Machine Learning Engineer Nanodegree - Udacity

Plot and Navigate a Virtual Maze

John Kinstler
07/26/17

Table of Contents

I. Definition

Project Overview

The goal of robot motion planning is to find a path for a robot to follow from a starting point to some distant waypoint without colliding with obstacles in the world (also called the configuration space) the robot is moving through. In this particular iteration I will be creating and testing algorithms to allow a robot mouse to plot and navigate a virtual maze.

Motion planning has applications in autonomous and automation robotics, in video game artificial intelligence, architectural design, medicine and biology. It is further used in the nascent self-driving car industry.

My own particular interest in this area arose from implementing reinforcement learning to teach a Smart Cab to learn the rules of the road for a previous project. Here, rather, different algorithms will be explored as a potential solution.

Problem Statement

This project takes inspiration from Micromouse competitions, wherein a robot mouse is tasked with plotting a path from the lower left corner of the maze to a 4 square center where only one of the squares has an opening. The robot mouse may make 2 runs in a given maze. In the first run, the robot mouse tries to map out the maze to figure out the path to the center. In the second run, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned.

In this project, I will create functions to control a virtual robot to navigate a virtual maze. A simplified model of the world is provided along with specifications for the maze and robot; the goal is to obtain the fastest times possible in a series of test mazes.

There are a variety of motion planning algorithms which are detailed here. Some further discussion on this topic can be found here, wherein the author describes 3 algorithms most commonly used to solve motion planning problems which are A* grid search, Visibility Graphs, and Flow Fields. Alternatively, I could explore more typical machine learning algorithms such as Neural Networks or Reinforcement Learning.

This project will likely be best accomplished using the A* algorithm for which implementation guidance can be found here.

I have also found another algorithm called D*-Lite search which takes a bit of different approach to A*. The implementation I used for this D*-Lite was borrowed from Sebastian Thrun's AI-Robotics class on Udacity.

Metrics

The Scoring Specification is given in the MLND Capstone Project Description - Robot Motion Planning:

“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. Its objective now is to go from the start position to the goal room in the fastest time possible. 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.”

For this competition, the robot is allowed to advance 3 cells in one movement either forward or backward with no rotations or to advance one cell after a rotation. This means speed will be enhanced not only by the shortest route but also by maximizing the sequences of non-turning cells in the path to the goal. With this in mind, we'll want to minimize the total number of movements to the goal. This will be the key metric for evaluating which path is the best path. As it turns out, the best path is not always the shortest in total length or even the least movements.

II. Analysis

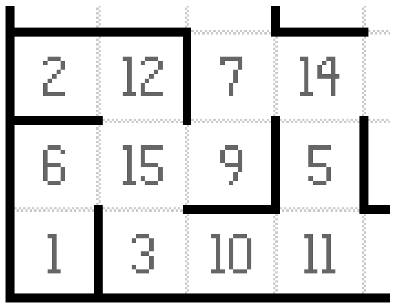
Data Exploration

The central data/input for this problem is the maze via csv text file. Additionally, the robot itself can sense the configuration space via 3 sensors mounted front, left, and right. The actual inputs will be whether the robot sees walls or open spaces to the left, center, and right.

The Maze Specification is given in the: MLND Capstone Project Description - Robot Motion Planning

"Maze Specifications

The maze exists on an n x n grid of squares, n 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.



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."

The Robot Specification is given in the MLND Capstone Project Description - Robot Motion Planning:

"Robot Specifications
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.

More technically, 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."

Exploratory Visualization

The first thing to visualize are the mazes themselves. For that there are several python programs that have been created to run the algorithms and help with the visualization:

- robot.py - This script establishes the robot class. This is the only script that will be used to implement the algorithms.
- maze.py - This script contains functions for constructing the maze and for checking for walls upon robot movement or sensing.
- tester.py - This script will be run to test the robot's ability to navigate mazes.
- showmaze.py - This script can be used to create a visual demonstration of what a maze looks like.
- test_maze_##.txt - These files provide three sample mazes upon which to test the robot.

Maze 1 (12 x 12):

I used the A* algorithm to create the solve_maze.py program to find the fastest route to the goal for any given input maze file in the aforementioned format.

Figure 1 shows the fastest solution for test_maze_01.txt. This is useful as a benchmarking tool to determine how well the robot is performing. When I independently count the moves for alternative routes, the next best yields 24 moves to the goal.

This route is 31 total grid spaces long, requiring 23 moves, and 17 rotations. The next best route is 31 spaces, 24 moves and 15 rotations.

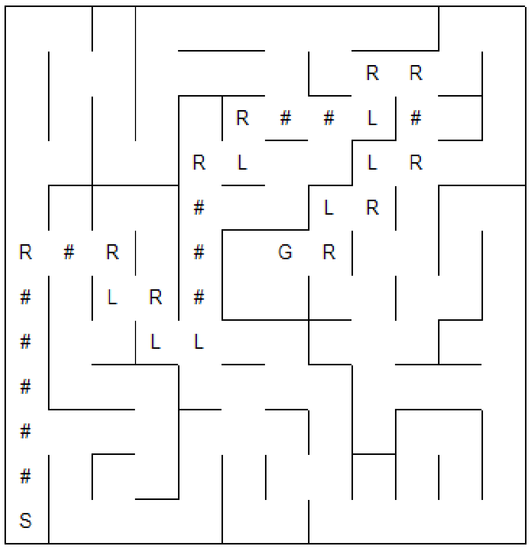


Figure 1: test_maze_01.txt - fastest route – 23 moves

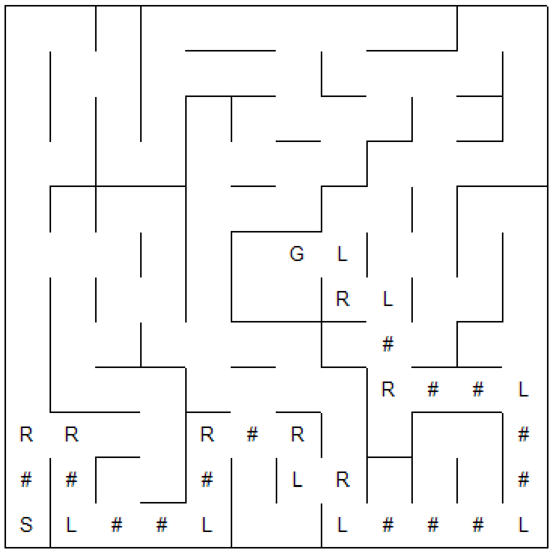


Figure 2: test_maze_01.txt - second best - 24 moves

Figure 2 shows the 2nd best route. This is the best the solver can do when the rotation-cost is larger than the no-rotation-cost. So imposing a rotation-cost will produce routes with fewer rotations, but not necessarily fewer moves.

Maze 2 (14 x 14):

For this second maze, you can see from Figure 3 that the red route is the shortest/fastest with 30 moves to the goal. However, Figure 4 shows the best route that the solver can find. This second best route was found with the COVERAGE_RESET_THRESHOLD set to 'goal' so that it resets the first run upon finding the goal. Although the red route is the best route, the robot can't find until it maps 94% of the maze. The score it receives for finding the 32 move route when it resets the first run on finding the goal is 35.533 and the score it receives for finding the 30 move route when resetting after 94% is 36.1. So, we can see that finding the best route is not as good as finding a good enough route faster.

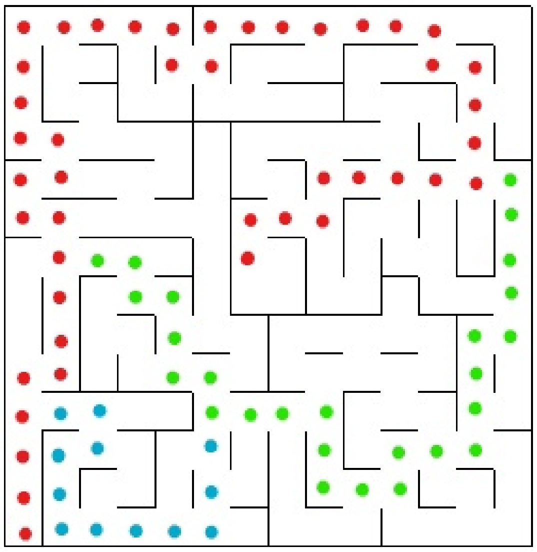


Figure 3: Red Route = 30; Blue Route = 32; GreenRoute = 32

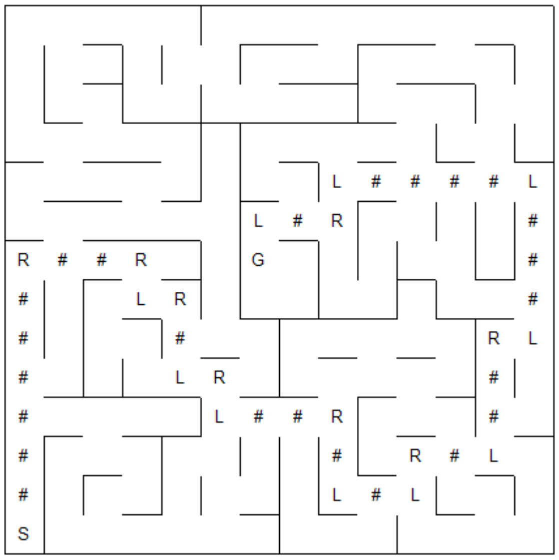


Figure 4: test_maze_02.txt - best route - 32 moves

The best route is 44 long, 30 moves, 20 rotations. The second is 44 long, 32 moves, 18 rotations.

Since this competition scores based on fewest moves as equivalent to least time, the robot will suffer increased time score due to this longer path.

Maze 3 (16 x 16):

For the third maze (following page), you can see that the solver is able to find the best route with 50 spaces, 33 moves and 18 rotations.

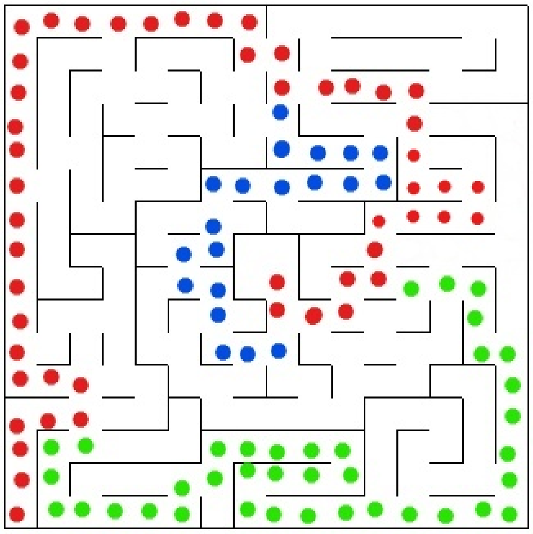


Figure 5: Red Route = 33; Blue Route = 35; Green Route = 34

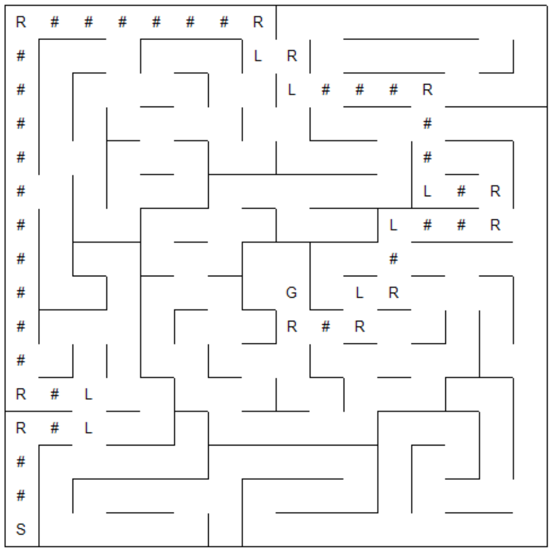


Figure 6: test_maze_03.txt - best route - 33 moves

Algorithms and Techniques

The core functionality for this project will start with the Robot class next_move() method in robot.py. This function will read in the sensor data, keep track of the time steps, determine which run the robot is performing and then activate either the explore() method or the exploit() method. It will then return the rotation and movement values back to tester.py

The explore() method updates the maze map and tracks which cells have been visited. It then determines if the goal has been found and then how much more of the maze is to be mapped: it uses the environment variable COVERAGE_RESET_THRESHOLD as a percentage of how many total possible movement directions for each of the grid cells have been mapped (4 per cell). When ready it sends the reset signal to the tester to begin the second run. Until then it activates whichever exploration controller method is to be used, receives rotation/movement values, updates the robot's position with these new values, then returns rotation/movement to the next_move method.

The exploration controller can be set with the environment variable MAP_STRATEGY which can have the values 'random', 'avoid dead ends', 'smart map 1', 'smart map 2'. As well, environment variable MAPPER can be set to 'mapper1' or 'mapper2' to test how well the different maze mapping methods work along with the different explorer methods.

The exploit() method initially runs the path finding algorithm, A* (I've also implemented D*-Lite as a check on A*). It applies rotate_cost and no_rotate_cost parameters when generating the cost_so_far map, which it then uses to decide which cell to move to to find minimum cost increases from cell to cell as robot moves from start to goal.

The a_star_search1() method takes the start and goal_door_location as inputs. It then:

1. Initiates a priorty queue, open, with the start cell (0,0) as the first entry with a priority of 0.
2. Initiates a dictionary, came_from, with the start cell (0,0) and a value of None (indicating that the start cell is the beginning). Every key in came_from is any cell in the maze_map. It's value is a neighboring cell that is chosen by minimizing the cost that combines the number of steps from the start cell plus the Manhattan distance to the goal cell.
3. Initiates a dictionary, cost_so_far, with the start cell (0,0) and a value of 0. Every key in cost_so_far is any cell in maze_map that has as it's value the move_cost which is the sum of the number of steps from the start cell where movements that require no turning from one cell to the next are given a value of no_rotate_cost and movements that require a left or right turn are given a value of rotate_cost.
4. initiates a loop that continues looping until either the goal has been found (success) or the open priority queue has been exhausted (failure to find a route).
5. It then removes the first (lowest value) entry in open and makes it the current cell for searching its neighbors.
6. If any neighbor to the current cell is in the maze_map and is not separated by a maze

wall, then it calculates the new move_cost from the current cell to that neighbor.

7. If that move_cost is less than the current cost for that cell, or if there is no cost entry for that neighboring cell yet, then this new lowest move cost is added as cost_so_far[neighbor cell] = move_cost.
8. The priority value for this neighbor cell is calculated as the sum of the this move_cost for this neighbor cell plus the Manhattan distance from this cell to the goal cell.
9. Then this neighbor cell is pushed to the open priority queue with current value of priority. The open priority queue automatically sorts all of its items according to their priority values from lowest to highest so the lowest value represents the cell with minimum cost.
10. The current cell (for which these neighbors are being checked) is then added to the came_from dictionary for the neighbor cell: came_from[neighbor cell] = current.
11. Steps 6,7,8 are repeated for every neighbor to the current cell.
12. This iterative loop ensures that every entry in came_from[next_cell] = current cell always contains the cell with the minimum cost of moving to the next_cell. This is how we find the straightest path with the least moves from the start to the goal.
13. Once the goal has been found through this continuous and iterative looping, the came_from dictionary can then be used to construct a list of these cells that form the shortest, straightest path from start to goal.
14. This shortest, straightest path is what a_star_search1() returns to exploit().

When exploit() receives the new goal_route path from a_star_search1() it then calls route_plan, which indexes through the new goal_route list to return rotation/movement values which are then returned to the next_move method.

Finally, it prints the map of the goal route when the goal is reached and will print out any reporting maps and values if TEST_REPORT is set to True.

All of the code necessary to run tester.py with robot.py is in the RobotMotionPlanning.ipynb notebook.

I have also set up several parameters to adjust for observing various performance characteristics.

rotate_cost and no_rotate_cost establish costs for moving from one cell to the next as the robot proceeds from start to goal. Keeping the rotate_cost higher than the no_rotate_cost creates higher costs for turning left or right as a means to keep the final paths as straight as possible to take advantage of movement increases of up to 3 consecutive cells during the second run.

COVERAGE_RESET_THRESHOLD will reset the first run after the specified percentage of the maze is mapped. The default is set to reset as soon as the robot finds the goal.

MAP_STRATEGY will select 1of 3 maze mapping strategies (default is 'smart map 2'):

- random: whenever robot has to choose between multiple cells to move to, it selects one at random
- avoid dead ends: strategy for determining that robot is in a dead end cell or corridor and marking them so robot does not re-enter them. Also prevents robot from re-enter the goal room or the start corridor.
- smart map 1: in addition to the avoid dead ends functionality, this controller uses a tiered selection strategy

  1. select the cell that has been mapped the least
  2. if all options equally mapped, chose the one least visited
  3. if all options equidistant from goal, chose the one closest to the goal
  4. if all options equally visited, chose one at random.
- smart map 2: updates the random choice in step 4 with a more structured method based on which quadrant the robot is in and whether it has found the goal

  5. if all options equally visited, chose the one in the direction based on the following strategy
     - if goal has not been found, select cell to move to based on the following direction precedence:

       i. first quadrant (lower left): 'r','d','u','l'

       ii. second quadrant (lower right): 'r','u','d','l'

       iii. third quadrant (upper right): 'd','l','r','u'

       iv. fourth quadrant (upper left): 'r','d','u','l'
     - if goal has been found:

       i. first quadrant (lower left): 'r','u','d','l'

       ii. second quadrant (lower right): 'r','d','u','l'

       iii. third quadrant (upper right): 'u','l','r','d'

       iv. fourth quadrant (upper left): 'l','u','d','r'

TEST_REPORT allows you to view various performance metrics as well as several maps: the final route selected, the movement costs for each cell calculates by A*, the number of times each cell was visited, and the number of walls mapped for each cell in the maze (map coverage). All of this was invaluable in helping to debug the code and analyze performance.

Benchmark

The robot needs to map the maze as quickly as possible and find the goal as quickly as possible to minimize the first run contribution to the final score – finding the goal is the key to ending the first run and resetting to begin the second run. Since more movements the exploring phase expends on finding the goal will result in longer run-times, we'll want to get to the goal in as few moves as possible while maximizing how much of the maze gets mapped along the way.

Since the second run contributes every move the robot makes to the final score, this will have the largest impact on the score so we'll want the robot to find the shortest route it can given the percentage of the maze it mapped during the first run.

To evaluate how well the robot performs, we'll need to know what the shortest/fastest possible route to the goal actually is to determine what the lower limit on the score can really be. For this I'll create a separate solver program that will simply open the test maze file, map it completely using my implementation and run it through the A* algorithm for the sole purpose of determining what the fastest route is for a fully mapped maze as a comparison to the actual route the robot finds.

I'll want to evaluate how well the algorithm itself does by comparing it to the shortest route that I, myself, can find. I've presented manual solutions above just as a first check on the algorithm's performance.

For this best score benchmark I'm using the routes that I mapped out by hand to evaluate how well the search algorithm worked.

Based on this, it can be determined what the shortest possible path is to the goal. However, this path is further truncated to the number of moves required to traverse the path that take advantage of movement increases bases on no rotations for up to 3 consecutive cells.

For test_maze_01, the best path requires 23 moves to the goal.

For test_maze_02, the best path requires 30 moves to the goal.

For test_maze_03, the best path requires 33 moves to the goal.

The final score will add 1/30th of the number of time steps the robot required to explore the maze during the first run. Hypothetically, the robot might have been able to follow the exact route to the goal without any deviations, thereby requiring the same number of time steps to explore the maze as the best path for each maze requires.

However, I've set up the exploration runs here to only make single cell moves, so that the robot can make maximum map coverage by taking advantage of the sensor values for every single cell it visits without missing anything. Therefore, instead of using the minimum number of moves to the goal, I'll use the minimum total path length to the goal for this best possible score calculation.

So then the best possible scores for each maze should be:

best score = least moves path + (total route length for least moves path)/30

Realistically, though, these scores are impossible to achieve. But they give us a sense of the absolute lower limit score any robot could ever possibly to achieve.

In addition to this though, we need to know what the worst possible score would be for each maze. Here, I'll imagine that the first plus second runs require the maximum 1000 time steps to complete both runs.

Then, for the first run it takes the robot 1000 time steps minus the number of moves required for the worst possible path to find the goal room during the second run, where the robot can take advantage of three step moves when able.

Obviously, exceeding the 1000 time step limit would be worse, but since there's no way to assign a score to that, I'll disregard it.

This worst path will be the one that requires the most number of moves to reach the goal during the first run, which is discovered through trial and error running the robot with the random_explore() and update_map1() methods.

So then the worst possible scores for each maze should be:

worst score = most moves path + (1000 – most moves path)/30

Since increases in metrics are intuitively more satisfying and easier to understand (for me anyway), I'll use as a performance score the percentage of the difference between the robot's score and the worst possible score for the total score range for that maze.

However, it must said, that this score, too, is likely impossible in practice. It turns out, to find the longest paths in the maze, the robot needs to find the goal in the first run faster than normal. This means that the robot maps typically less than 50% of the maze, which is why the exploit() finds such long twisty paths that require so many movements.

If the robot had taken over 900 time steps in the first run, it likely would have mapped nearly or completely 100% of the maze and thus exploit() would have found the best possible path.

Thus, it may seem that a more legitimate version of the worst score would that robot takes as long as possible for the first run while leaving just enough time to solve the second run

However, it turns out that using what should be the worst performance mode ('random', 'mapper 1', reset on 'goal') to generate the worst scores, the robot was still able to return worse scores than this hypothetical worst score metric. It turns out that it doesn't get much worse than the hypothetical and never gets as good as the hypothetical best.

Since these are actual scores, though, I will use them instead.

So, then I'll compute the performance score for each maze as follows:

performance = (worst score – score)/(worst score – best score)

Therefore, the lower the time score the robot receives, the higher the performance score, which will indicate better performance the closer it is to 1.

III. Methodology

Data Preprocessing

No preprocessing is necessary for this project as the data is the test mazes themselves that are handled by the Maze class python script.

Implementation

Within the *tester.py* script is the following variable:

```
sensing = [testmaze.dist_to_wall(robot_pos['location'], heading)for heading in
dir_sensors[robot_pos['heading']]]
```

which assigns the robot's sensor data to the variable *sensing.* This is what is passed to the robot constructor that my algorithms will use to determine the next move.

In order to populate this variable with values, I need to execute

```
run tester.py test_maze_01.txt
```

in the same Jupyter notebook. Upon doing that the code runs and returns the following:

```
Starting run 0.
Allotted time exceeded.
Starting run 1.
Allotted time exceeded.
```

Which is the result of not having any algorithms implemented yet. However, I can now see what information the variable *sensing* actually captured:

```
sensing
[0, 11, 0]
```

So starting from the the lower left corner of this maze (0,0) we see that the left sensor returns 0 because of the left wall, the front sensor return 11 for the 11 open cells in front of it, and the right sensor returns 0 because of the right wall.

I have set up all of the necessary functions for the robot to execute it's next_move method within the Robot class.

There are several printing methods for displaying the various python dictionaries:

maze_map, cost_so_far, count_map, rmap (route map).

is_mapped() determines how much of the maze is mapped and whether the robot is done mapping.

goal_door() determines whether the goal door has been found and identifies it.

update_position() maintains the location and heading of the robot for every time step.

update_map2() builds the maze_map dictionary and allows robot to take advantage of the sensor readings to update wall locations for as many cells as possible without actually visiting the cell. For example, if robot senses a wall 4 cells to the right, the update_map2 function will add the distance to that wall for each successive cell in that direction. This allows the robot to map more of the maze than it actually visits while it searches for the goal. This has proven to be a very effective strategy.

update_count_map() tracks how many times each cell was visited by building the count_map dictionary.

dead_ends() determines whether the robot is in a dead cell or corridor. It returns a list of sensor indices of which cells don't lead to a dead end as well as the list of these cells.

map_strategy() lays out the list of direction precedence describe above.

nearest_unmapped() finds the nearest unmapped cell to robot's current location. Intended to be used with a second smart map strategy to improve on the above map_strategy function. Not implemented, since first smart map strategy actually works very well.

random_explore() is the random searching method for exploration.

avoid_dead_ends_explore() is the dead end avoiding exploration method.

smart_map_explore1() is the smart mapping exploration method described above that uses and random.choice() selection method when the prior logic is exhausted.

smart_map_explore2() is the same as the smart_map_explore1(), that uses the explicit map_strategy() method selection precedence when the prior logic is exhausted.

The next series of functions after these are used for the exploitation phase for the second run.

I have created implementations of both A* and D*-Lite search algorithms for the exploit function.

a_star_search1() implements the A* search algorithm which is a combination of Dijkstra's search algorithm and Greedy-Best-First search. It evaluates each neighboring cell to any location by computing the cost of moving to the neighbor from the current location. This cost is computed as the number of steps it took to arrive at the neighbor from the beginning plus the Manhattan distance from that neighbor to the goal. It updates each cell's movement cost with whichever is the minimum cost of arriving to that cell. It then sorts these cells according to least cost value and adds the lowest one to a path list consisting of the lowest cost moves from the start to the goal. It returns a list of cells in order from start to goal.

A* requires a map of the maze, the starting position and orientation of the robot, and the location of the goal to work. Although the maze map does not have to be complete for A* to work, the more the maze is mapped the more likely it will find the best path to the goal. So clearly, fast maze mapping will greatly enhance A*'s output.

The cost function for A* takes the form:

$$f(n) = alpha*g(n) + (1 - alpha)*h(n)$$

where f(n) is the A* cost function, g(n) is the movement cost, h(n) is the heuristic, in this case the Manhattan Distance, that measures the number of steps to the goal from the current location:

$$h(n) = |x_{goal} - x_{cell}| + |y_{goal} - y_{cell}|$$

and alpha is a weighting parameter that increases the predominance of either g or h in calculating f. (As it turns out from repeated testing, this has very little effect so it was disregarded for the Robot class.) For my implementation, since the robot can get a substantial speed up from moving in straight lines, I have modified the movement cost to be 1 for no-rotation movement and 1.1 (or larger) for rotations. However, it turns out that this doesn't always yield the fastest route as can be seen with test_maze_01 above.

a_star_search2() reverses the path search direction of the a_star_search1. It searches from goal to start. However, it was not created the correct movement costs and so is not used.

d_star_lite_search() was implemented as a means to evaluate how well A* is working. Whereas A* begins searching from the starting location, D*-Lite begins at the goal. All the movement values for each cell are initiated to an arbitrarily high value in the value dictionary. The values for each cell over every direction are then reduced based on connectedness to neighboring cells and ability to move to it using any of the available movement directions. Similarly, movement costs are imposed to prioritize which of the available neighbors is the least costly to move into from any given cell using one of the directions. So each neighboring cell is assigned a movement cost for each of these directions that are available. The path from the goal to the start is thereby constructed as the series of cells with the least increase in movement cost based on the four available directions for movement.

It turns out that D*-Lite is actually somewhat complex to understand. Although it produces similar results to A*, it does so at considerably slower compute times, which could be problematic in the context of a MicroMouse competition. Additionally, it is not able to find the shortest/fastest path through test maze 2 either. So it winds up being a more complicated, higher overhead, equally performing algorithm. But it does in fact verify that A* is working as well as to be expected.

To verify this on your own, you will have to comment/uncomment the following lines of code in the exploit method of the Robot class:

```
self.goal_route = self.a_star_search1(self.start, self.goal_door_location)
#self.goal_route = self.d_star_lite_search()
```

The route_map() and route_plan() methods use a_star_search1()'s path to return the necessary rotations and movements to next_move as well as create the rmap dictionary to display the route.

Above are all of the core functions necessary to run the robot class in the context of tester.py.

However, I wrote tester_1.py to take advantage of the various environment variables that I created to further test analysis the code for the robot class.

I also wrote the notebook RobotMotionPlanning.ipynb to set up and import all the necessary environment variables and packages for running the tester_1.py script as well as the showmaze_1.py script for creating images of the desired maze with the final solution goal route laid on top.

I wrote solve_maze.py to simply produce the best route with the least moves that the robot would be able to find if it had fully mapped the entire maze.

Since all of the controller methods rely on random.choice() as the final direction selector during exploration, there will be considerable variation in the scores that the robot will produce. As such I wrote another function strategy_test.trials() to run each controller through a number of trials and average the scores for each combination of mapper1, mapper2 and random,avoid dead ends, smart map 1 for compiling the results in the Results section.

So, the hyper-parameters used for resolving benchmark limits was num for the number of trials.

The hyper-parameters rotate_cost and no_rotate_cost where varied as well to find good values that return the straightest paths with the least moves.

Finally, the hyper-parameters COVERAGE_RESET_THRESHOLD, MAP_STRATEGY, and MAPPER where used to verify improvements to the final algorithm.

Refinement

From the beginning I had planned to implement the A* algorithm for finding the path to the goal. However, it needs a map of the maze from which to generate this path. Therefore, the explore() method was going to need to map as much of the maze as possible and find the goal as quickly as possible to reduce the time score component of the first run.

The easiest algorithm to assign to this task would be a simple random choice direction selector whenever the robot entered a cell that had 2 or 3 choices to move toward. Additionally, this should also be a good benchmark to measure any other improvements against. The only way to do worse than random selection would be to implement an algorithm to actually produce worse results, which is pointless.

From observing results of running this controller it was obvious that the robot was revisiting the goal cells, the start cell, and dead ends quite a lot, which is a waste of time. So the next improvement was to prevent the robot from revisiting all three. This seemed to improve scores a bit, but also it was obvious that more gains could be found with more directed movements and faster mapping.

The method update_map1() was the first implementation for building the maze map. It simply received the updated heading and location with the sensor values and updated the distances to the walls for each cell that the robot was currently in.

However, since the robot was receiving sensor values of distances to the walls in all four directions, it was soon obvious that those sensor values could be used to update all the cells with this information in the direction of each sensor. Additionally, it was clear that even when there was a wall in a given direction that the neighboring cell on the other side of this wall also shared this wall and that that neighboring cell could also be updated with this information.

Thus the method update_map2() was created to take advantage of these sensor values to greatly speed up how fast the robot was able to map the maze as it searched for the goal.

Before implementing update_map2(), though, smart_map_explore1() was created to employ some logic as to which cell to choose when multiple directions were available. When that logic was exhausted it simply selected the next cell randomly. But again, we're not guaranteed convergence to the best result due to the randomness.

Finally, I implemented smart_map_explore2() to explicit list out selection precedence based on which quadant the robot was in and whether the goal had been found. After some fine tuning, it became clear that this working exceptionally better than previous efforts along with guaranteeing convergence to the best route to the goal in the least amount of time.

However, as can be seen in the results, as the mazes get larger there is considerable room for improvement. This justifies further refinement for the future.

All of the results for these refinements are reported in the tables in the Results section below.

IV. Results

Model Evaluation and Validation

From the Benchmark section above, I've established to following equations for computing a benchmark:

best score = least moves path + (total route length for least moves path)/30

worst score = most moves path + (1000 – most moves path)/30

performance = (worst score – score)/(worst score – best score)

When I run the robot in smart map mode, with reset upon finding the goal (which always yields the shortest time to reset), along with setting rotate_cost = 3 and no_rotate_cost = 1, tester.py returns the following scores:

Test Maze	# Moves Best Path	Length Best Path	# Moves Worst Path	Best Calculated Score	Worst Calculated Score
1	23	31	44	24.03	75.87
2	30	44	57	31.47	88.43
3	33	50	57	34.67	88.43

Table 1: Summary of Calculated Scores

These scores are used for reference only to qualify the actual scores reported below.

To return these scores I run the script strategy_test.trials('test_maze_##.txt', num = 500). I then choose the lowest score for each maze as the Best Score and highest as the Worst Score to calculate performance scores for all of them.

The decrease in performance across mazes is surely attributable to the increasing size and complexity of the mazes.

However, it can clearly be seen that there is significant performance improvements with the update_map2() (mapper2) method as well as all of the controllers as we proceed to test through each one.

Test Maze	Best Score	Worst Score	Average Score Mapper 1	Performance Score Mapper 1	Average Score Mapper 2	Performance Score Mapper 2
1	24.3	56.53	38.2	0.57	35.82	0.64
2	33	71.5	50.51	0.55	46.75	0.64
3	34.8	73.13	51.69	0.56	48.97	0.63

Table 2: Score Summary – Random Explore – Reset On Goal

Test Maze	Best Score	Worst Score	Average Score Mapper 1	Performance Score Mapper 1	Average Score Mapper 2	Performance Score Mapper 2
1	24.3	56.53	38	0.57	36.07	0.63
2	33	71.5	50.27	0.55	47.32	0.63
3	34.8	73.13	52.21	0.55	48.83	0.63

Table 3: Score Summary – Avoid Dead Ends Explore – Reset On Goal

Test Maze	Best Score	Worst Score	Average Score Mapper 1	Performance Score Mapper 1	Average Score Mapper 2	Performance Score Mapper 2
1	24.3	56.53	30.34	0.81	26.24	0.94
2	33	71.5	49.48	0.57	38.65	0.85
3	34.8	73.13	50.13	0.6	40.23	0.86

Table 4: Score Summary – Smart Map 1 Explore – Reset On Goal

Test Maze	Best Score	Worst Score	Score Mapper 1	Performance Score Mapper 1	Score Mapper 2	Performance Score Mapper 2
1	24.3	56.53	31	0.79	25.43	0.96
2	33	71.5	45.33	0.68	35.53	0.93
3	34.8	73.13	43.23	0.78	39.6	0.87

Table 5: Score Summary – Smart Map 2 Explore – Reset On Goal

Justification

If we take all of the performance outcomes calculated above, we should be able to look at how the effect of improving the mapper and the controller over the random controller using the original mapper implementation has had on the overall performance for each maze as a percentage of that baseline performance measure.

As an example:

$$\frac{\textbf{(Performance(smart map 2; mapper 2)} - \textbf{Performance(random; mapper1))}}{\textbf{Performance(random; mapper 1)}}$$

So from the below we can see substantial performance improvements over the random exploring, basic mapping implementation.

This is clearly a justification for this solution.

Test Maze	Mapper 1				Mapper 2				Performance Improvement
	Random	Avoid Dead Ends	Smart Map 1	Smart Map 2	Random	Avoid Dead Ends	Smart Map 1	Smart Map 2	
1	0.57	0.57	0.81	0.79	0.8	0.77	0.96	0.97	70.18%
2	0.55	0.55	0.57	0.68	0.64	0.63	0.85	0.93	69.09%
3	0.56	0.55	0.6	0.78	0.63	0.63	0.86	0.87	57.14%

Table 6: Performance Improvement Over Baseline

V. Conclusion

Free-Form Visualization

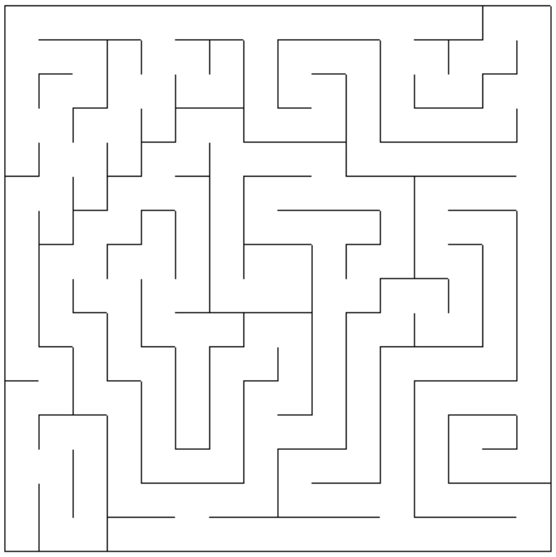


Figure 7: Test_Maze_04

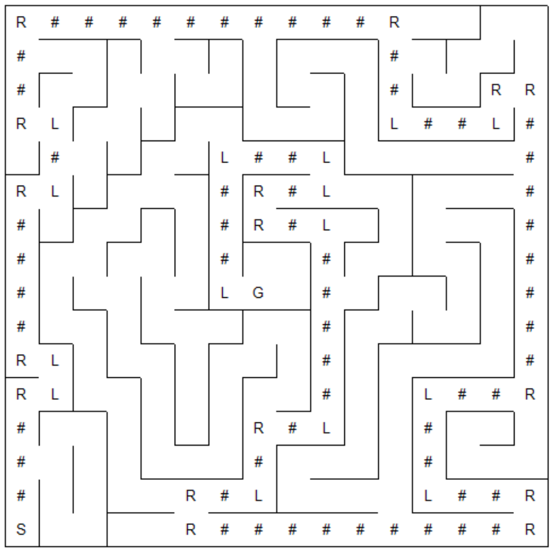


Figure 8: test_maze_04.txt - fastest route - 61 moves

I created another (16 x 16) maze to test code against. I added in a dead end that is actually a loop which is the first turn right of the start corridor. This actually raised an issue with my smart map logic. Before this I had second preference set to closest to goal after the least mapped. But that order was causing the robot to get stuck in this dead end loop at the beginning.

This caused me to switch the second and third choice preferences. So now the preferences are least mapped, least visited, closest to goal. That solved that problem and didn't affect performance on any of the others.

The best route for this maze using mapper 2 is 95 long, 61 moves, and 31 rotations. The worst path that I could get the random_explore() method with mapper1 to find required 76 moves.

Test Maze	# Moves Best Path	Length Best Path	# Moves Worst Path	Best Score	Worst Score
4	61	95	78	64.17	108.73

Table 7: Summary of Calculated Scores

As you can see, running in 'random'/'mapper 1'/'goal' mode can yield even worse scores than what is calculated.

Controller	Best Score	Worst Score	Mapper 1		Mapper 2	
			Score	Performance Score	Score	Performance Score
Random	67.23	120.67	87.05	0.63	84.32	0.68
Avoid Dead Ends	67.23	120.67	87.11	0.63	83.28	0.7
Smart Map 1	67.23	120.67	81	0.74	81.17	0.74
Smart Map 2	67.23	120.67	80.13	0.76	77.67	0.8
Smart Map 2 (Reset on 97%)	67.23	120.67	75.03	0.85	74.9	0.86

Table 8: Score Summary – test_maze_04 – Reset On Goal

Performance Improvement = (0.8 – 0.63)/0.63 = 27%

The last row was added due to a the lack of performance improvement between Smart Map 1 and Smart Map 2 with reset on 'goal' for both mapper 1 and 2. From the coverage map I could see a hole in the coverage map with the cells necessary to finding shorter routes. Therefore, with the fact that reset on goal map coverage was 95%, I started rerunning tester_1.py with 1 point increases to coverage until landing on 97% as the one finding the best path at 61 moves and a score of 74.9 for Smart Map 2/mapper 2 – a substantial improvement.

This is directly the result choosing the cell closet to the goal when multiple direction choices are equally mapped and equally visited at a crucial cell. Unfortunately, at this time I don't have a good solution for an explore() controller figure out how much longer the robot should keep mapping once the goal has been found. It would have to involve the controller running exploit() internally to find it's best route given the map and then keep exploring until it finds an unvisited cell. Then run explore() again to see if there's improvement in the number of moves, as well as keep track of the score. Finally, it would need logic for determining how much more searching would likely yield a better score improvement – every new route it finds with a decrease of 1 movement would be offset by an increase of 30 moves during the first run. So, if it finds a better route in under 30 additional moves beyond finding the goal it could accept that or roll the dice and keep looking. I will leave this for future improvements.

The improvement from the coverage increase was found only after discovering that I needed to gradually up the rotation_cost to 3.1. Prior to this, the best route the above set up was finding was 66 moves with a score of 77.67. It needed more rotate_cost to finding better routes.

I had had success with no_rotate_cost = 1 and rotate_cost equal 1.1 – 3.0 for mazes 1 – 3. But when I bump the rotate_cost to 3.1, A* finds a straighter albeit longer path for maze 3, thereby reducing it's score from it's best of 39.6 to 40.6. The other mazes seem to solve the same regardless.

So, now this makes that case that there is some gain to be had in making exploit() robust to new maps by giving it the ability to rerun a_star__search1() through multiple values of rotate_cost = 2, 3, 4 for every maze before choosing the goal_route. Again, though, I will leave this for future improvements.

Reflection

The first step that took the longest was how to understand the problem space for approaching this project. After a bit of research, it become clear that I would need to take Udacity's AI-Robotics class to better understand some of the implementations for the A* algorithm that was used for finding the best route solution to the maze.

Additionally, during my research I relied heavily on Amit Patel's RedBlobGames website to better understand his version of the A* and for a far better in depth discussion of how it was developed and how to think through setting up an implementation that took advantage of Python's dictionary data structure.

I further explored the D*-Lite solution and the AI-Robotics course implementation of it as well.

While reviewing the example report that was offered in Udacity's capstone project introduction to the Robot Motion Planning project, I discovered that the student used a bit of a cheat in that he simply fed the fully mapped maze as a grid into his A* implementation, which struck me as skirt a singular challenge for this whole project.

So, I decided early that I would approach as though I was participating in the actual competition. As such, my robot would only have one maze map to feed into A* and that would be whatever it managed to acquire while exploring.

From there, I set about thinking about to structure the maze-map itself and how the robot would build it. With that it soon became clear what all the other functionality would be required to achieve that end. I would also need to implement the absolutely easiest, simplest code to return rotation/movement values back to the tester to have a baseline to improve upon. I would need a position updater, a map updater, a goal finder, and an explorer. From there I would then need the A* algorithm.

I wanted to take advantage of python dictionaries as the main data structures as my understanding of them is that they are far more efficient to work with than arrays.

The relationship between A*, the cost function, finding the neighbors, updating the cost map, and how all of these methods would rely on the robot's maze map was particularly challenging. There were many similar implementations of how to check whether any particular cell in the maze was in bounds and cell checking in general that would not related too well to they way I constructed the maze map dictionary. Again, this was part of the challenge of converting from other implementations using arrays to mine using the dictionary structure.

The next significant challenge was implementing methods for analysis problems in my logic. I had to create multiple print functions and data map structures so that I could more easily visualize how the algorithms were proceeding so that I could locate where errors were occurring and, ultimately, why. Displaying the cost maps, which are what A* was assigning to each cell as it was working through the maze_map that was passed to it, helped me to understand different problems with my code. In one implementation I tried to make A* begin at the goal, without actually knowing the exact goal door location and work it's way back to the start. The cost map showed me that it was just creating values that weren't making sense. This why I ultimately went back to directing A* from start to the actual goal door location.

Similarly, setting up visualizations for the goal route was extremely helpful in evaluating whether A* was converging to the best route and what effects changing values for rotate_cost and no_rotate_cost had as well as changing the percentage of the maze that was mapped.

With the goal route path in hand, I then needed to implement a display for each movement as well as finding values for rotations and movements that would take advantage of multiple cell movements in straight lines.

Concurrently to all of this, I realized better methods for exploring the maze. Although I came up with avoid dead ends fairly early on and worked through it, I had been working on better methods for choosing which direction to select when multiple directions were available. This was another large challenge that I had been exploring various algorithm to implement. I looked at left-wall followers and right-wall followers and Tremaux and didn't really love any of them by themselves. I finally settled on simply running through a decision tree of how to choose a direction with the finally layer being a random choice. I still didn't like this because it didn't guarantee a convergence that could be relied on every time: the random choice would create too much variation in the explore, thus cause bigger swings in the scores than I would have liked for an actual competition.

That's when I started to toy with simply creating a precedence list of choices based on where in the map the robot was and whether the goal door had been found. Ultimately this works pretty well, although performance drops off considerably with larger more complex mazes, suggesting maybe revisiting Tremaux, or flood-fill or something else.

Beyond this, I created more interactivity between the tester_1.py function and robot to make analysis and presentation easier and clearer by creating the RobotMotionPlanning notebook.

Through extensive analysis of the algorithm, I have decided, as mentioned in the section Free-Form Visualization, to set rotate_cost = 3 and no_rotate_cost = 1. These values give good robustness across the three test mazes although rotate_cost = 3 gives slightly worse results for my test maze. Even though it produces a worse score for maze 4, I am hopeful that it will it still be useful for other mazes.

Improvement

Consider if the scenario took place in a continuous domain. For example, each square has a unit length, walls are 0.1 units thick, and the robot is a circle of diameter 0.4 units. What modifications might be necessary to the robot's code to handle the added complexity? Are there types of mazes in the continuous domain that could not be solved in the discrete domain?

By and large this would entail adding logic for handling the probabilistic nature of making all the discrete cost calculations. Additionally, there would need to be an accounting of the steering noise, steering drift, distance noise, sensor noise, and path smoothing. All of this would go toward making adjustments to the value grid/cost grid/etc. when making adjustments to the steering and speed.

Further, we wouldn't be dealing with discreet rotations and distances of 90 degrees and 1 cell. Rather, we'd need to account for steering angle and velocity. We would also need real time optimizations of these values using PID controllers.

My implementation here would fail in a continuous domain where there w need for an angular or circular path finding capability, since my robot can only make 90 degree turns. Additionally, the continuous domain would have to modified for distinguishing between turning and moving along straight paths. Rotate_cost vs. no_rotate_cost would have to be substituted with a continuous 'straightening' parameter set of some kind.

Other improvements I would consider would be to add an explore() mode logic to make some guesses as to whether continued exploration based on the current mapping of the maze (with goal found) is warranted to potentially finding a better route for a better score.

The other improvement would be to the exploit() mode for actively looking for straighter routes by increasing rotate_cost.