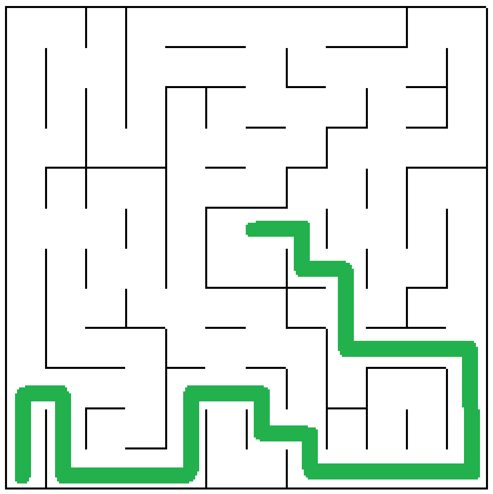

Figure 2: Optimal solution for Maze 1 from test mazes set provided by Udacity

Observations from Figure 2:

1. One may immediately note that most moves in an optimal path assume left or right turns.
2. Path to the center is definitely not straightforward and includes portions where robot moves in an opposite direction from the maze center.
3. During each step robot only has input from three sensors denoting the distance to the nearest wall in this direction, thus at every step, maybe except of the final one, it's hard to provide the robot with clues about what rotation and moving distance are preferable in terms of optimal path search.
4. Even when some path from start to finish is found, we cannot be sure that this path is optimal for the given maze while we have maze cells unvisited by the robot thus making the maze routes space complete.

In order to explore maze route space we can take two approaches: deterministic and random.

Deterministic approach assumes consistent visiting of every available cell. One possible way is as follows:

- we limit robot movement by exactly one cell
- for each cell we memorize its one cell away neighbors to the left, forward and right and put every reachable neighbor to 'unvisited' stack
- move to the left cell, if possible, otherwise move forward, otherwise move right, otherwise move towards first unvisited cell from the stack
- repeat until the 'unvisited' stack is not empty

This algorithm guarantees visiting every maze cell that is reachable from the start, but is slow in terms of time-steps required.

Random exploration assumes more or less random roaming through the maze, maybe with some optimization heuristics, with hope that the robot will be ‘lucky’ in discovering short path to the center without visiting every cell.

Those considerations suggest random exploration during first phase of the competition in order for the robot to learn given maze configuration trying to find out as many available routes to the center as possible. At the same time we have limited time steps budget and penalty for every excess move during exploration phase so the algorithm should take into account exploration-exploitation dilemma while deciding whether to continue exploration or to stop on finding some path that may not be optimal for given configuration.

Proposed solution

My general approach is to build a maze connectivity graph during an exploration phase and on each step of exploration find an optimal solution for travelling from start to goal cells. Since each cell has a unique address that may be represented by an integer (for example from 0 to 255 in 16x16 maze case) we can define graph vertices as cells and define graph edges as routes we can travel from given cell to its neighbor in exactly one time-step.

Initially the connectivity graph has no edges. Each exploration step adds edges to the graph according to the data obtained from robot sensors.

For optimal solution search we can use Dijkstra’s algorithm since it has complexity upper bound O(V-squared), where V is the number of graph vertices. According to problem statement maximum number of vertices is limited to 256 (in case of 16*16 maze grid), so performing Dijkstra optimal path search on each step of exploration phase looks as feasible task in terms of computational efficiency.

In this framework final phase is just an implementation of optimal path found during exploration in case it exists and was found with boundary condition of 1000 time-steps.

Exploration phase may be defined as reinforcement learning task (similar to Train a smartcab to drive exercise in the course) with strategy described by some heuristics dedicated to minimize time required for optimal path search.

Exploration heuristics

- robot recieves award for moves towards the maze center
- robot is penalized for moving to a previously visited cell
- exploration stops immediately if a path with time-length of 10% or less of total number of cells is found
- robot is penalized for vertical moves towards the edges of the maze
- if a move to the center room is possible and the available cell inside the room has not been visited yet, the robot must enter the center room

Exact values for awards and penalties were to be defined during the implementation of the algorithm and described below in the document.

Exploration heuristics explained

By awarding center moves and penalizing moves towards the edges we are trying to prevent the robot from random roaming through the maze and prioratizing softly the movement towards the center.

Penalties for movement into previously visited cells allow to avoid loop movement and multiple dead ends visits (see Figure 3). It’s important to notice that in order to prevent loop movement effectively repeatitive cell visiting penalty should be higher than center move award.

Time-length unconditional stop is an implementation of exploration-exploitation dilemma.

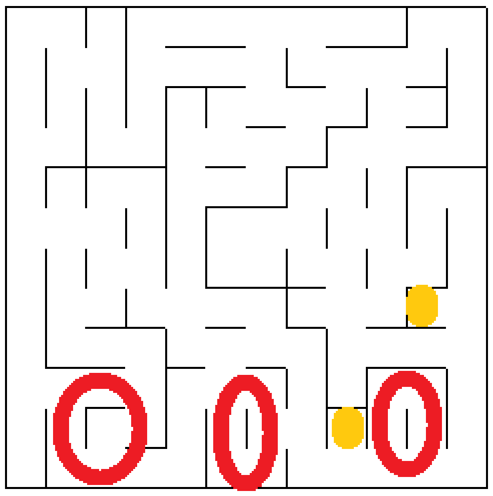

Figure 3: Examples of dead ends (orange) and loop movement (red)

Final heuristic allows to avoid unnecessary randomness in exploring the cells that are central room 'gates'.

Metrics

Model benchmark

Model perfomance on any given maze can be scored as average number of times-steps required to find and follow a path with arbitrary number of attempts (by now 10 attempts seems as enough). We need several attempts for given maze configuration due to probabilistic nature of reinforcement learning algorithm.

This benchmark allows comparison between different models on a given maze configuration. To be more clear we'll explain the benchmark with an example. In a table below each row represents (imaginary) total number of time-steps required for completion both explorational and final phases.

No of test run	Model 1	Model 2
1	100	95
2	100	103
3	100	101
4	100	97
5	100	111
6	100	91
7	100	92
8	100	93
9	100	89
10	100	93
Average time-steps required	**100**	**96.5**

The benchmark for the Model 1 is 100, the benchmark for the Model 2 is 96.5. Since Model 2 has better benchmark than Model 1, one can claim that Model 2 coped better with a given maze configuration.

Evaluation metric

In order to evaluate model perfomance we can also compare the benchmark with lower and upper bounds for optimal path search on a particular maze. It could be useful to judge whether a model needs further optimization.

Lower bound defined as a shortest path length (expressed in the number of time-steps) in a given maze.

Upper bound defined as 1/30 of time-steps needed to build a complete connectivity graph by visiting all the available cells for a given maze plus the lower bound. There is some ambiguity in upper bound definition since for most maze configurations there are several ways to visit every cell. In order to avoid the ambiguity my proposal is to implement an analogue of the depth-first-search in a graph where the robot is allowed to move no farther then exactly one cell in any direction during the exploration and, where possible, prefers left turns to forward moves and forward moves to right turns subsequently.

The difference between upper and lower bounds gives a scale where benchmark described above can be expressed in % thus giving a metric to judge whether the model performance closer to ideal optimal path defined by lower bound or slow maze exploration by visiting every cell, that is upper bound. The example of that metric is as follows:

Let in an arbitrary maze optimal path length consists of 10 time-steps and slow maze exploration with visiting every available cell takes 260 time-steps. In that case:

- Lower bound equals 10.33 (10 time-steps divided by 30 for an exploration phase and 10 time-steps for a final run)
- Upper bound equals 18.67 (260/30 for exploration phase plus 10 time-steps for a final run)

Let the model benchmark evaluated with benchmark algorithm described above equals 111.6. Than the metric will be evaluated as follows:

(111.6 - 10)/30 + 10 = 13.37

percent_metric = ((13.37 - 10.33) / (18.67 - 10.33)) * 100% = 36.5%

The value of a metric can provide some insights on how the model performance is far from an ideal pathfinding behaviour. The ideal value of a metric is 0%.

Note

In case if a robot choose to follow non-optimal path according to learning heuristics the value of a metric could be more than 100%.

Threshold

We can declare any model with metric value higher than 100% as failing to pass the threshold for a given maze configuration. This thresholding approach has easy explanation since it's pointless to have a model that performs worse than an easy algorithm with deterministic behavior and 100% success chance in achieving maze solving goal.

III. Methodology

Project design

A robot carries 'onboard' a maze connectivity graph that represents path structure of a maze. Vertices of the graph correspond to maze cells whereas edges correspond to passages between neighboring cells. Edges have weights corresponding with time to reach appropriate cell from current cell. That allows to take into consideration time robot needs to make a turn left or right. Also, according to suggested project pipeline, the project is an implementation of next_move() function in the *robot.py* file.

Graph initialization procedure

Connectivity graph 'lives' in a separate Graph class. Instance of this class is being created at the beginning of maze exploration phase inside ___init() function of *robot.py.* Since we know maze dimension at this step we can fill graph vertices set and connect four central vertices (target cells) with four edges, since target cells are always a square of four cells without walls between them by problem formulation.

Inner implementation of the connectivity graph is adjucency list since this is more appropriate for Dijkstra algorithm in terms of memory consumption and running time.

Since we have initial maze orientation, we can unequivocally number vertices from 0 by rows from left to right, where vertice No 1 corresponds to starting cell. Also we can always calculate the vertices numbers for target center cells using given maze dimension.

The result of graph initialization procedure is an extremely sparse graph with No of vertices corresponding to maze dimension (for example, 144 vertices for 12x12 maze) and only four edges between central cells.

In this framework the goal of exploration phase is filling connectivity graph with available edges trying to make the passage to central cells as fast as possible.

Robot movement rules

During exploration phase robot movement defined by maze exploration strategy.

During final run robot movement defined by shortest path towards the maze center calculated with Dijkstra algorithm on connectivity graph from current vertice. If no path found in the connectivity graph robot don't move at all.

Maze exploration strategy

The following strategy applied consequently during each step of an exploration phase.

- given the sensors input robot adds edges to connectivity graph from current vertice to neighboring vertices
- robot runs Dijkstra's algorithm on a connectivity graph from the starting point to any of the goal vertices (we can arbitrarily choose any vertice since they are connected during graph initialization). If such path is found and its length is 10% or less of total vertices number (see *exploration heuristics* described above) **next_move()** returns ('Reset', 'Reset') tuple thus haulting exploration phase
- if the path is found but its length does not satisfy the condition above, robot haults further exploration with probability of one half of total steps made by the moment divided by 1000 (total time steps allowed for run). This condition partially prevents the robot from further ineffective maze space exploration when some path, probably not optimal, already found. 'One half' coefficient for this probability may be corrected during my final perfomance tests of the model
- robot maintains a list of vertices, that are not visited yet, but available for the exploration according to sensors input obtained so far
- every maze cell belongs to a virtual layer, where target central cells belong to layer 0, all cells that surround central cells belong to layer 1, and so on up till the outer layer, that contains the starting cell. Total number of layers equals half of a maze dimension. The idea of layers gives a way to stimulate moves that lead to the center of the maze and penalize moves towards outer borders
- final stage of a strategy is a choice of next cell that robot is intended to move into. It has to choose one point from a tuple of at maximum 12 destination points. Every point in a tuple has a score that represents the value of moving into this point. Tuple points scoring requires two steps
- first step. Each available point scored from -3 to 3 based on how many layers this point is closer to the center from the current robot position (we'll call that number 'layer difference'). Than every point marked as 'visited' recieves additional -3 penalty.

- second step. Robot traverses 'not visited' list and calculates shortest path from the current point to every point from this list. Any tuple point lying on that path recieves added score that equals layer difference divided by path length in time-steps.
- final move selection. If a tuple with weighted points obtained by the above procedure contains points with positive weights, robot throws away all points with non-positive weights and chooses randomly next point from remaining positive-weighted with probability normalized by point weight. If there are no positive weights in a tuple, all weights are increased by largest absolute weight and robot makes random selection based on a point probability given by normalized weight.

The goal of a sophisticated next move selection procedure described above is to implement robot's behaviour similar to depth-first-search graph exploration that is focused to reveal available paths leading as straight to the maze center as possible. Also this procedure is a detailed implementation of exploration heuristics described in appropriate section above.

Implementation

Graph class

Graph class is an adjucency list implementation of classical undirected graph that represents connectivity graph 'carried' by a robot. Vertices of the graph are integers corresonding to unique cell address, edges represent neighboring cells that are reachable from the given cell in one time-step.

init method creates a data structure, fills the graph with vertices and adds edges connecting the central room cells.

```
    def __init__(self, maze_dim):
        '''
        Creates vertices and edges sets, keeps basic data structure
        parameters
        '''

        self.vertice_set = dict()
        self.maze_dim = maze_dim
        for i in range(maze_dim * maze_dim):
            self.add_vertice(i)
        #Connect four central vertices with four edges
        vert1 = (self.get_dimension() - 1) * self.get_dimension() / 2 - 1
        vert2 = vert1 + 1
        vert3 = (self.get_dimension() + 1) * self.get_dimension() / 2 - 1
        vert4 = vert3 + 1
        self.add_edge(vert1, vert2)
        self.add_edge(vert1, vert3)
        self.add_edge(vert4, vert2)
        self.add_edge(vert4, vert3)
```

Apart from usual methods providing access to graph data structure elements, Graph class contains Dijkstra algorithm implementation that is core for maze solving task solution.

```
    def search_path(self, source, target):
        """
        Returns shortest path between Source and Target as a list of
        vertices. Uses simplified Dijkstra approach since all edges have
        equal lengths
        """
        if not self.has_vertice(source) or not self.has_vertice(target):
            print 'Incorrect call to search_path() method. One or more vertices not found:', \
                source, target
            return None

        nodes_to_visit = {source}
        visited_nodes = set()
        distance_from_start = {source: 0}
        tentative_parents = {}
        while nodes_to_visit:
            current = min([(distance_from_start[node], node) \
                            for node in nodes_to_visit])[1]
            if current == target:
                break
            nodes_to_visit.discard(current)
            visited_nodes.add(current)
            edges = self.get_neighbors(current)
            unvisited_neighbors = edges.difference(visited_nodes)
            for neighbor in unvisited_neighbors:
                neighbor_distance = distance_from_start[current] + 1
                if neighbor_distance < distance_from_start.get(neighbor, float('inf')):
                    distance_from_start[neighbor] = neighbor_distance
                    tentative_parents[neighbor] = current
                    nodes_to_visit.add(neighbor)
        return self._construct_path(tentative_parents, target)

    def _construct_path(self, tentative_parents, target):
        """
        Builds a list of graph vertices as a result path of Dijkstra's
        algorithm
        """
        if target not in tentative_parents:
            return None
        cursor = target
        path = []
        while cursor:
            path.append(cursor)
            cursor = tentative_parents.get(cursor)
        return list(reversed(path))
```

search_path method returns a list with vertices that is the path from *source* to *target* cells if one exists.

One more nonstandard Graph class method worth mentioning is **get_layer** that returns layer number for any given cell number inside the maze.

```
    def get_layer(self, vertice):
        """
        Returns layer number for a given vertice
        Maximum layer number defined by maze dimension
        For maze with dimension 12 maximum layer number is 5
        Four central cells of a maze belong to layer 0
        Starting cell belongs to layer with maximum number.
        For maze dimension 12:
        starting cell (number 0) belongs to layer 5
        cell with number 33 belongs to layer 3
        cell with number 120 belongs to layer 5
        cell with number 91 belongs to layer 1
        cell with number 25 belongs to layer 4
        cell with number 106 belongs to layer 4
        cell with number 66 belongs to layer 0
        """

        row_number = (vertice / self.get_dimension() - self.get_dimension()/2 \
                      if vertice <= self.get_dimension()**2/2 \
                      else vertice / self.get_dimension() - \
                           self.get_dimension()/2 - 1) + 1
        col_number = vertice % self.get_dimension() - self.get_dimension()/2 + 1 \
                      if vertice % self.get_dimension() < self.get_dimension()/2 \
                      else vertice % self.get_dimension() - self.get_dimension()/2
        return max(abs(row_number), abs(col_number))
```

Robot implementation

Robot behavior defined by **next__move** method in Robot class, that returns next move for virtual robot interpreter.

Capstone project includes two different implementations of exploration strategy:

- **solver** mode (default) that implements random strategy with learning heuristics
- **benchmark** mode that implements deterministic approach with visiting every available maze cell as described in "Analysis" section. This mode used to calculate model metrics. In order to activate 'benchmark' robot behavior Robot class must be called with 'benchmark' parameter as follows: *robot = Robot(testmaze.dim, 'benchmark')*

Robot class variables are universal for both modes (except for 'solver' mode parameters named with capital letters). Their descriprion are as follows:

```
def __init__(self, maze_dim, mode = 'solver'):
    '''
    Use the initialization function to set up attributes that your robot
    will use to learn and navigate the maze. Some initial attributes are
    provided based on common information, including the size of the maze
    the robot is placed in.
    '''

    self.location = np.array([0, 0])
    self.heading = 'up'
    self.maze_dim = maze_dim
    self.connectivity_graph = Graph(maze_dim) #connectivity graph
    self.steps_counter = 0 #time steps counter
    self._orientation = {'up': (1, 0), 'down': (-1, 0), \
                         'left': (0, -1), 'right': (0, 1)}
    self._unvisited = list() #list of a cells to visit during exploration phase
                             #with 'benchmark' mode
    self._orientation_sequence = ('up', 'right', 'down', 'left')
    self.visited = set() #set of visited cells during exploration phase
    self._visits_counter = dict() # dictionary for counting cell visits during exploration
    self._reachable = set() #set of cells that are reachable but still not visited.
                            #Used in 'solver' mode
    self.max_sensor_view_range = 3 #max sensor sight range in cells
    self.max_move_range = 3 #max cells robot can move forward in one time-step
    self.mode = mode  #exploration mode: 'benchmark' for slow full exploration or
                      #'solver' for the fast one
    self.path = list() #real path of a robot during all runs
    self._local_path = list() #helper list for storing paths towards next unvisited cell
    self.run = 1 #Number of run. 1 - exploration phase, 2 - first step of a final phase,
                 #3 - other steps of final phase
    self.stop_run = False # inner flag that represents current run needs to stop
    self._TOTAL_STEPS = 1000 # No of total steps allowed for maze travelling

    # Exploration phase tuning constants used in 'solver' mode
    self._SHORT_PATH_COEFF = 0.10 # used for calculation of immediate stop criteria
                                  #during exploration phase
    self._PROB_REDUCE = 0.8 # probability reduce coefficient for random stop when
                            # probably suboptimal path found
    self._SECOND_VISIT_PENALTY = 100 # penalty coefficient for repeatitive visit
                                     # to the same cell
    self._CENTER_MOVEMENT_BONUS = 1.5 # weight bonus coefficient for movements
                                      # towards the center
```

Four last constatnts are 'solver' mode parameters that can be tuned in order to achieve better model performance.

Benchmark mode implementation

Robot goal in 'benchmark' mode is to produce two metric values for the given maze:

- minimum possible score where robot follows optimal path both in exploration and final phases
- model score for the deterministic exploration mode. Note, that this score is invariant for given maze configuration.

Core 'benchmark' mode method is **benchmark** together with helper method **next_cell_benchmark** that returns rotation and movement for the robot according to deterministic step-by-step exploration strategy. **benchmark** just puts unvisited cells in stack and subsequently calculates the path to the next cell from it until the stack is empty.

Other helper methods used in **benchmark** method with code omitted in this document are:

- *update_graph* updates connectivity graph given sensors info
- *update_position* returns new robot position and heading given robot rotation and movement
- *get_shortest_path* returns shortest path from current to target cell using Dijkstra's algorithm defined in Graph class
- *target_coord* returns cell values for four center cells of the maze
- *convert_to_moves* converts list of cell numbers to list of (Rotation, Movement) tuples
- *truncate_path* Shortens path for 'benchmark' mode by replacing one-cell-forward moves with two or three cells forward where possible

```
    def _benchmark(self, sensors):
        """
        Returns next move in a benchmark mode
        """

        if self.run > 1:
            if self.cell() in self._target_coord():
                return 'Reset', 'Reset'
            return self._next_cell_benchmark()
        self._update_graph(sensors)
        while (len(self._unvisited) > 0) and (self._unvisited[-1] in self.visited):
            self._unvisited.pop()
        if len(self.visited) == self.maze_dim ** 2 or len(self._unvisited) == 0:
            rotation = 'Reset'
            movement = 'Reset'
            self.run = 2
        else:
            rotation, movement = self._next_cell_benchmark()
            self.location, self.heading = self._update_position(rotation, movement)
        return rotation, movement

    def _next_cell_benchmark(self):
        """
        Returns rotation and movement for the robot to get to the next cell in
        the exploration phase for the 'benchmark' mode and for final phase
        for any modes
        """

        if len(self._local_path) > 0:
            step = self._local_path.pop()
            return step[0], step[1]
        else:
            if self.run == 1:
                target = self._unvisited.pop()
            else:
                target = self._target_coord()[0]
            local_path = self._get_shortest_path(self.cell(), target)
            if local_path[0] == self.cell():
                local_path = local_path[1:]
            local_path = self._convert_to_moves(local_path)
            if self.run > 1:
                local_path = self._truncate_path(local_path)
            self._local_path = local_path[:-1]
            return local_path[-1][0], local_path[-1][1]
```

Solver mode implementation

Solver is a default mode for the robot. Robot's goal in 'solver' mode is as follows:

- during exploration phase: produce next rotation and movement values for the given robot position according to exploration heuristics described above
- during final phase: produce next rotation and movement values for the given position in order to follow an optimal path discovered during exploration.

Core solver mode method is **strategy** together with its helper **next__cell__solver** that implements exploration heuristics. **next__cell__solver** maintains a list of all possible moves from the current cell and selects a move from this list randomly with respect to the weight that is calculated according to learning heuristics and Robot class 'solver' mode parameters.

```
    def _strategy(self, sensors):
        """
        Returns next move in a solver strategy mode
        """
        if self.run > 1:
            if self.stop_run:
                self.stop_run = False
                return 'Reset', 'Reset'
            path = self._get_shortest_path(self.cell(), self._target_coord()[0])
            next_cell = path[1] if self.cell() != 0 else path[0]
            rotation, movement = self._find_move_solver(next_cell)
            self.location, self.heading = self._update_position(rotation, movement)
            return rotation, movement
        self._update_graph(sensors)
        rotation, movement = self._next_cell_solver()
        self.location, self.heading = self._update_position(rotation, movement)
        self.stop_run = self._stop_condition()
        if self.run == 1 and self.stop_run:
            self.run = 2
        return rotation, movement

    def _next_cell_solver(self):
        """
        Returns rotation and movement for the robot to get to the next cell in
        the exploration phase for the 'solver' mode
        """

        neighbors_set = set(self.connectivity_graph.get_neighbors(self.cell()))
        possible_moves = list(neighbors_set)
        moves = np.array(possible_moves)
        weights = list()
        for cell in possible_moves:
            layer_difference = self.connectivity_graph.get_layer(self.cell()) - \
                               self.connectivity_graph.get_layer(cell)
            layer_difference *= self._CENTER_MOVEMENT_BONUS #enhance center targeting
            weights.append(layer_difference + 0.0001)
        for i in range(len(weights)):
            if possible_moves[i] in self.visited:
                weights[i] -= self._SECOND_VISIT_PENALTY * \
                               self._visits_counter[possible_moves[i]]
            if (possible_moves[i] in self._target_coord()) and \
                    (not possible_moves[i] in self.visited):
                weights[i] = sys.maxint # must visit central cell if it is possible
                                        # and it has not been visited yet
        #update Unvisited nodes set
        self._reachable = self._reachable.union(neighbors_set.difference(self.visited))
        for target_cell in self._reachable:
            for i in range(len(weights)):
                path = self.connectivity_graph.search_path(target_cell, possible_moves[i])
                if path != None and possible_moves[i] in path:
                    layer_target = self.connectivity_graph.get_layer(target_cell)
                    layer_source = self.connectivity_graph.get_layer(self.cell())
                    weights[i] += 1.0 * (layer_source - layer_target) / max(len(path), 1)
        weights = np.array(weights)
        if np.max(weights) <= 0:
            weights += abs(min(weights))
        else:
            pos = np.argwhere(weights <= 0)
            moves = np.delete(moves, pos)
            weights = weights[weights > 0]
        weights /= np.sum(weights)
        target_cell = np.random.choice(moves, 1, p=weights)
        return self._find_move_solver(target_cell)
```

Refinement

Model perfomance with different parameters for 'solver' mode discussed in Results section of this report.

Apart from debugging and code refactoring issues one could think about additional heusistics that can potentially improve model perfomance and have not been implemented yet in the current version of the model. For example one can consider changing model's behaviour after the path to the center has been found. Current model version is stimulated softly to explore the regions near the central cells (due to _CENTRE_MOVEMENT_BONUS that is assumed to be greater than 1.0). But we can add additional heuristic that turns on after the central cells are visited and awards the robot for moves towards the starting point. Though this technique requires dynamic change of maze layers numbering during exploration thus making the code more sophisticated, it may potentially lower the number of steps required to produce optimal or sub-optimal path.

IV. Results

Declared Capstone project scope was to implement the algorithm described above and tune the exact values of model coefficients in order to get satisfactory benchmarks on different test mazes. We'll describe the process of model parameters tuning by selecting some models that suit better to some predefined intuitive behavior and comparing them using benchmark described above.

Is it better to stop exploration as early as possible, or to continue exploration after one path is found?

We'll begin test phase description by comparing two different models. One that encourage random roaming after central room is reached and the other that tends to stop as soon as any path is found.

Model coefficients	Model 1	Model 2
Immediate stop path length in % (1)	10%	40%
Chance to stop when probably suboptimal path found (2)	0.1	0.5
Cell revisit penalty	3	10
Layer change towards the center bonus	1.2	5

(1) This coefficient compares the length of a shortest path found so far and, if this path is less than given No in % compared to total number of cells in the maze stops further exploration immediately
(2) This coefficient (denoted in the formula below as coeff2) defines the chance to stop exploration after the probably suboptimal path is found. The exact probability of exploration stop are calculated by the following formula:

p(Stop) = coeff2 * (No of moves done so far) / 1000,

where 1000 is a maximum number of robot moves during the session.

As one can see from the proposed coefficient values Model1 tends to rather roam randomly through the maze showing weak preference to moves towards central layers, whereas Model 2 tends to rupture to the center and stop disregarding the length of a path found.

Test results for 10 runs are in the table below.

Scores	Model 1, maze1	Model 2, maze 1	Model 1, maze 2	Model 2, maze 2	Model 1, maze 3	Model 2, maze 3
Run 1	27,2	22,333	29,367	30,733	34,9	32,233
Run 2	23,233	23,867	36,967	34,667	36,933	34,1
Run 3	25,467	24,533	32,5	28,767	34,167	33,1
Run 4	24,633	25,233	33,5	33,867	33,733	32,9
Run 5	26,8	30,633	30,633	28,433	32,6	33,933
Run 6	25,3	23,433	34,8	25,733	35,767	40,667
Run 7	24,4	20,233	37,4	26,467	38,867	40,833
Run 8	23,6	22,833	35,687	30,967	31,3	32,867
Run 9	28,367	25	42,433	30,433	34,067	33,533
Run 10	26,7	25,1	33,233	29,167	30,933	31,567
Average	**25,57**	**24,3198**	**34,652**	**29,9234**	**34,3267**	**34,5733**

Now let express obtained average results for both models in terms of benchmark metric described in Analysis section. First let calculate benchmark metrics for all three mazes.

Benchmark scores	Maze 1	Maze 2	Maze 3
Best possible score	17,567	26,867	29,967
Slow deterministic exploration score	24,967	36,2	42,633

Now let express average results in a scale given by benchmark metrics for each maze. Best possible score in this metric is 0%. The higher the score, the worse is the result.

Model metrics	Maze 1	Maze 2	Maze 3
Model 1 average	108%	83%	34%
Model 2 average	91%	33%	36%

Model 2 outperforms Model 1 on Mazes 1 and 2, while both models perform equally on Maze 3 that is largest. Also we may note that both models perform rather badly on a relatively small Maze 1.

So, early stop seems to be better strategy than long maze exploration.

Can we do better?

Let improve a bit model 2 by further tweaking model parameters. From the experiment results table we can see that Runs No 5 and 6 on maze 3 gave the relatively worst scores. The cause is that the robot stopped exploration being satisfied with long suboptimal path. So let significantly decrease immediate stop path length coefficient to, say, 10%. Also let decrease suboptimal path stop probability and increase cell revisit penalty. New model 3 parameters are in a table below:

Model coefficients	Model 2	Model 3
Immediate stop path length in %	40%	10%
Chance to stop when probably suboptimal path found	0.5	0.12
Cell revisit penalty	10	1000
Layer change towards the center bonus	5	5

And here are the test results for comparison between Model 2 and Model 3. Test results for 10 runs are in the table below.

Scores	Model 3, maze1	Model 2, maze 1	Model 3, maze 2	Model 2, maze 2	Model 3, maze 3	Model 2, maze 3
Run 1	26,067	22,333	30,3	30,733	30,7	32,233
Run 2	27,033	23,867	27,8	34,667	31,933	34,1
Run 3	22,533	24,533	32,833	28,767	37,433	33,1
Run 4	20,667	25,233	29,2	33,867	33,367	32,9
Run 5	21,967	30,633	28,567	28,433	29,033	33,933
Run 6	21,8	23,433	31,633	25,733	31,967	40,667
Run 7	26,167	20,233	34,4	26,467	32,433	40,833
Run 8	26,267	22,833	33	30,967	34,467	32,867
Run 9	22,3	25	27,067	30,433	32,533	33,533
Run 10	21,6	25,1	37,8	29,167	33,233	31,567
Average	**23,6401**	**24,3198**	**31,26**	**29,9234**	**32,7099**	**34,5733**

Benchmark metrics comparison between Model 3 and Model 2:

Model metrics	Maze 1	Maze 2	Maze 3
Model 3 average	82%	47%	22%
Model 2 average	91%	33%	36%

First of all, both Model 3 and Model 2 satisfy threshold condition, defined in Benchmark&Metrics section of the report. Metrics score is lower than 100% for all runs.

Since Model 3 performs better on two of three mazes and its performance is significantly better than slow deterministic approach used for mazes tested we can state that we found satisfactory set of parameters for maze exploration in a given project pipeline.

Final model parameters set used for submission:

Model coefficients	Model 3
Immediate stop path length in %	10%
Chance to stop when probably suboptimal path found	0.12
Cell revisit penalty	1000
Layer change towards the center bonus	5

Solution robustness

First of all, due to probabilistic nature of the algorithm used, there is always a non-zero chance for a robot to fail exploration phase on a maze that has a solution. Exact calculation of failure probability even on a given maze is a complicated task itself and deserves to be a subject of separate research.

From the other hand there are some considerations supporting the claim that failure probability is in reality insignificant.

1. Proposed solution has no explicit preferences towards the shape and length of any solution path, and, in particular, three test mazes provided by Udacity. It means that one can consider three Udacity mazes as random configurations given as input to the algorithm. During parameter selection phase the algorithm was run on those mazes more than a hundred times and the exploration phase never took more than 400 time-steps that is two and a half times less than 1000 time-steps threshold.

2. We may test a model on a maze with special 'hard' solution path in order to get more confidence in algorithm robustness.

Maze on Figure 4 (courtesy to Naoki Shibuya, former Udacity student, link from sample Capstone provided in project briefing) was specially constructed to contain extreme quantity of loops and dead-ends together with long solution path, so it can be considered as complicated maze, that has high level of failure risk during exploration phase.

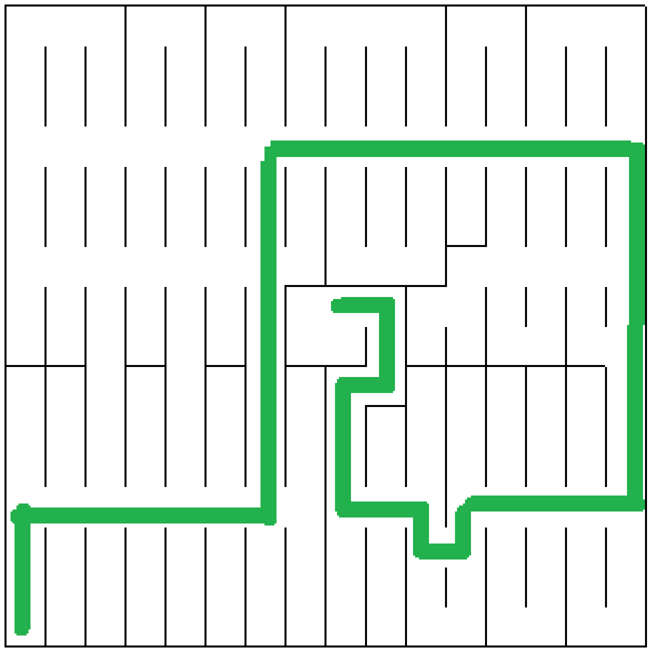

Figure 4: Example of maze with 'hard' solution

Table below contains results of five test runs on this complicated maze.

No of test run	Exploration path length in time-steps
1	261
2	603
3	815
4	716
5	695

We can see that the robot coped well with all test runs though having significantly longer exploration phase than usual. It adds a bit of confidence that the algorithm will converge to some solution in overwhelming majority of real life cases.

V. Conclusion

Visualization of robot paths dusring exploration phase

Because of the probabilistic nature of the proposed solution there is no much sense in drawing robot's exploration path during one particular run in a selected maze.

In order to visualize preferred robot paths on a particular maze we can build a heatmap representing number of visits to each particular cell during exploration phase for several test runs. Figure 5 shows cell visits frequency for 5 test runs on 12x12 Udacity test maze (maze No 1). The intensity of the blue color on a heatmap represents frequency of cell visits.

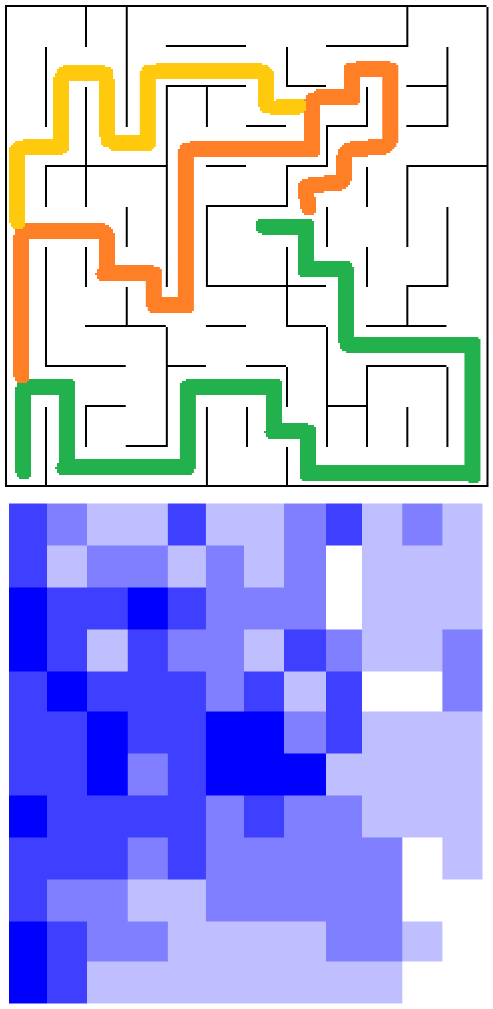

Figure 5: Cell visits during exploration phase heatmap. Cumulative No of visits on 5 test runs

We can see from the figure that due to model parameters robot tends to explore more thoroughly shortcut paths leading straight to the maze center thus preferring to choose more lengthy suboptimal paths shown in orange and yellow. This is the visual explanation of comparatevely inefficient model score on Maze 1 that was the subject of parameter selection discussion above. We may try further parameter tuning by reducing award for moves towards the center thus stimulating the model to find optimal path in that maze but it will likely come with the price of worsening model perfomance on mazes 2 and 3. So let leave this fact as an illustration that final parameter selection is always a subject of compromise.

Personal reflection

My personal reflection from this project was the possibility itself to come up with implementation of an algorithm that is at the same time highly probabilistic and definitely outperform traditional deterministic approach that seem natural for this kind of tasks. Also the fact that rather small selection of simple and intuitive heuristics can significantly influence model performance was a pleasant bonus.

My personal project walkthrough can be divided into three main parts:

- the idea to reflect possible moves space into graph domain and implementation of the graph data structure with Dijkstra algorithm for shortest path search
- implementation of the ‘benchmark’ based on a deterministic approach that will be used later for model metrics
- implementation of probabilistic reinforcement learning algorithm with optimization heuristics

Most interesting part of the project is actually hidden behind the curtains almost completely, since it was rather painstaking parameter tuning in order to provide most obvious and universal way of finding proper model parameters set.

VI. Final notes

This section dedicated to possible development of the model. By now I see following general ways for further improvements.

- refactor maze cell addressing to (row, column) tuples from current addressing as single integer
- learning algorithm improvement to learn more effectively after some path is found
- using neural net for learning optimal values for model parameters
- taking into account less abstract physical details such as more accurate timing and robot movement imperfection

The choice to address maze cells as integers were made to simplify debugging (since debug prints of graph shortest paths look much shorter) but it came with a price of more sophisticated graph layer calculations and several helper functions to decode cell addresses from tuples to numbers and visa versa. Simple changes in graph class in order to convert vertices to (row, column) tuples would make those helper functions obsolete, _get_layer() method simpler and would improve model perfomance a bit.

The idea of changing learning priorities after some path to the center is found is pretty straightforward. It looks worth to award the robot to take moves towards the start after it made its way to the center rather than prioratize further roaming around the center cells. This change requires dynamic switch in layer definition during execution with changing starting point for layer calculation from center to starting cells.

The task of optimal model parameters selection is actually an optimization task in a multidimensional space of possible robot moves inside a given maze. My personal selection of ‘optimal’ parameters was rather intuitive, based on a simple implicit assumptions and certainly did not take into account complicated shape of this space. A neural net, even trained only on three available mazes provided, could produce better sets of model parameters.

Accurate accounting for possible robot movement imperfections, sensors errors and more accurate timing (since it obviously takes more time for a robot to make a turn and then drive three cells forward than just move one cell ahead) creates really vast area for further model development.

References used in a document

- Micromouse competition description
- Udacity project pipeline
- Dijkstra’s algorithm (with running time estimate)
- Maze Solving Algorithms for Micro Mouse
- An Efficient Algorithm for Robot Maze-Solving
- A New Shortest First Path Finding Algorithm for a Maze Solving Robot
- Naoki Shibuya Plot and Navigate a Virtual maze capstone, provided by Udacity as sample project