Plot and Navigate a Virtual Maze

Xiao Wang

July. 26 Wednesday

Abstract

This project is to program a robot mouse that can explore a virtual maze and later find an optimal way moving from the start location to goal area. At first the robot has no knowledge about the maze except for the maze dimension, thus it need to use the first run to explore the maze to increase its knowledge of the maze. In the second run, the robot should follow the instructions made by the navigator based the the maze known so far. Our goal is to let the robot reach the goals use as less moves as possible. So, we can't use too many moves in the first run just to explore all the maze. At same time, we should assure that the maze's explored area is not too low, which will be hard for the navigator to find the true optimal moves.

1 Definition

1.1 Problem Statement

The project contains two main participants, virtual maze and robot. All test mazes are square maze, formed as nxn grid of squares, and n is even. Along the outside perimeter of the grid and some of the edges between two consecutive grid are walls, and the robot can not move through any wall. The goals lie in the middle of the grid, and there exist at least one path from any other location in the maze to the goal area. The actual maze is provided by a text file, with lines of numbers. the number is calculated as $a*1+b*2+c*4+d*8$, a, b, c, d each represents a direction at a specific cell, and the value can be 0 or 1, 0 means that there is a wall in that specfic direction, while 1 is the opposite. Do remember that the maze is totally unknown to the robot except for the dimension at the beginning, robot has to explore and remember the maze by itself.

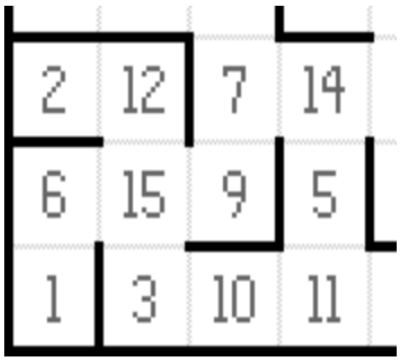


Figure 1: Maze specification, 15 is calculated as 1+2+4+8, and 9 is 1+8, which means the down and right side are walls

The robot can be considered resting in the center of the square at the current located cell, and pointing in one of the cardinal directions of the maze(up, left, right, down). At each location, the robot can sense 3 readings, which are distances to the wall at three different directions relative to the current heading direction. If the robot's heading is left, the readings list contains three readings of the distance to the wall in cardinal direactions(down, left, up). At each timestamp, robot can first make a rotation (or not) and then move in the same direaction up to 3 steps as long as there are no wall in the way. Additional, the robot can also move back up to 3 steps.

Robot can make two runs through the maze, the first run is a exploration run. The robot does't know the maze, so it has to move in the maze and update its knowledge of the maze according to its readings after each time stamp. This is a graph traversal problem, the goal is to explore the graph as much as possible with less moves, and the goal area must be visited. Potential problems we should avoid are deadend path and circle path. The second run can use more advanced methods to find an optimal way because the maze is now known to the robot(hope so). This is a path finding problems, methods we can use including Dijsktra, A* search. The robot is located at the corner at the beginning of each run.

1.2 Scores

At each timestamp, robot can move up to 3 steps according to its controller. In the first run, the robot can continue to explore the maze after it hits the goal area, but it must visit the goal at least once in order to end the first run. The second run has much more weight than the first run steps. The score is calculated as

$$S_1/30 + S2 \tag{1}$$

S_1 is the total steps made by the robot in the first run and S_2 is steps made in the second run. And the total time of the robot to finish one trial, a complete first, second run and reach the goal area, is 1000 steps.

2 Analysis

2.1 Data Exploration and visulization

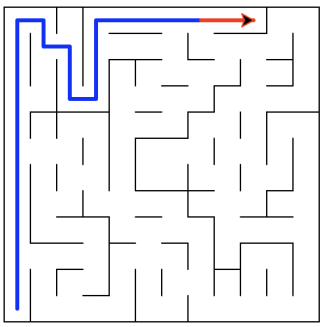


Figure 2: Maze deadend

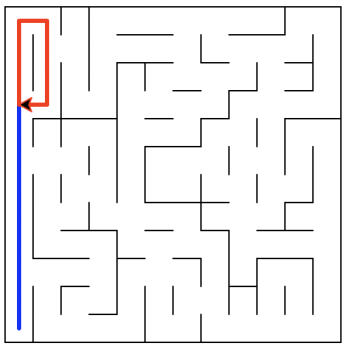


Figure 3: Maze loop

The robot knows its location, heading at each timestamp. Also it has three sensors, the first one is the distance to the nearest wall to the left of the current heading of this robot, the second one is the distance to the nearest wall to the front the current heading and the last one is to the right of the robot. So hopefully, robot can gain more knowledge than just the current cell of the maze each time and it need to store and update the maze knowledge accordingly. The robot can not move through any wall. Additionally, we assume

that sensor readings and movement are perfect, there is no randomness. The start location is [0,0], and the goal area lies in the center of the maze.

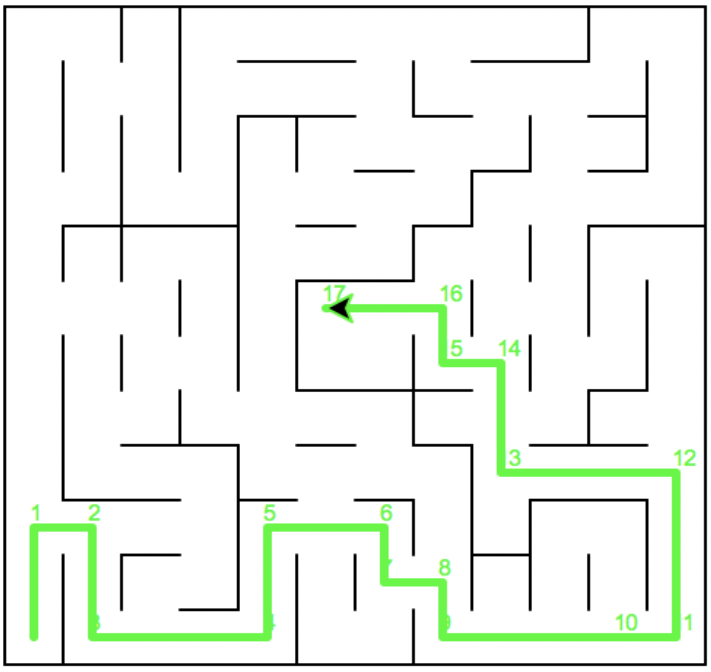
There are some interesting structures need our attention.

- deadend: cells where robot can only get out by going back, showing in red in Figure 2. It is useless for robot to visit deadends too many times, so we need figure out a way to prevent robot going there once deadends are detected.
- loop: robot can fall into the same path again and again within the loop if robot doesn't have a good navigator. One simple loop is displayed as red path in Figure 3.

.

In all mazes provided, every cell has a way leads to the goal area, which means that this maze is a fully connected graph, thus we can use graph traversal algorithms to solve the problem. Robot's sensor reading can contains many information, like at the start position of the first maze, we can know that every down side of the leftmost cell is not a wall, so we must take advantage of the readings to make less move.

Figure 4: Maze 1 with red line indicating the path



One possible move path from start location to the goal for the first maze is shown in the picture below, the robot need 17 movements to get to the goal and the path's actual length is 30.

2.2 Benchmark

For the benchmark, we need to define some kind of reasonable score for each maze that we can compare our results to. The score contains 2 parts, the exploration and the optimal move. Given a valid maze, the optimal moves is fixed if the maze is toally known to the robot, like in the first maze, the optimal move is 17.

But, we can't make sure how many steps we will take in the first run to explore the maze. We can assume the maze is a perfect one, so it takes $dim * dim$ steps to visit every cell within this maze. We can treat this as a lower benchmark(actual score can be smaller than this). So for the benchmarks, we have:

- Maze 1: $12 * 12/30 + 17 = 21.8$
- Maze 2: $14 * 14/30 + 22 = 28.53$
- Maze 3: $16 * 16/30 + 25 = 33.53$

3 Methodoloy

3.1 Data Preprocessing

The maze provided is accurate and the sensors readings are also correct, we don't need to worry about the randomness and possible error in the environment, so there is no need to do the data preprocessing.

3.2 Proposed Algorithms

For path finding problems, there are several algorithms we can consider:

- Dijkstra Algorithm: This is an algorithm for finding the shorest paths between nodes in a graph. It works by visiting every cell or node in the graph starting from the start location. Each step, updating distance information for nodes not marked as visited and choose the closest node, which has the lowest distance, marked as new current node. The original node is marked as visited and will not be visited again. In our case, the cost for each move is just 1, so it is guaranteed that we can find the shortest path from the start location to the goal area.
- Best First Search: This is a greedy algorithm. In each time step, instead of choosing the node with lowest distance to the start location, we choose the node with closest to the goal area. This method may not find the optimal solution, but it act faster than Dijkstra algorithm, because it know where to move instead of going to goal area by 'chance'.
- A*: This method is some kind of combination of Dijkstra and BFS, for the choice of the next step to take, we will consider both the length traveled so far and the heuristic value to the goal location. Heuristic is pretty import, a good heuristic can imporve the time greatly. For the maze case, there is a safe and simple heuristic, just the manhattan distance.
- Q-learning: This is a leraning method that is a little bit different from previous methods, robot tries to learn the optimal policy through its interaction with environment, robot learn from the history of state, action, reward $< s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_n, a_n, r_n \cdots >$, and try to update socalled Q value as

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma maxQ(s_{t+1}, a) - Q(s_t, a_t)) \tag{2}$$

  This method need not remember or rebuild the entire maze, but it can eventully know what to do at each state with the knowledge of location and sensor readings. But this may not be very helpful in this problem, we can move at most 1000 times, it is not enough for the optimization.

Many of the path algorithms require the full knowledge of the maze at first, and most of them are not suitable for exploring the maze. So we may need other methods for exploration. The simplest one is just random exploration and choose to stop after we gain about 70-80 percent knowledge out of the entire maze. Obviously, this does not work well, the robot will go the deadend path many times and it has to spent more time to get out of the deadend. The worst situation is loop path, the robot can't get out of the loop once it entered there. So we must add some variaitons to the random version. First, we can detect the deadend and try to avoid visiting there once the robot know where are deadends. To avoid infinite loop path, we can make

the robot choose to visit the less visted locations. This can alleviate the problem of random exploration, but this method is very inefficient, the robot can move zig-zagly and not visit some location for a long time.

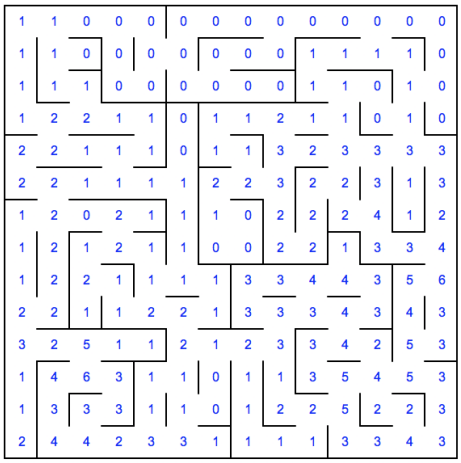
The goal of the exploration is to know as much as possible about this maze. In other words, we want each robot's move decrease the 'entrophy'(uncertainty) as much as possible. What's more, we know that the sensor reading is not limited to this location , it can also provide information of other cells in the same row or column. So, each timestamp, we can update multiple cells' uncertaity. At each timestamp, the robot will choose to visit the location with the largest uncertainty(assume unknown edge as open). Because, some locations in the planned path may be blocked, which we didn't know at the time the robot plan the path, so the every time encouter wall, the robot will choose another target location to move to. In this way, robot is guided toward the location which can maximize the knowledge of the maze by visiting the location, the maze can be explored very quickly.

3.3 Implementation

3.3.1 Mapper

The Mapper class is used for storing the knowledge of the maze discovered so far. In my implementation, I used a 3 dimensioal list to store the wall information for each cell. For each location (i,j), it also has four elements indicate the status of 4 directions, in the order of "left,up,roght,down". There are three possible values for each element, -1 for unknown status, 0 for open edge and 1 for wall. Mapper can update many cells' information based on three sensor readings. Moreover, sensor reading contains not only the local information of the current cell, if the sensor reading is larger than 1, than we can also update other cells in that direction. One thing to remember is that, be careful to uodate the wall information simutinously for consecutive cells, for example if the current cell's left side edge is wall, the left side edge of the cell to the right of the current cell should also been set to wall.

At the beginning, I also implemented two seperate list with the maze to help exploret the map, one is called deadend, used to indicate which cell is deadend, the robot will avoid going to these locations again. And also a simple counter map to store the number of visits to the same location, this can be useful to guide the robot to choose to explore the cell that is less visited, thus can avoid infinite loop problem. In Figure 5, you can see the number each cell has been visited, the upper maze hasn't been visited, and the most frequent visited cell has been visited 6 times. Figure 6 display the deadend discovered by the first run, 1 indicates deadend.



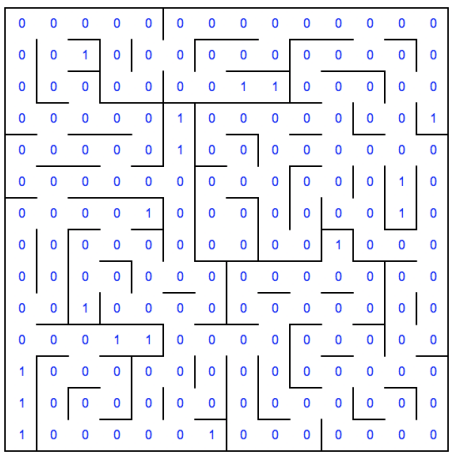


Figure 5: Counter Map for maze2 using counter searchFigure 6: Deadend map for maze2 using counter search

But later, I found that just move to the cell according to the counter map is too restrictive, it only use the local information, which means the robot may spend a lot of time getting rid of local traps. The better idea is to divide exploration problem to several sub-problems, the robot can plan to a target location which need to explore the most. The way we define the urgency of target locations is use uncertainties. For each cell, the uncertainty is just the count of unknown edges of that cell, for example, if a cell all four sides status are unknown, the uncertainty is simple 4. The edge cells have uncertainty of 3 because one side is edge and 4 corners has onuncertainty of 2. The uncertainty should been updated at each timestamp in the similar manner of wall knowledge update. Some small tricks can be used here, if 3 sides are known as wall and the other one can be inferred to be open.

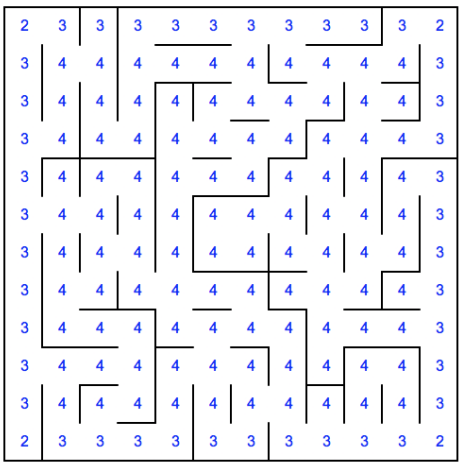


Figure 7: Uncertainty Map at beginning

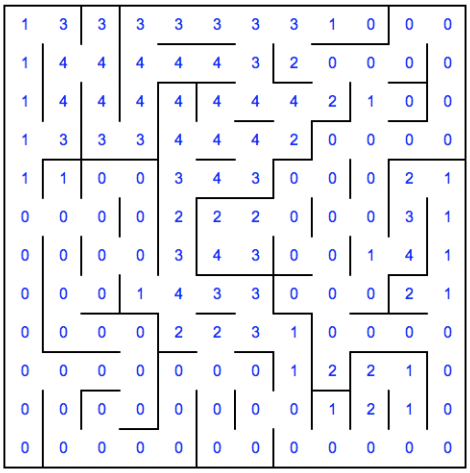


Figure 8: Uncertainty Map after 60 steps

The mapper also has util methods like draw the maze and display current state of the maze knowledge.

3.3.2 Navigator

This class helps robot decide which way to goal at the current location based on the mapper known so far. At first, I used the method named counter serach, it will choose the most least visited cell that is not deadend and adjacent to current location. The problem of this method has been mentioned above, it will spend many time locally without touch cells that are more important in a global sense.

The method I found more stable is called target search. If the robot is in the first stage, the exploration phase, robot will solve many small subproblems to gain knowledge of the maze. Robot needs to explore the map as much as possible, in other words, robot needs to visit cells that has the largest uncertainty(defined previously). In each timestamp, the navigator will give the robot a target location with the largest uncertainty, if there are multiple cells, then we will choose the closest one. Once the target location is given, the problem is just finding the shortest path from current location to target location, I used Dijkstra algorithm here to find the shortest path. For unknown edges cells or edges, we treated them as open while calculating the path. The shortest is path calculated is not exactly the move we should take because the robot can move up 3 steps per time, so we should also tranfer the path to actually moves. One thing to think of is whether the moves transferred from the shortest path is also the shortest path we can make, there may be possible that a longer path can have shorter moves. Moreover, the moves calculated is based on the assumption that all unknown edges are open, so we will stop carrying on the moves calculated before if the robot encounter any walls and repeatedly to find a new target location to go.

If the maze is totally known, we can use A* search for finding the shortest path from start location to

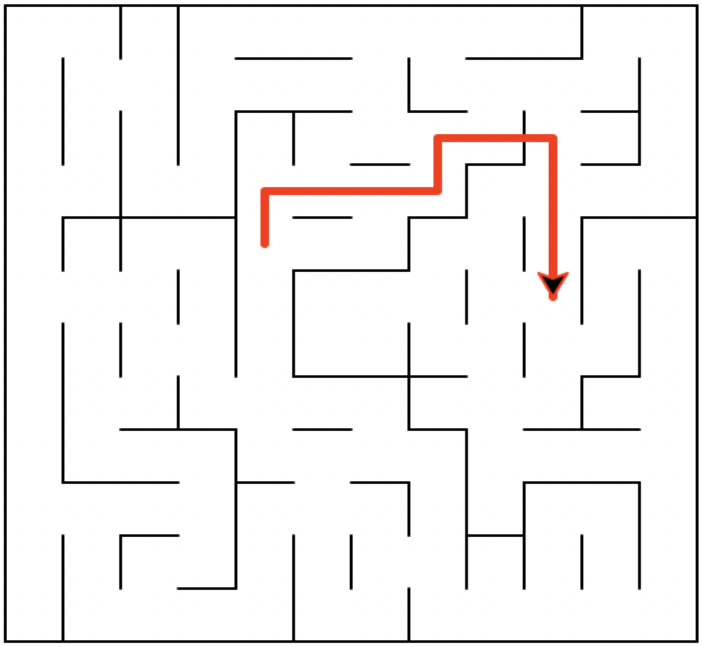


Figure 9: Target serach process, the current location is [5,8], the calculated target is [10,7], and red is the shortest path, assumed the unknown edge as open, so you can see the path go through the wall, that is because that wall is not yet discovered.

the goal and transfer the path to actual moves. But when can we say the maze is explored enough for the second run? We don't need to explore the maze to 100 percent actually, that way robot waste a lot of time on visiting un-informative cells. Once any of goal locations is visited at least once, we can calculate two shortest paths, one assume all remainning unexplored cells or edges are closed, we can find at least one path because the goal has already been visited or known, we call this close-path. Another one assumes that all unknown edges are open,we call this open-path. It is easy to prove that open-path is always better or equals to close-path, for the reason that the close-path is contained within all possible candidate paths for open-path. And for the reason that A* search can always find the optimal path given the map, so once the close-path is as good as open-path, we can no longer find any better optimal path, which means the optimal path has been found. One concern is that, the optimal path is optimal in the sense of path length, it is not guaranteed the moves got from the paths is also optimal, so we should take care of this while tranferring the path to actual moves.A simple way to partly solve this problem is start from the goal location and move back to start location. After the A* start search, using Manhattan distance as heuristic, we can get a value map for the maze we visited in the process of searching. We can use the map to get many candidate paths from goal to start, and then transfer them into actual moves and keep the one with minimum moves.

Another thing we should pay attention to is that the goal location must be visited at the first run, it is possible for the robot not visiting any of the locations but can still find the optimal moves based on the method mentioned above, so we should let the robot goes to any of the goal location if the goal is never actually visited even the optimal move has been found.

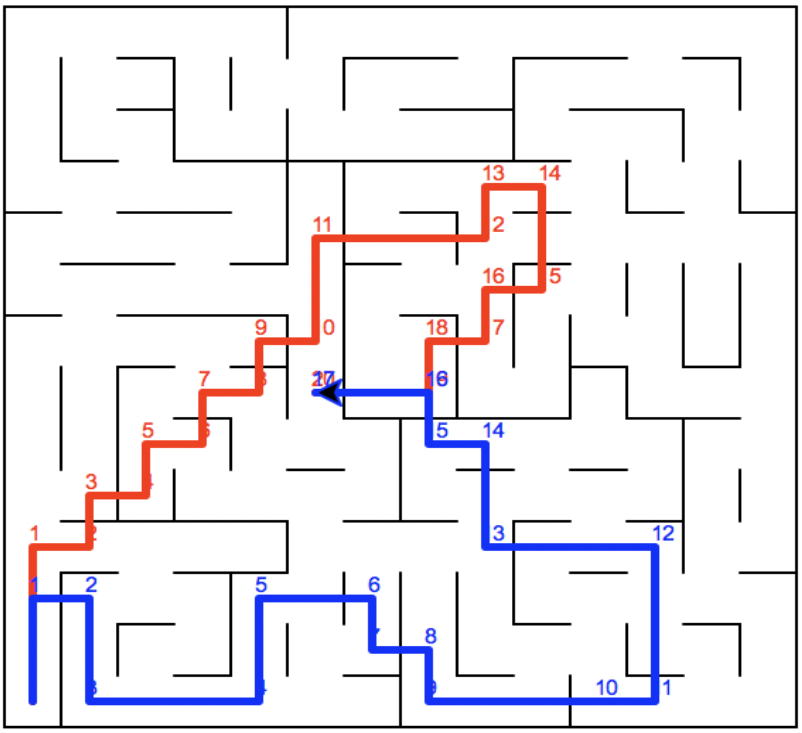


Figure 10: Compare of close and open pat, the red one is open path, path length is 29 and blue path with length 30, but red path is infeasibl

3.3.3 Robot

Based on the provided skeleton, used mapper to store maze information and use navigator to figure the next step to move. I also update the location and heading of the robot accordingly. The navigator will return a flag 'Reset' to indicate the end of first run and then the next move calculated by navigator is just carry out the opitmal paths' movement at each timestamp.

3.3.4 Utils

I created a util PriorityQueue class used to implenmented the A* search algorithm.

3.4 Results

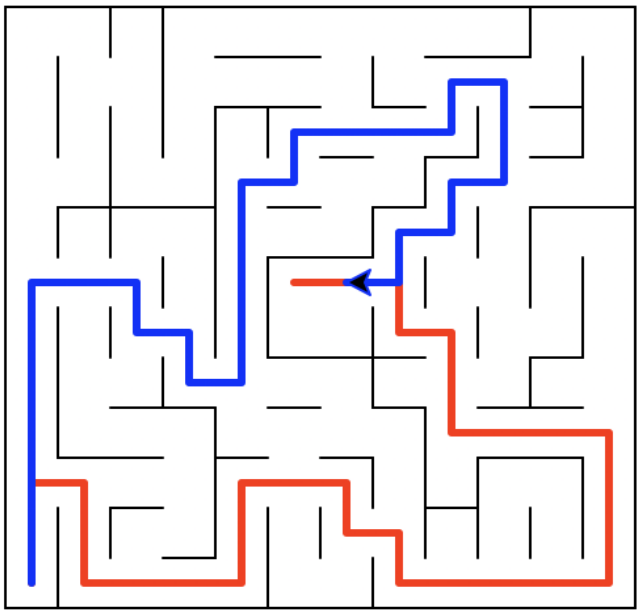
For couter_search, there is another parameter we can control, the area already visited, the search will continue even the goal has already been visited until the threshold has been met in the first run. I used 2 threshold to test, 50% and 75%. Counter_serach is a stochastic method, so I run the method for each maze 5 times to calculate the average. While the target search is a deterministic method, we will get the same result for every run. In general, target_search is better tha counter search, because it avoid many redundatnt moves in the first run by going to the most uncertain location. Coutner_serach can get lower score than target_serach, but most of the time it spent more steps. Maze1' best move is 17 steps, the target search can get score of 20.833. The couter search with lower threshold can achieve average score of 22.11, which is not too bad, but it can not always get the best move, which means it stoped early sometime that the robot can't find the opitmal move. The 75% threshold has worse score of 24.3198, while it can find the optimal moves.

Table 1: Scores

method	move length	average score
counter_search (at least 50%)	19.4	22.11
counter_search (at least 75%)	17	24.3198
target_search	17	20.833

So for this map, extral exploration for the maze is not worth for getting a lower score. Another thing to

Figure 11: Result for maze 1



notice is that, based on my implementation, the robot can only get optimal path first and then transfer it into moves. But two path with same path length are propably to have different moves length. You can see in the figure 1, blue and red path all have path length 30, but it only 17 steps for red path which it needs 19 steps tp finish the blue path.

Table 2: Scores

method	move length	average score
counter_search (at least 50%)	22.4	31.8534
counter_search (at least 75%)	23.8	38.567
target_search	23	27.967

Similar result for maze2, target search can get the 27.967 while target search's average is above 30. Some of targe_search run can get a score lower than 27 but it is really rare.

Figure 12: Maze 2 target serach

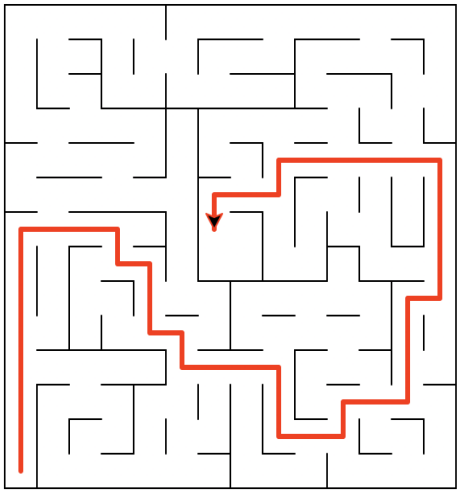
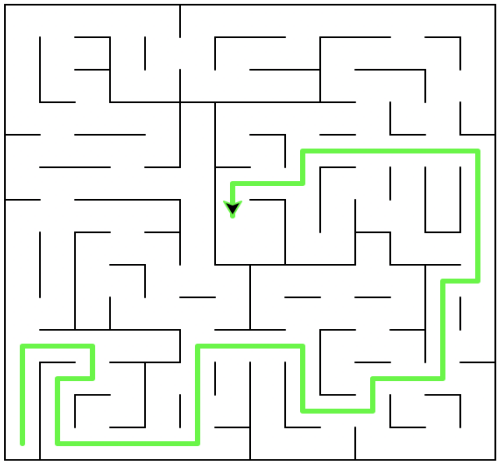


Figure 13: Maze 2 counter search best



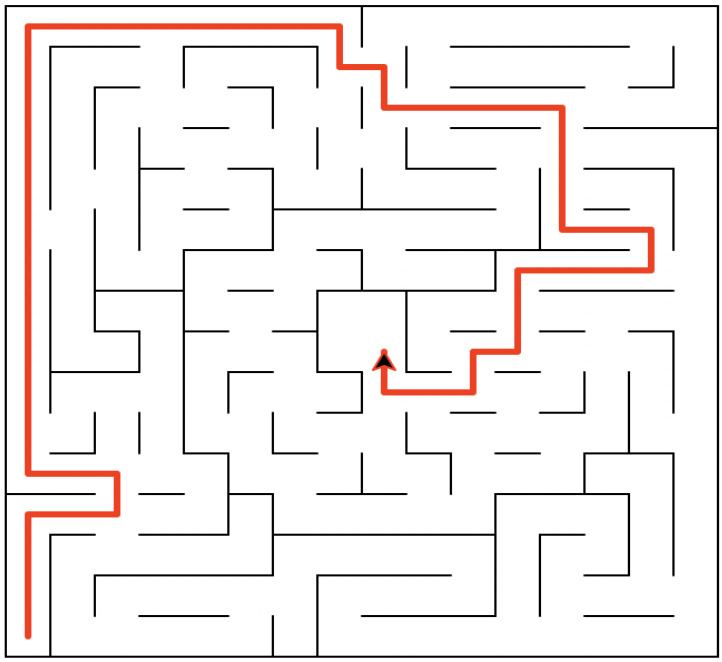
A proof for the optimal path and optimal move relationship, the target search can get the best path of length 43, while best target search move can have a length of 44. But the move length is 23 compared to 22, target serach is the best. As you can see in Figure 12 and Figure 12, target search's reuslt path is shoreter but its movement need one more step then the coutner search.

Table 3: Scores

method	path length	average score
counter_search (at least 50%)	25.2	39.566
counter_search (at least 75%)	25.4	37.84
target_search	25	32.633

Very similar result for maze3. For complex maze, the threshold for counter search is not too important for threshold not too extreme, because the maze will be discovered for a great portion before the robot get to the goal. Also, with more complex maze, the target search has more advantage over counter search.

Figure 14: Result for maze 3



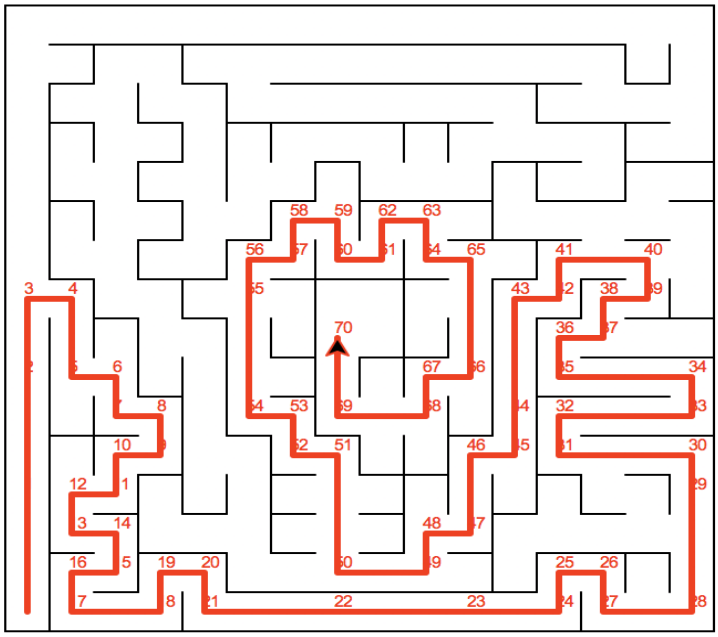
4 Free-Form Visuliazation

I am using the maze from 2016 mircomouse worldchampionship. This maze has many deadends and many turns, start location and goal area are the same. The target_search algorithm can easyly get to the goal area and achieve the goal within 1000 steps.

Table 4: Scores

method	path length	average score
target_search	70	83.967

Figure 15: Free-Form map from micromouse, gree line display the fastest path



5 Reflection

I always have high interest in robot and related areas, so I decided to choose this robot navigation project as my capstone project for the MLND and I really enjoy the process. First, we need to get several candidates algorithms for the second run, which means we have maze known for most of the maze. There are many algorithms we can use, DFS, BFS, Dijkstra, A* and reinforcement learning, we can verify these methods using the existing maze. And the A* search seems to be the most easy to implenment and very robust and quick.

So the main problems is to explore the maze. The first question to ask is how we store the maze, the information we need is the edges information of each cell, so we can use a list to indicate status of four sides of the cell. The information we can get at each timestamp are location, heading of the robot and 3 sensor

readings, we can use the readings to update many cells edge information and update those information in the mapper class.

Have a way to represent the maze within the robot, next we need to figure our a way to explore the maze as much as possible while use lesser steps to achieve such goal. The first idea is easy, just use some extral map to store some information to avoid entering deadend and trapped in infinite loop. This implementation is local, because each time robot only consider the cells nearby and doesn't have a view for distant locations, which may be much more important for the exploration.

Given 2 locations, the maze will been explored more if the robot choose to the location that has larger uncertainty, thus I created another map to store all the uncertainty's information. For the robot, each time we are not only make the decision based on local information, we are using the global information and the exploration has been divided into several small path finding problem. This way, robot can avoid making useless move because it has some guide to achieve the goal quicker.

Then we need to find a path from current location to next target location. Dijstra algorithm is an efficient way to find the shortest path from one point to another in a graph. To use this, we are need to assume the known cells to be open, because Dijkstra need the graph to be a connected one. The path got from the method is based on one move per time patter, we need to transfer it into actual moves the robot can make. And also be aware of that the robot not moving to wall because the path is got based the open edge assumption, which is not often the case.

The remaining problem is when to stop the exploration, this is easy, we need to compare two possible paths based on open and close edges assumption seperately, and the open assumption path is always better than the close one, the time we get this two paths' length equal, then we find the optimal path. Because the path we found is based on 1 move per step, it is possible that the optimal path doesn't correspond to an optimal move.

6 Improvement

This project is interesting and yet not too difficult, we have very simple and regular square map, walls have no thickness and robot is described as a dot. Moreover, robot's movement is perfect that at each time step the rotation and move can be carried out without any error. The environment is static without any noise and robot's sensor reading is also perfect is perfect. All these assumptions are unreasonable in real life situation.

Having the wall with large thickness and robot as a circle, if all other conditions not change, for example, the robot still move 1 per timestamp, the data will be affected is the sensor readings. We can transfer the new readings to original format and still use the same code and algorithm. But if the noise is introduced into robot's movement or sensors, we can't use the deterministic alogrihtm any more. The probabilistic model should then be introduced that robot is never sure where it is, it can only be confident where it is. When the map is known, we can use Kalman filter to enable the robot know the its current location. The kalman filter assume gaussian noise in measuremnt or movement, and it updates its confidence of the location with each or several sensor readings using some gaussian distribution calculation. But Kalman filter is only unimodal, it is not able to track multiple modals which may incurs that it can lose some possible locations, which is often the case in a environment with many similar locations(just like the maze here).

Thus we can use a simpler method to achieve the same goal, the Particle filter. Particle filter has the same bayesian inference logic but uses small particles to represent the distribution of confidence of the possible locations of the robot instead of gaussian distribution. But the problem is we don't know the map either, so SLAM can be used here. SLAM enable robot to simultaneously build the map and yet can localize it self in the unknown map.

For real robot, the physical move of the robot also need to be considered. A simple PID controller can be used here to control the robot to maintain desired heading and speed. Thus providing a reliable foundations for the robot's navigator.

References

[1] Implementation notes *http://theory.stanford.edu/ amitp/GameProgramming/ImplementationNotes.html*. 2007.

[2] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.