Robot Motion Planning

Plot and Navigate a Virtual Maze

Capstone Project

Vishwas V
April 26th, 2018

I. Definition

Project Overview

This 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 multiple 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. In subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned.

The maze is made up of a 16 by 16 grid of cells, each 180 mm square with walls 50 mm high. The mice are completely autonomous robots that must find their way from a predetermined starting position to the central area of the maze unaided. The mouse will need to keep track of where it is, discover walls as it explores, map out the maze and detect when it has reached the goal. Having reached the goal, the mouse will typically perform additional searches of the maze until it has found an optimal route from the start to the center. Once the optimal route has been found, the mouse will run that route in the shortest possible time.

Problem Statement

In this project, we 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; our goal is to obtain the fastest times possible in a series of test mazes. 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 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.

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 counter clockwise 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.

Metrics

For each of the test mazes, the virtual robot moves through the maze twice. In the first run, the virtual robot can move freely through the maze in an attempt to explore and map it. It is free to continue exploring the maze even after entering the goal area. Once the virtual robot has found the goal, it may choose to end the exploration run at any time.

The second run of the environment, the agent will be returned to the starting position and orientation. The agent's goal is to then navigate to the goal area in fastest time possible, minimizing actions (time steps) taken by the agent.

The score awarded to the robot for each maze is the sum of the two following terms:

- The number of steps taken while exploring the maze during the first run divided by 30.
- The number of steps taken to reach the goal during the second run of the maze.

Each maze is limited to a maximum of 1000 steps for both runs combined.
A virtual robot with a smaller score performs better than one with a larger score.

It is important to note that this scoring metric penalises both robots that fail to find the optimal path to the goal as well as those that explore the maze in an inefficient manner. That said a robot that limits exploration and fails to find the optimal path to

the goal is likely to, but may not necessarily, perform worse than a robot that takes the time to explore the maze sufficiently and finds the optimal path to the goal.

II. Analysis

Data Exploration

The starter code provided by Udacity includes the following files:

- robot.py - This script establishes the robot class. This is the main script where modifications were made to the project.
- maze.py - This script contains functions for constructing the maze and for checking for walls upon robot movement or sensing. This is the file which will provide the sensory inputs required at each step.
- tester.py - This script will be run to test the robot's ability to navigate mazes.
- showmaze.py - This script will be used to create a visual demonstration of what a maze looks like.
- test_maze_##.txt- These files provide three sample mazes upon which we will test our robot. The ## is replaced by 01, 02 & 03 for three sample mazes.

The contents of a maze file test_maze_01.txt looks like this:

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

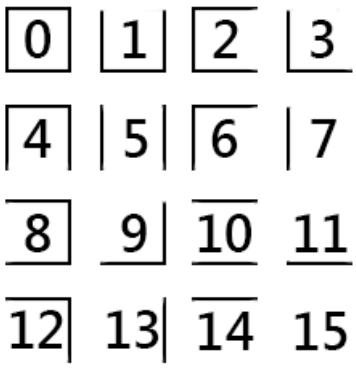
Mazes are provided to the system via text file. On the first line of the text file is a number describing the number of squares on each dimension of the maze *n,* in this case it's 12. On the following *n* lines, there will be *n* comma-delimited numbers describing which edges of the square are open to movement. The numbers are in range of 0 to 15. 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).

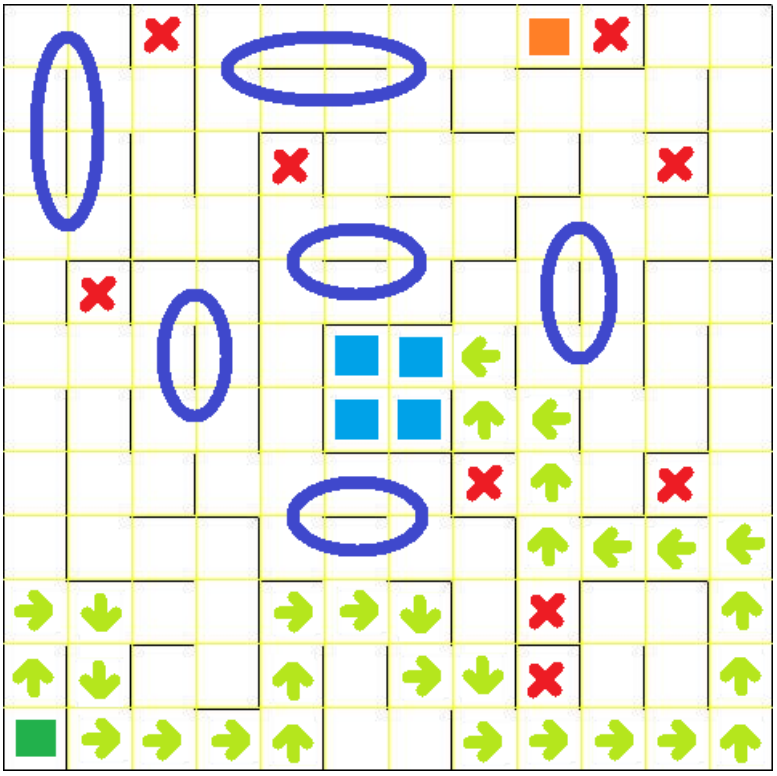
As for the robot, at the start of a time step the robot will receive sensor readings as a list of three numbers indicating the number of open squares in front of the left, centre, and right sensors (in that order) to its “next_move” function. The “next_move” function must then return two values indicating 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 counter clockwise, 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

A visualization of the walls corresponding to the range between 0 – 15 as described in the previous section is shown below.



The visualization of the first maze mentioned in the previous section is shown in the below figure. The start location is denoted by the green square and the goal area is located in the center shown with the blue squares. The shortest path from the start location to the goal area is indicated by the green arrows. The robot is allowed to take maximum 3 steps in one time step. Therefore, the path shown takes 17 time steps to reach the goal, however total no of steps are 30. This also means the robot should prefer straight path than left/right turns. The robot should identify and avoid dead ends as well as recognize any one-way path to a dead-end. The dead ends are marked with red X marks and one way path to the dead ends with an orange square. The robot should avoid going through loops and some of the potential loops have been highlighted with blue circles. During the first run, the robot should keep track of how often it has visited each location in the maze and prefer to proceed to unexplored cells.



Algorithms and Techniques

First Run (Exploration):

The maze here is completely unknown to the robot at the beginning and it cannot make decisions in an unknown environment. So, the first task will be to simultaneously explore and learn the environment. The robot will accomplish this by using its three sensors to receive inputs. The algorithm used to perform the exploration task will be a modified random movement algorithm. The robot will pick random directions to move at each step and will reverse from the dead ends after blocking them.

- The robot must keep track of its location and heading
- The robot must be able to draw the map of the maze with the wall locations
- The robot must prefer going towards unexplored cells
- The robot should be able to detect the goal
- The robot should be able to detect, block and avoid dead ends
- The robot must explore enough maze to conduct the shortest path search

Second Run (Optimization):

During the second run a bunch of path finding algorithms will be used on the known maze. The algorithms we will be using require a prior knowledge of the maze. Since we have mapped the maze in the first run, it is expected that these algorithms will work well. The approach here is that, having a map of

the maze with blocked cells the robot can apply a search algorithm to find the goal and define an optimal path to the goal. It will then create a policy and follow that policy to reach the goal. So at the start of the second run, first a policy will be created by applying a path search algorithm on the maze map and then the robot will begin to move following the policy.

- Graph traversal algorithms. There are two types of graph traversal algorithms, depth-first search and breadth-first search. As the names suggest, depth-first and breadth-first differ in how they select the next node to expand as a part of the search process. Depth-first search will expand a single path until it comes to an end and then start with the next neighbouring node. Breadth-first search will expand all neighbouring nodes progressively working through the graph until the target is discovered. The two strategies can either be a good or poor choice depending on what information is known about the map, and the relative locations of the start and target nodes. Both families of algorithms could be used to explore the maze as well as find the optimal path once the exploration is complete. Both approaches would complete these tasks, but they would not receive the same score. The breadth-first search, which is a special case of Dijkstra's algorithm, requires that the robot continually revisit explored parts of the maze to access the different parts of the unexplored boundary. This strategy would result in a very large exploration scoring penalty.

- Dijkstra's algorithm is used to find the shortest path between nodes in a graph. The algorithm takes a graph and then iteratively steps through each possible path through graph storing the smallest path cost for each node. The algorithm selects the node with the smallest path cost and selects all of the neighbours for that node. If the path cost through the current node to reach a given neighbour is less than the current path cost for that neighbour, then the path cost is updated for the neighbour with the lower value. This process is repeated until the fastest path to the target node is found. The optimal path can then be constructed by walking through the graph and following the nodes with the lowest path costs. Once the robot fully explores the maze and there is a graph that describes each fork, then Dijkstra's algorithm could be used to find the optimal path through the graph.

- A* is an algorithm used for path finding and graph traversal. When given a starting position and target position, A* will progressively search through all possible paths of length n increasing n through the search space and around any found obstacles until the target is found. As soon as the target is found, that path of length n must be the shortest path to the target. There are less naive variants of A* that will also calculate the direct route to the target until an obstacle is found and only then naively search for a path around the obstacle by evaluating all possible paths around the obstacle. Once the obstacle is circumvented, the algorithm continues to take a direct path to the target either until it is found or until the next obstacle is located.

- Flood Fill / Dynamic Programming algorithm determines the area connected to a given node in a multidimensional array. The same algorithm is used in bucket fill tool of paint programs which fills the connected and similarly-colored pixels or area with a new same color. There are many ways in which the flood-fill algorithm can be structured. For this problem, the algorithm will start from the goal location and track back to the starting location giving each surrounding cell a number along the path. The number will represent the minimum steps required to reach the goal from that location. Dynamic programming is a technique which returns an optimal path to reach the goal from any location. Given a map and the goal location, dynamic programming creates the best policy to follow from every position in the map. The algorithm will use a value function to calculate a value (the shortest distance to the goal) for each location in a recursive fashion by considering the best neighbour cell, and adding its value and the cost to move to next cell.

Benchmark

The Benchmark Model for the agent directly relates to how the evaluation metrics are done for scoring the agent's path planning regarding steps taken within the environment. 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 and the agent is restricted to a total of one thousand time steps total for both runs within the environment.

Benchmark Score = (no. of steps in trial 2) + (no. of steps in trial 1) / 30

For the benchmark model we are going to use Breadth First Search Algorithm. Larger area coverage will take more steps and lower area coverage will take less. We need to see how this affects the robot's performance.

Breadth First Search:

Maze	Area Explored	Steps Taken		Score	Average Score
		Run 1	Run 2		
1	78.47	212	21	28.067	28.32
	70.14	223	19	26.433	
	89.58	279	17	26.300	
	96.53	310	20	30.333	
	90.28	314	20	30.467	
2	72.96	366	30	42.200	38.61

	70.41	248	26	34.267	
	95.41	487	25	41.233	
	93.88	401	27	40.367	
	84.7	300	25	35.000	
3	70.31	387	25	37.900	43.15
	79.69	456	29	44.200	
	77.73	351	28	39.700	
	97.27	728	25	49.267	
	80.86	351	33	44.700	

We will take the average score of five test runs as the benchmark for each maze.

III. Methodology

Data Pre-processing

The maze specification and robot's sensor data has been provided and is accurate. Therefore, there is no data pre-processing is required.

Implementation

The process involves four files mainly: tester.py, maze.py, showmaze.py and robot.py.

- The main code for the robot with all the algorithms is written in robot.py.
- The maze.py file is used for sensing the environment and to build the maze.
- The showmaze.py is for visualizing the maze and robot's path.
- The tester.py is for running the robot and visualizing the maze. It calls the robot.py, maze.py and showmaze.py modules.

Exploration:

In the exploration phase, the robot will explore and map the maze using a modified random movement algorithm. The exploration will be carried out by the explorer_robot function. The important part of this phase is that the robot should explore enough area of the maze so as to discover the best path available in the maze and without taking too many steps. To accomplish this we will keep blocking the dead ends as we encounter them and give preference to unexplored cells.
The robot's knowledge of the maze is stored in a two-dimensional numpy array, called Maze Grid. This list stores the information gained on the actual

structure of the maze walls identified from the robot's sensors during its exploration. This working-memory list, populated by the wall_finder method, will enable the robot to know the locations of walls and openings in the maze, and what area of the maze is undiscovered. Starting each time-step, the robot receives a new set of sensor readings from its left, front, and right sensors. Combining the sensor readings with information describing the direction the robot is facing and its current location, the robot is able to generate and store a four-bit number describing the surrounding maze walls and openings for its specific location. In addition to the Maze Grid, a two-dimensional numpy array, called Path Grid was created that represented a grid of the undiscovered maze with 0's on every cell. Additionally, each cell in the Path Grid the robot visits will result in adding 1 to the corresponding cell in the Path Grid, essentially keeping count of the amount of times the robot has visited a particular space in the maze.

While exploring the maze the robot can run into dead ends time and again which could cost us an increase in the score. So to prevent this we can block the dead ends whenever it is detected. The logic to be applied here is that when the robot enters a new cell, all three sensor values will be 0 for a dead end or; for each heading a particular wall value will be a dead end. In addition, we must also take care of the previous cell for which there exists only one valid action which can lead to the dead end. For example, a robot heading 'up' in a cell of wall value '5' may reach a cell with a wall value of '4'. The cells meeting these conditions will be blocked by putting a -1 in those locations in the maze_grid. The robot will check if it has encountered a dead end. If the robot reaches a dead end, it will reverse without any rotation. It will then choose a different action leading to a new cell as the dead end we encountered would be blocked.

The robot will keep making random movements until it has found the goal and reached the area threshold for exploration. With a preference given to unexplored cells the robot is expected to move faster in the maze without making multiple visits to a single cell. However, if there is no choice the robot can still move to the explored cell until it has found a new unexplored cell. This will also prevent getting stuck in a loop up to an extent. Every time the robot takes an action, its rotation is changed using the dir_rotation dictionary and the new heading is changed to the action taken. For example, for a heading of 'up' a rotation of -90 degree is applied if the action taken is left, 0 degree if the action is 'up' and +90 degree is it is right. Once the goal is found and the area threshold is reached, the robot will reset and enter the optimization phase. This way the robot will learn about its environment and in the next run it will try to use this learning to accomplish the task it has been assigned.

Optimization:

After exploration the robot knows the maze and has a map of it. In the optimization phase the robot will first apply a path search algorithm on the map it has created to find the shortest route to the goal and then create a policy to follow before it actually begins to move. The robot will then follow the

policy to reach the goal. The optimization is carried out by the optimized_robot function. At the beginning of this function we will choose an algorithm to perform a search which will create a policy. Then the robot will start moving following that policy. The algorithms available for us to apply are listed below with their function name:

Breadth first search: breadth_first_search()
Depth first search function: depth_first_search()

The breadth first search and depth first search algorithms maintain a dictionary of parent nodes for every extended node till the goal. After reaching the goal, the algorithm stops and backtracks to the starting position using the parent dictionary. The policy grid is built using the path grid by starting at the goal location and putting the action taken in the parent node of its current location and it moves backwards to the parent node and sets it as the current node and repeats the process until it reaches back to the starting position.

Dijkstra's: dijkstra_algorithm()

The Dijkstra's algorithm applied here will search the maze until it reaches the goal. After the goal is reached the path grid is used to backtrack to the starting location creating a policy. It starts from the goal location and moves back to the cell it came from by using the action taken in the current location. Then that action is put into the policy grid for the cell we just moved back to. Then that cell is set as current location and we move back again using the path grid until we reach the starting location creating the policy.

A*: a_star_search()

A star extends the dijkstra algorithm by adding a heuristic guide to strengthen its search. There are three types of heuristics used in A* i.e. Manhattan distance, Diagonal distance and Euclidean distance. Since our problem deals with discrete locations and there are four possible directions to move, we will use Manhattan distance heuristics. Manhattan heuristics is the sum of absolute value of differences between the x & y coordinates of a location and the goal. It also extends the nodes with the lowest cost. To create the policy, it will follow the same process as dijkstra, which is discussed above.

Flood Fill / Dynamic Programming: flood_fill()

This algorithm creates a value grid by taking the walls into consideration and then assigning each cell a value which represents the no of steps that cell is away from the goal location. This value grid is used to create the policy by the compute value function by calculating a value (the shortest distance to the goal) for each location in a recursive fashion by considering the best neighbour cell, and adding its value and the cost to move to next cell. The value grid will be initialized with a large value for each cell and then the compute value function is applied recursively until each cell has a minimum value for reaching the goal.

Refinement

The initial code was setup to block the dead ends and the immediate path leading to the dead end. However, this proved to be insufficient for deeper dead ends with more than one cell block leading to a dead end. This is because after moving back 2 spaces when it found a dead-end along with a cell leading to that dead-end, it had no clue what action to take next and had a tendency of wanting to travel right back into the dead-end when its only possible move was still forward after it had retreated. In order to solve this problem, an improvement was made in the code. The robot is made to go reverse, and block the path leading to these dead ends. It also helps in keeping the step counts low to an extent, as the robot will not enter these deep dead-end paths again.

IV. Results

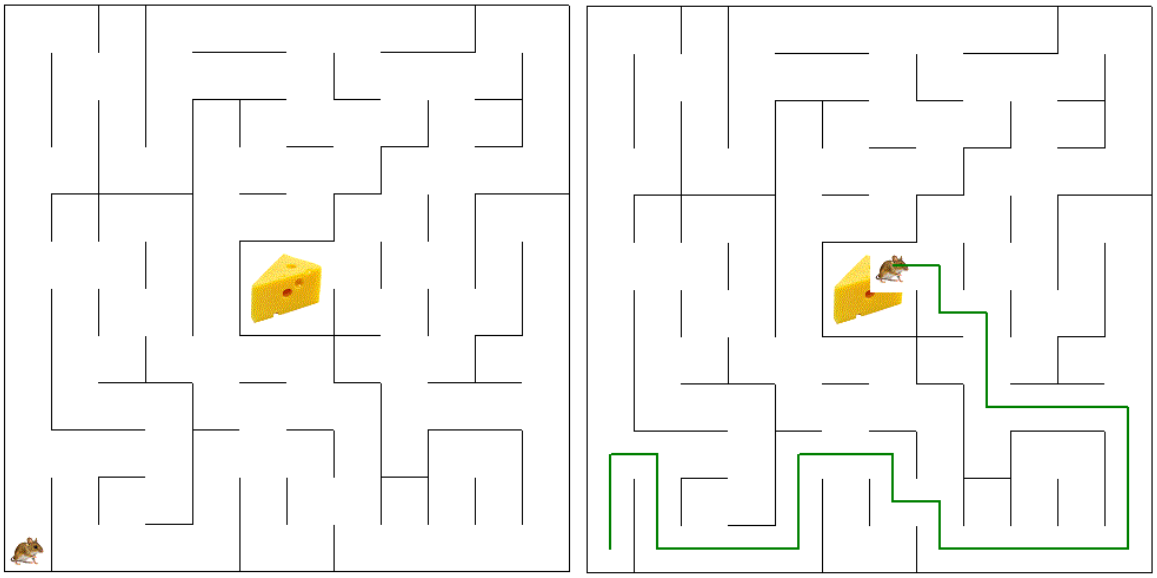
Model Evaluation and Validation

Since the exploration phase uses random algorithm, the probability that the robot will not map the maze perfectly with the best available path every time is high. However, once the maze is mapped, the robot must choose the best path available in the mapped maze. And if there exists two or more paths which require equal steps to the goal, the robot should prefer the path with less turns. The robot can take three straight steps at max, so the path with more turns will require more steps and hence should avoid it. For final evaluation, each algorithm will be run 5 times on each maze and an average score will be calculated.

Optimal Moves:

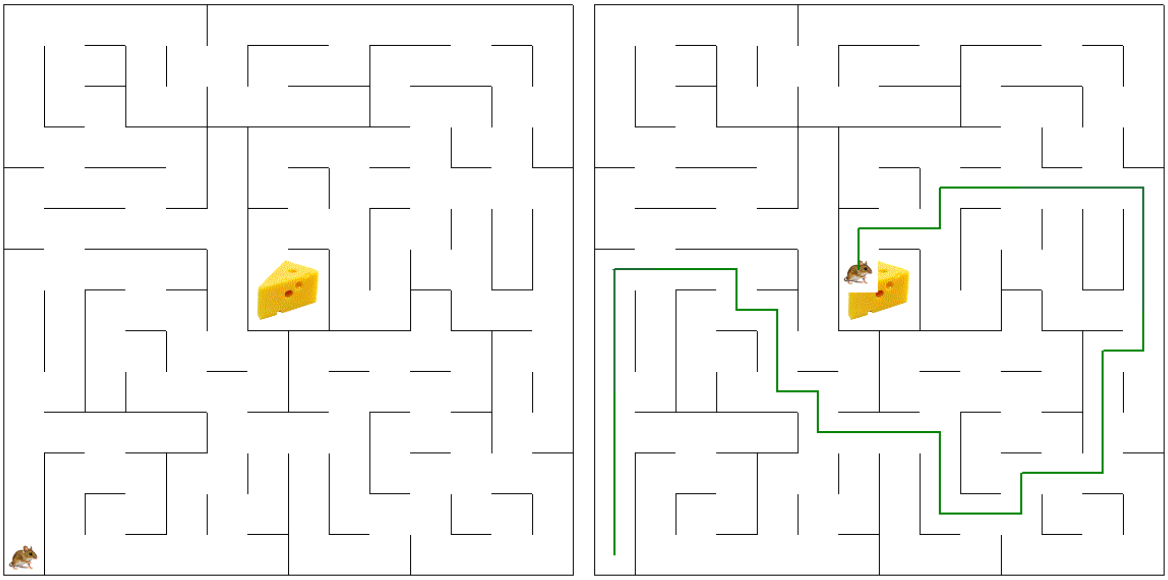
Test maze	Path Length	Time Steps
01	30	17
02	43	23
03	49	25

Maze 01:



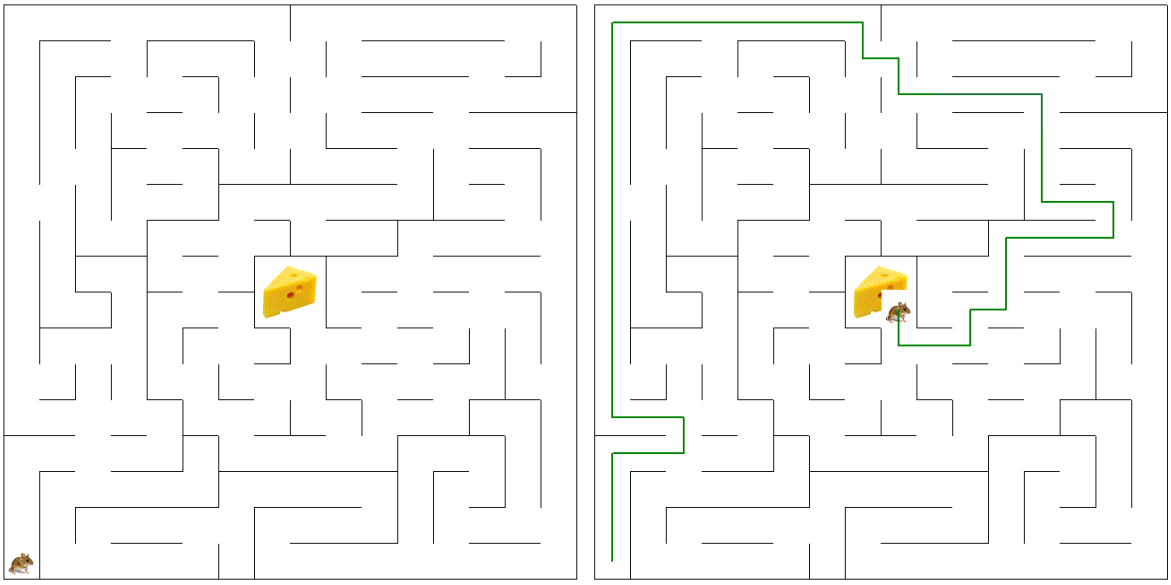
Left: Maze Visualisation | Right: Optimal Path Visualization

Maze 02:



Left: Maze Visualisation | Right: Optimal Path Visualization

Maze 03:



Left: Maze Visualisation | Right: Optimal Path Visualization

Depth First Search:

Maze	Area Explored	Steps Taken		Score	Average Score
		Run 1	Run 2		
1	70.14	212	23	30.067	32.21
	78.47	193	21	27.433	
	95.83	459	19	34.300	
	80.56	302	23	33.067	
	97.22	395	23	36.167	
2	70.41	208	36	42.933	60.00
	85.20	539	53	70.967	
	96.43	458	45	60.267	
	93.88	540	48	66.000	
	82.65	265	51	59.833	
3	70.31	366	39	51.200	65.50
	92.58	644	78	99.467	
	93.75	463	44	59.433	
	79.69	431	45	59.367	
	87.11	361	46	58.033	

The depth first search algorithm seems to be inefficient. The average scores are higher than the benchmark we set. And individual scores vary a lot. Even after mapping a large area of the maze, the robot fails to recognize the best path available. Thus, it is not a good model for our problem.

Dijkstra's Algorithm:

Maze	Area Explored	Steps Taken		Score	Average Score
		Run 1	Run 2		
1	70.14	258	23	31.600	30.79
	95.83	480	17	33.000	
	95.14	438	20	34.600	
	77.01	269	17	25.967	
	87.50	353	17	28.767	
2	72.45	218	27	34.267	38.37
	70.41	226	27	34.533	
	89.8	442	30	44.733	
	79.59	373	25	37.433	
	95.41	416	27	40.867	
3	84.38	421	30	44.033	41.44
	70.31	244	27	35.133	
	82.42	379	28	40.633	
	76.95	335	30	41.167	
	94.53	487	30	46.233	

The dijkstra's algorithm seems to perform better than the depth first search algorithm. However, the average scores are still higher than the benchmark we have set. And individual scores vary quite a bit. Even after mapping a large area of the maze, the robot fails to recognize the best path available for maze 2 and 3. Thus, it is not a good model for our problem.

A* Algorithm:

Maze	Area Explored	Steps Taken		Score	Average Score
		Run 1	Run 2		
1	70.14	151	24	29.033	29.81
	85.42	260	20	28.667	
	89.58	315	20	30.500	
	90.97	203	20	26.767	
	96.53	512	17	34.067	
2	94.90	596	28	47.867	40.21
	70.41	197	27	33.567	
	78.57	228	33	40.600	
	86.22	407	28	41.567	
	84.69	313	27	37.433	
3	70.31	350	27	38.667	40.27
	91.40	393	29	42.100	
	74.22	480	26	42.000	
	80.86	330	28	39.000	
	72.27	288	30	39.600	

The performance of ‘A star’ is similar to dijkstra’s. These results are very near to the expected scores defined in the benchmark. Overall, this is a good model.

Flood Fill / Dynamic Programming:

Maze	Area Explored	Steps Taken		Score	Average Score
		Run 1	Run 2		
1	84.03	266	17	25.867	25.55
	70.14	121	20	24.033	
	88.19	200	20	26.667	
	92.36	247	17	25.233	
	71.53	179	20	25.967	
2	70.41	182	25	31.067	35.67
	84.69	309	28	38.300	
	90.31	433	25	39.433	
	76.02	252	26	34.400	
	72.45	214	28	35.130	
3	70.31	361	25	37.033	38.65
	72.27	340	28	39.333	
	84.38	377	28	40.567	
	79.3	311	28	38.367	
	74.61	298	28	37.933	

Flood fill / Dynamic programming has proven to be the best so far. The results clearly show that it outperforms all other algorithms above. The algorithm not only crossed the benchmark, but attained a really good score as well. Overall, this is the best fit model for all mazes.

Justification

The best performing algorithm is the flood fill / dynamic programming algorithm in terms of the score value. The Benchmark scores for the original 3 test mazes are as follows along with the maze dimensions, path length, optimal moves and the robot's optimized scores.

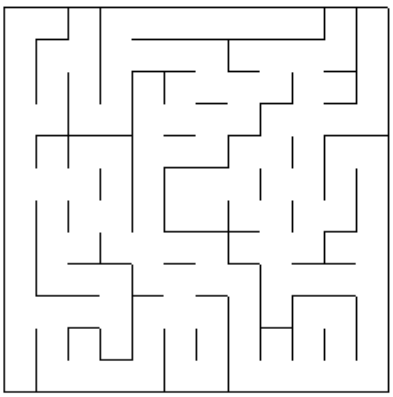
Maze	Dimension	Path Length	Optimal Moves	Benchmark Score	Optimized Score
01	12	30	17	28.32	25.55
02	14	43	23	38.61	35.67
03	16	49	25	43.15	38.65

The final results of the optimized model outperformed the benchmark model, however the model failed to find the optimal solution for Test Maze 2 (23 steps) but the routes the model did find were not too far off.

V. Conclusion

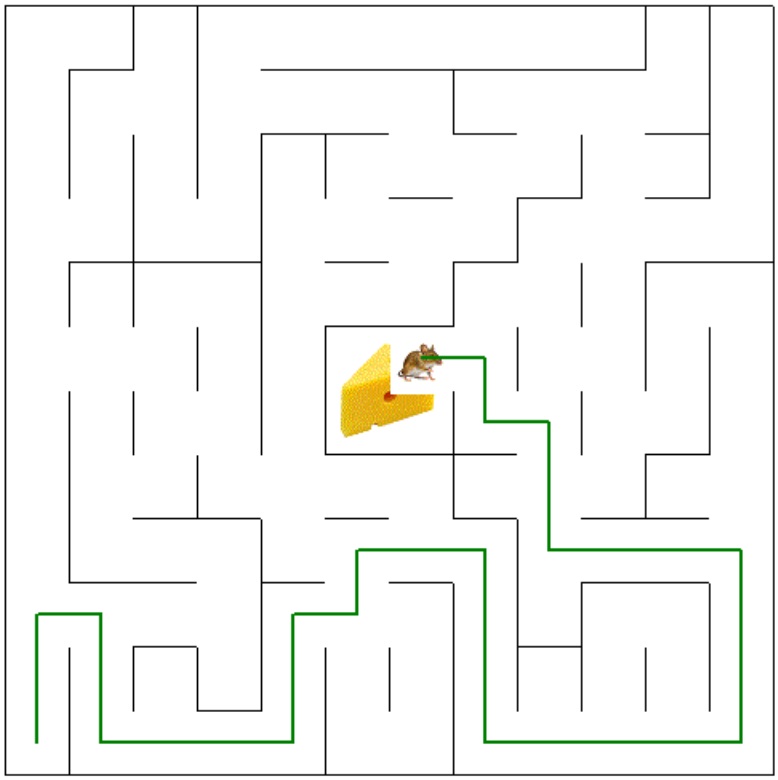
Free Form Visualization

I have created a maze which contains some new examples of dead ends and loops. The text file of the maze has been included in the project files among other three mazes. I have tried to create some deeper dead ends, and some dead ends which constitute a block of cells. A few examples being the top left corner and the top right corner. Examples of loops being the bottom right and bottom center.



The algorithm was able to perform very well in these circumstances too and successfully find the optimal path fairly consistently. One of the best runs is shown below.

Maze	Area Explored	Steps Taken		Score
		Run 1	Run 2	
04	70.14	203	17	23.767



Reflection

I was inspired to take on this project because I have always had an interest in robotics. At its simplest, the problem and solution for this project can be described by the following process:

- Receive a small amount of information about the maze (sensor readings).
- Combine this information with the heading and location of the robot.
- Update the maze map with any new information inferred directly or indirectly from the sensor readings.
- Given the current knowledge of the maze, decide where to travel to maximise the acquisition of new information.
- Decide how to get from the current location to the desired location.
- Move toward the desired location.
- Repeat steps 1-6 until the optimal path from the start to goal locations has been found.
- Start the next run of the maze.
- Follow the optimal path from the start to the goal.

As straightforward as this seems, the actual implementation of the robot that can consistently and successfully complete this process in an efficient manner even when faced with a large number of maze permutations is considerably more difficult. Through trial and error, lots of debugging, and a lot of persistence, the robot was able to efficiently explore the maze and generate an optimal path to the goal area. Despite this difficulty, the algorithms used to map and navigate the maze are quite straight forward. It is interesting to consider how a small number of reasonably

simple algorithms, when combined in a coherent way, can result in quite complex behaviour and manage to solve the problem at hand so well.

Improvement

The problem so far has been has been that of a discrete domain. But if this was a real world robotic mouse competition, then it would lie in the continuous domain. Our model is not equipped to deal with the challenges of continuous domain and needs a lot of adjustments and improvements. All the factors like sensing, rotation, breaking, acceleration, speed, amount of movement will drastically change when we move from a discrete to a continuous domain. If the diagonal movement is also allowed then it can be added to improve the robot's performance. Apart from these needed adjustments, we would also need to apply new algorithms required for the continuous domain. The robot would need to perform SLAM (simultaneous localization and mapping) to explore the maze. Moreover, the robot needs to use PID control to continuously adjust the direction and turns so that it can wander around in the maze without colliding with the walls. The challenges of moving from a discrete to continuous domain are exponential but makes for an interesting problem.

References

Capstone Project Link

https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5IUM5OpAXJYiutFuSljTzV_E-vdE/pub

Micromouse – Wikipedia

https://en.wikipedia.org/wiki/Micromouse

Maze Solving Algorithm – Wikipedia

https://en.wikipedia.org/wiki/Maze_solving_algorithm

Motion Planning Algorithms

https://en.wikipedia.org/wiki/Motion_planning#Algorithms

Breadth-First Search Algorithm

https://en.wikipedia.org/wiki/Breadth-first_search

Depth-First Search Algorithm

https://en.wikipedia.org/wiki/Depth-first_search

Dijkstra's Algorithm

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

A* Search Algorithm

https://en.wikipedia.org/wiki/A*_search_algorithm

Flood Fill Algorithm

https://en.wikipedia.org/wiki/Flood_fill

Udacity’s Artificial Intelligence for Robotics Course

https://in.udacity.com/course/artificial-intelligence-for-robotics--cs373