Capstone Project: Plot and Navigate a Virtual Maze

Machine Learning Engineer Nanodegree

Xu Shiyan

June 23, 2016

Definition

Project Overview

This project aims to solve virtual mazes by designing a robot (in program) such that it can explore and navigate to goal cell inside the mazes. The robot is initially placed at one corner of the maze and allowed to explore it for the first time. The robot is supposed to find an optimal path to the goal based on the map sensed during exploration. At second time, the robot will be placed back to its initial position. It should be able to navigate through the optimal path and reach the goal as fast as possible.

Problem Statement

Maze

The virtual maze is a *n* x *n* grid of squares, where *n* is 12 or 14 or 16. The below picture illustrates a corner of a sample maze. Black lines indicate walls that robot cannot pass through. Grey dotted lines indicate open sides of grid cells. A complete maze will have walls all along the perimeter and random walls among the grid cells. The goal cell is a 2 x 2 grid cell at the center with no wall inside. Number inside a grid cell is a numerical way to describe which side of the cell is open and which is close. Let *t*, *r*, *b*, *l* be Boolean (1 or 0) to indicate whether a wall exists at top, right, bottom, left side of a grid cell, where 1 indicates open (no wall) and 0 indicates closed (walled). Then number inside a cell, S, will be a sum such that

$$S = 1t + 2r + 4b + 8l$$

where *t*, *r*, *b*, *l*, is either 1 or 0.

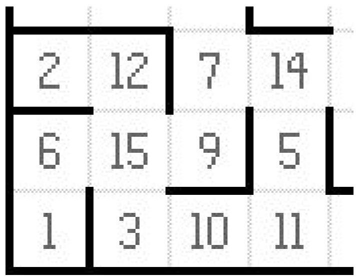


Figure 1: Left bottom corner of virtual maze (Ref [1])

The bottom-left cell will be the starting cell with top open and other sides walled. Therefore, its cell value $S = 1*1 + 2*0 + 4*0 + 8*0 = 1$.

Problem

As mentioned in the overview section, the robot will run twice. At the beginning of each run, robot will be placed at the bottom-left cell and always faces up (face to the top side of cell). Robot cannot pass through walls; it can only go to another cell through an open side, by first rotating -90, 0 or 90 degree then followed by moving forward or backward over 1 to 3 cells. At each time step, robot will receive a set of distance information, indicating which sides of its current cell are open (robot's front facing side and robot's left and right sides) so as to determine the next movement.

The problem is to design robot's `next_move` function, which takes in the sensor distance information, to realize these results: during first run, it can gather and store mapping information of the maze (include center goal cell). During the second run, it should navigate to reach center goal cell as fast as possible.

Metrics

During first run, robot must enter goal cell and then choose either to end the run immediately or to continue exploring. During second run, robot should move as fast as possible to enter goal cell. The total time steps for two runs should not exceed 1000. The performance is evaluated using number of time steps used for each of the two runs. Let *t1* be time steps used in the first run, *t2* be time steps used in the second run.

$$Performance\ metrics\ P = t1/30 + t2$$

where $t1 + t2 <= 1000$

Obviously, as the metrics suggest, smaller the P, better the performance. Time spend on first run has much less impact to the final performance. It would sometimes help to improve the performance by spending more time to explore and gather more data of the maze. With more data, the robot would be likely to find a better path so that in the second run, it could navigate to goal cell faster.

Analysis

Data Exploration

The robot is able to sense its front, left and right sides. At each time step, robot first senses distances to walls from those sides, and then makes a move, which consist of a rotation and a following movement. Rotation is -90 or 0 or 90 degrees. Movement is an integer in [-3, 3] inclusive, indicates how many cells it can advance or reverse. It advances or reverses until a wall blocks the way.

At time 0, robot will always rotate 0 degree and move forward at least one cell because the initial cell is always open at top side only and robot always initially faces up.

During first run, the robot will always process sensor data to complete the map and will not take forward or backward movement larger than one cell because its belief of how the maze is structured usually changes at next time step, and it should take every opportunity (i.e., every one movement step) to map out the maze as much as possible.

During second run, the robot will not process sensor data any more. Rather, it should have already found out an optimal path to the goal based on the map data it gathered during first run. The job of second run is to navigate to the goal as fast as possible hence there will be sometimes more than one cell advancement at single time step.

Example: test_maze_01

This is a 12 x 12 maze. The goal cell (2 x 2) has only one half open side on the upper right; this is the only entry to the goal. By observation, there are many paths that can lead to the goal cell. The optimal path takes 17 steps, which is shown in next section. There are also quite a few dead-end cells with only one side open; once robot enters it, a reverse is needed to move out.

Exploratory Visualization

As shown in the pictures below, the initial cell at left bottom corner is fixed in the way that only upper side is open, and the initial heading of the robot is always upward.

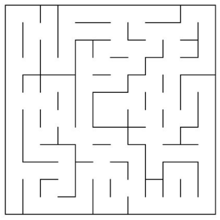


Figure 2: Sample virtual maze visualization

There are many paths that can lead to goal cell in `test_maze_01`. As shown in pictures below, red path is one possible path to the goal, however, it is not the best optimal path. The green path shown below is the optimal path, which takes 17 steps in total considering it takes 3 or 2 cell advancements at some segments, which shortens the time to reach goal.

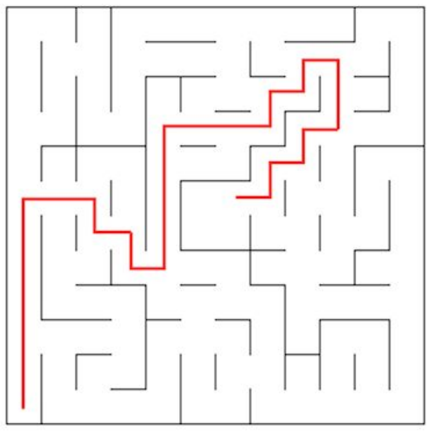


Figure 3: A path to goal

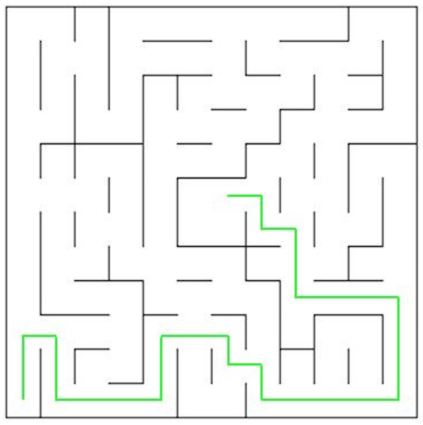


Figure 4: The optimal path to goal for this maze

Algorithms and Techniques

A* search algorithm will be used for the first run. The first run aims to map out the maze in preparation for the second run. The first run aims to explore unknown environment and record data such that the environment can be reconstructed in the robot's "memory" as much as possible. The most important thing to achieve during the first run is to reach the goal cell (which is also a hard rule) so that a path can be found from start cell to it. In the context of this problem, starting cell is always at left bottom corner and goal cell is always at the center of the maze.

A* search algorithm matches the characteristic of this problem. Firstly, it is a search algorithm that guarantees to find a path to goal if one exists. It expands search path by trying out all possible next movements from one cell and maintains a list of unvisited cells for subsequent expansions. Secondly, A* prefers to work with a heuristic matrix that guides the search path to expand towards the goal cell more quickly. This problem's context provides exactly a heuristic for A* to use, which is the distance to the center goal cells. It can set that heuristic value being the minimum for goal cells and the maximum for cells along the perimeter and decaying as moving closer to the center. Thirdly, to carry out the search within finite time, some constraints of the maze have to be applied, such as boundary of the maze and location of walls that could block movements. These information has been provided as input data in text files.

Dynamic programming will be used for the second run. The robot takes input data from its first run's mapping results. As observed from maze visualization, there are many variations of the path leading to the goal; as the robot may have chance to take different paths to goal (due to equal evaluations on paths done by A*), it could map out different part of the maze from time to time. It is not likely to map out the entire maze because heuristic matrix helps A* algorithm work faster to find the goal by not visiting unnecessary cells. Therefore, since reconstructed map based on first run result could vary from time to time, the robot needs "dynamic" policies to follow during second runs; dynamic programming is a match to this characteristic.

Dynamic programming considers all possible combinations of location and heading of the robot and assigns value to each combination based on minimum cost over all possible next movements. The values propagate from goal location (with 0 value) to other connected cells. No matter how different each time the robot maps out the maze, dynamic programming could always find an optimal policy to follow. Although it has high computation complexity, in this problem's context, maximum maze size is only 16 x 16, which gives only 1024 combinations of robot's states (with 4 possible headings). With 16 possible movements at each state (4 directions and maximum 3 unit distances gives 16), any modern computer can easily calculate those values within looping logic. (1024 x 16 = 16384 value calculations per loop).

Benchmark

Assume robot needs to visit every cell twice during search to obtain sufficient information, the first run will score `maze_size ^ 2 x 2 / 30`. For second run, the idea path to goal is a "L" shape path directly leading to goal area, which has length of `maze_size`. It is reasonable to think that the actual path is more twisted and could be twice the length of idea one. Therefore second run score will be `2 x maze_size`. The benchmark for the final score would be

`n ^ 2 x 2 / 30 + n x 2`, where n = maze_size.

Methodology

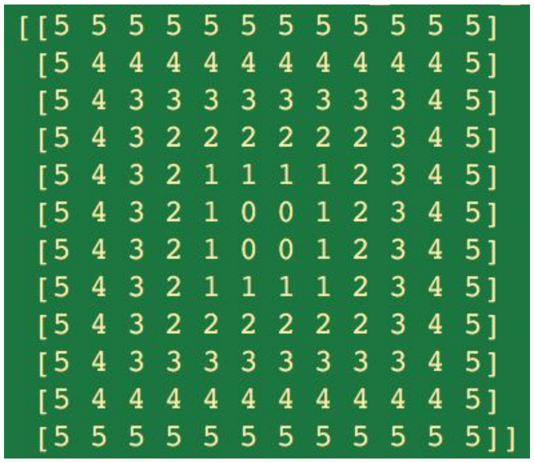
Data Preprocessing

No data preprocessing is needed because the starter code has implemented class and methods (`maze.py`) to construct the maze environment for robot to explore. The starter code also implements sensor specifications and processing logic (`dist_to_wall()`) to feed the robot during exploration.

Implementation

In this section, all discussions regarding implementation details will be using `test_maze_01` as example, which has 12 x 12 cells.

During robot initialization, the first thing to do is setup heuristic matrix. As discussed previously, heuristic helps A* search find goal more quickly. In this problem, because the maze is fixed to even number size and the goal is fixed at the center, it is feasible to increase heuristic values radially from the center, which has 0 values for the 2 x 2 area. This tends to guide robot to move to the center to reach goal more quickly. For `test_maze_01`, the heuristic matrix looks like this:



```
[[5 5 5 5 5 5 5 5 5 5 5 5]
 [5 4 4 4 4 4 4 4 4 4 4 5]
 [5 4 3 3 3 3 3 3 3 3 4 5]
 [5 4 3 2 2 2 2 2 2 3 4 5]
 [5 4 3 2 1 1 1 1 2 3 4 5]
 [5 4 3 2 1 0 0 1 2 3 4 5]
 [5 4 3 2 1 0 0 1 2 3 4 5]
 [5 4 3 2 1 1 1 1 2 3 4 5]
 [5 4 3 2 2 2 2 2 2 3 4 5]
 [5 4 3 3 3 3 3 3 3 3 4 5]
 [5 4 4 4 4 4 4 4 4 4 4 5]
 [5 5 5 5 5 5 5 5 5 5 5 5]]
```

Figure 5: Heuristic matrix of test_maze_01

The heuristic values are certainly admissible for exploration because they are always less than or equal to the cost incurred in reaching goal, given that robot will only advance or reverse one cell during exploration, reason of which has been explained in data exploration section.

There are three other important matrices (all have the same size as maze 12 x 12) to setup during initialization.

- First is **`walls`**, which stores wall information of each cell inferred from sensor data. It is a matrix of dictionary (e.g.: `{'u':1,'r':0,'d':0,'l':0},` where 1 means open and 0 means closed.), which is easy to query the ability to move from one cell to another given direction string like `'u'`, `'r'`, etc.
- Second is **`pathCounts`**, which stores visiting counts for each cell. These counts will serve as penalty multipliers to cost calculation; the larger the count is, the higher penalty will incur in search algorithm. Robot will tend to avoid moving to visited cells during exploration so as to gather more maze data. These counts will be retained for boundary checking at the second run; those cells haven't been visited at the first run are considered as unreachable and will not be visited at the second run.
- Third is **`paths`**, which stores sequence for each cell that the robot has visited. It is not required to run successfully but helpful in debugging and proving A* algorithm works as expected; it can also show how the robot navigate to goal at second run.

First run implementation

At the beginning, robot has no information of the maze except the starting cell, which only opens at upper side. Hence all other cells are marked as open in all sides.

At each time step, a set of sensor data is fed to robot and tells it how many cells away each wall locates at from its left, front and right side. This information is used to update **`walls`** matrix in **`update_walls(self, sensors)`**. The tricky part in this wall update is: there are usually two adjacent cells sharing the same wall in the maze and both cells need to update its own wall information. This method update all related cells' related sides covered in this round of sensor scan. This set of data therefore increases the robot's belief of the maze.

After a small part of the maze data is gathered, the robot can perform a search on the maze given the goal cell's location. At the beginning, the robot may naively find out an "L" shape path directly to the goal. This is acceptable because only a small part of maze is known to the robot. As more sensor data is processed, the robot is able to search paths that are more "realistic".

The A* search algorithm, as explained in analysis section, guarantees to find a path to goal if one exists. It expands the search by evaluating the costs of moving to next possible cells and choosing minimum cost cell. The cost is consist of heuristic value, step cost and past visit penalty. Heuristic value has been discussed in this section previously. Step cost is set to 1 (because this problem is in discrete context) and will incur in every movement during search. Past visit penalty is set to multiple of visit counts. Let the multiplier be **p**, then if a cell has been visited twice, the total past visit penalty will be 2 x **p**.

Multiplier **p** is set to maximum value in heuristic matrix. The reason is, for a cell that has been visited, it should be avoid visiting again. By setting it to maximum of heuristic value, a cell with just 1 more visit than another will guarantee to have larger cost than the other. In the example of `test_maze_01`, imagine there are two cells (A and B) to continue the search. A has visit count of 2 and heuristic value of 3; B has visit count of 1 and heuristic value of 5. The maximum heuristic value is 5 hence **p** = 5. The transition cost for A is 2 x 5 + 3 + 1 = 14; the transition cost for B is 1 x 5 + 5 + 1 = 12. Here B is the preferred cell to continue searching even it has larger heuristic value. The rationale is that heuristic values is a naive guidance whereas actual visit is a much stronger factor to consider for search.

After the robot reaches goal for the first time, this implementation chooses to continue exploring the maze only if the percentage of visited cells over total cells is below a certain threshold **t**. This is to increase the probability that robot becomes aware of an alternative yet better path to goal by ensuring sufficient information of the maze. Threshold **t** can be further fine-tuned to achieve better results, which will be discussed in "Results" section.

Second run implementation

As discussed in "Algorithms and Techniques" section, dynamic programming is used for policy searching. It is suitable here because mazes mapped out from first run are different from time to time due to different search paths. Therefore a dynamic policy is needed to handle this dynamic input. The essence of this algorithm is to calculate all possible scenarios; it puts robot in all possible states (maze_size ^2 x 4 headings) and considers all possible transitions (4 directions x 3 distances = 12). A 3D matrix is (maze_size x maze_size x 4 headings) needed to store best policies. An equal-sized matrix is needed to store values to compare between policies.

Values indicate how many time steps required from one location to goal. Hence goal locations are assigned to 0 values, regardless of headings. Those location and heading combinations that may take one single time step to goal are assigned to value 1. For example, the location just next to goal cell is of value 1; the location of 3-cell far away from goal is also of value 1 because robot can move maximum 3-cell distance at once.

The algorithm is designed such that value matrix is initialized to very large numbers (999). All value elements will be recalculated as long as one of the elements is found to have a smaller value, which is in fact the trigger of value updates propagation; when goal values set to 0, its adjacent cells obtain much smaller values (usually 1). The same happen to farther adjacent cells, which causes chain effect that minimizes all values and finally stops the loop when value updates are stable. In the process of value updates, the corresponding policy (rotation and movement) of each minimum value was captured in policy matrix.

Refinement

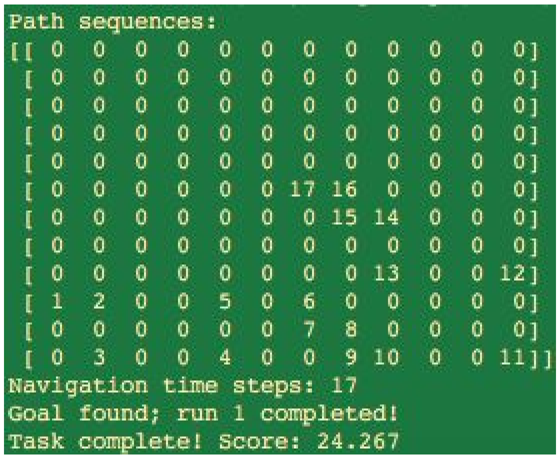
Initially the cost of A* search is set to step cost (1) plus heuristic value. It was observed that robot always exceed 1000 time limit to reach goal in the first run. This can be reproduced by setting **`self.visited_cell_penalty`** to 0. The reason behind this behavior is that robot may go circling along some paths in the maze without realizing that those cells have been visited in the past. Hence, by incorporating past visit penalty into search cost, robot can be guided to avoid going back to old paths and tend to find new paths.

When robot first reaches goal during search, it is possible to continue to explore when cells' visiting ratio is below the threshold **t**. When start to continue exploring, the heuristic value should be inverted as in the innermost cells should have largest values whereas outermost cells should have smallest values (0). This could help to drive robot to explore more of non-center area so as to find potential better paths to goal. Matrix **`inv_heuristic`** is used to store these values.

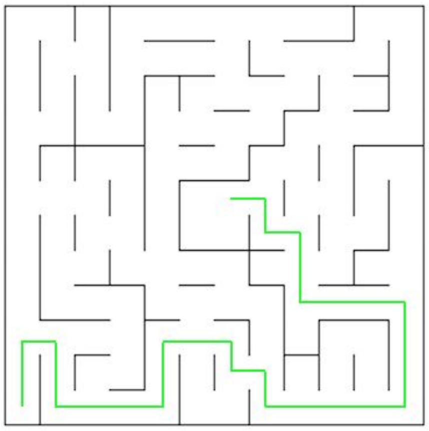
Results

Model Evaluation and Validation

Result of one test run against test_maze_01 is shown in the screenshot below. As shown in the path sequence matrix (`0, 1, 2, 3, … , 17`), robot was able to follow the optimal path and reach the goal.



```
Path sequences:
[[ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0 17 16  0  0  0  0]
 [ 0  0  0  0  0  0  0 15 14  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0 13  0  0 12]
 [ 1  2  0  0  5  0  6  0  0  0  0  0]
 [ 0  0  0  0  0  0  7  8  0  0  0  0]
 [ 0  3  0  0  4  0  0  9 10  0  0 11]]
Navigation time steps: 17
Goal found; run 1 completed!
Task complete! Score: 24.267
```



To compare with maze visualization on the right, the green line that indicates optimal path has matched path sequence printed out from the program on the left.

A looping tester (`tester_loop.py`) is developed to carry out large number of tests to test the robot model's robustness. It has the same testing logic as `tester.py` except that it has an outside loop trying out different "sufficent_visit_threshold" **t** each for a fixed number of times (`test_count`). Threshold **t** can be tuned to help robot achieve better performance.

Line 143 of `tester_loop.py` guarantees the success of each test on robot because if any of test fails (due to exceeding time limit), that assertion will also fail.

The loop tester is set to test **t** of 0.0, 0.2, 0.4, 0.6, 0.8. Each **t** value is tested 100 times. With 500 tests on the robot passed for all 3 mazes, it can be concluded that the robot model is robust enough to solve similar mazes.

The average scores are recorded in this table

	No. of tests for each t	Average scores					Score difference (worst -best)
		t=0.0	t=0.2	t=0.4	t=0.6	t=0.8	
test_maze_01	100	24.333	24.265	24.708	24.760	25.667	1.402
test_maze_02	100	39.923	39.971	40.665	36.438	35.516	5.149
test_maze_03	100	46.021	45.230	45.623	41.450	38.801	7.220

** Scores in red are the best average score for each maze, blue are the worst*

t=0.8 gives the best average score for maze 02 and 03. Although it doesn't give the best score for maze 01, it still achieves an average of 25.667, which is only 1.402 score difference from best score. Consider score difference for maze 02 and 03 are much larger than maze 01, meaning t=0.8 improves performance on maze 02 and 03 much more than it worsens for maze 01, it can be concluded that t=0.8 is the chosen threshold for the robot.

Justification

Based on the benchmark proposed in previous section,

`n ^ 2 x 2 / 30 + n x 2`, where n = maze_size

each of the three test mazes has following benchmarks.

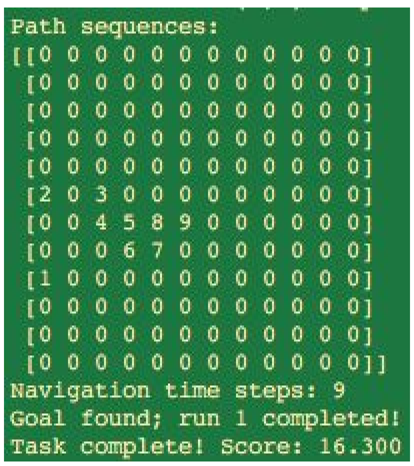
	maze_size	benchmark	worst average score
test_maze_01	12	33.600	25.667
test_maze_02	14	41.067	40.665
test_maze_03	16	49.067	46.021

Compared to worst average score table in previous section, all 1500 tests have passed the benchmark, indicating the robot model is very good and robust at solving mazes.

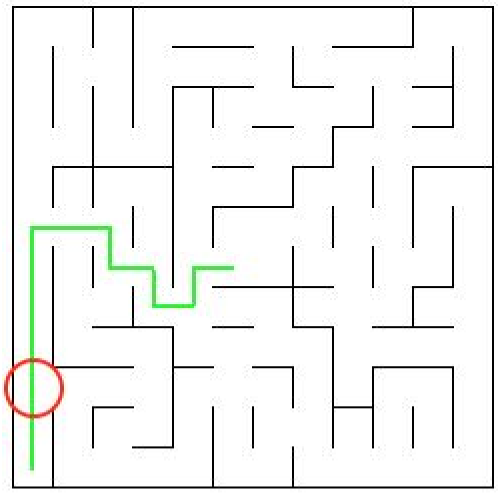
Conclusion

Free-Form Visualization

By making a small variation a test_maze_01 (as shown below), a better optimal path with only 9 steps can be found as indicated by green line. And the robot model is able to find that path as well.



```
Path sequences:
[[0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [2 0 3 0 0 0 0 0 0 0 0 0]
 [0 0 4 5 8 9 0 0 0 0 0 0]
 [0 0 0 6 7 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]]
Navigation time steps: 9
Goal found; run 1 completed!
Task complete! Score: 16.300
```



There is a chance that when robot is at junction circled in red, it turns right and misses out the optimal path completely during exploration. By manually ran 10 times for this modified maze, there were 9 times that robot was able to find this new optimal paths, which proved its robustness.

Reflection

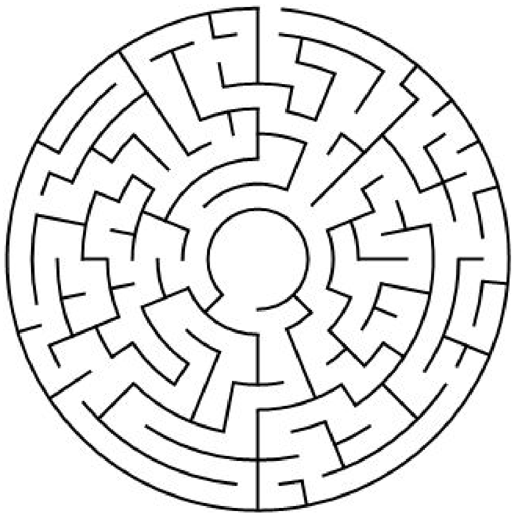
This problem was solved in a structured approached. It was first analyzed thoroughly by studying the maze structure, robot characteristics (movements and sensors) and game rules. Then appropriate algorithms were chosen to tackle different problems: exploration and navigation. Finally, after carefully implementing those algorithms, loop tester logic was setup to further fine-tune parameter and test robot's robustness.

The most interesting and also difficult part of this problem is to discover an appropriate way to calculate costs for A* search. It is common to only consider heuristic and step cost at the beginning. However, by printing out path sequence and path count matrices, it could be revealed that past visit penalty plays an important role in exploring the mazes effectively.

Improvement

By introducing thickness and size to walls and robot, the distance information provided by sensors will be different from previous discrete setup. The robot model will no longer receive 0 distance that indicates current cell having wall previously. Instead, 0.2 unit distance (1/2 - 0.1 - 0.4/2) will indicate that. Similarly, a distance measurement larger than 0.2 unit will be subtracted by 0.1 unit (subtract 0.2 from current cell and plus 0.1 as thickness of wall hitted) to indicate that wall belongs to a cell located that many units away. Therefore, function `update_walls` will need to be modified accordingly to adapt the above changes.

Round mazes in continuous domain as below could not be solved in discrete domain because to navigate in maze like this, robot will have infinite headings instead of 4 in the discrete domain. Also the heading needs to change continuously along with the movement so as to travel along curved ways. Both A* search and dynamic programming will become inefficient because there is too much computation to process.



Round maze (Ref [2])

Ref:
[1]
https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5IUM5OpAXJYiutFuSIjTzV_E-vdE/pub

[2] http://www.mazegenerator.net/static/theta_maze_with_20_cells_diameter.png