Machine Learning Capstone-- Robotic Motion Planning

Long Wan

1. Definition

1.1. Domain Background

This project was inspired from Micromouse competitions, which became popular in late 1970s. In this competition, there was a maze with 16 by 16 grid of cells, where a mice, actually a robot, tried to find the fastest way from its original place to the central point of the maze. The robot would collect data throughout a number of trials and attempt to reach the target with what it had learned. In this process, various searching algorithm might be used to train the robot. Previous algorithms used includes Bellman Flood-fill method, Dijkstra's algorithm and many others regarding tree traversal.

The Micromouse issue could be regarded as the miniature of self-driving system, to find the optimal path, to avoid collision and to learn based on data collected. So to try designing the method to help the mice find its way is of great importance to start a self-driving career. For me, I will strive to accumulate hands-on experience of designing algorithm and data processing, and further strengthen my understanding of machine learning and robotics.

1.2. Problem Statement

Robot is expected to start from the bottom-left corner of the maze, and is supposed to move to the center of the maze. We should find the best algorithm for the robot to learn and find the optimal way to reach the goal.

The maze designed in the project will be in an n*n dimension, where n is even, so that the center of the maze could be a 2*2 enclosed area with only one open exit. Walls in the maze can block the way the robot heads so the robot should change their direction to avoid colliding with walls. The starting cell is enclosed by 3 walls, therefore, the robot can only move to one direction (upward here) at the beginning. The entire maze is fully enclosed and the robot will not go outside the maze. Mazes are provided via text file in the format of 4 digit binary number, representing the existence of walls from 4 directions, upwards, right, bottom and left side, respectively. For example, 0101, which equals to 5,

means that there are walls on the right hand and left hand side, and ways are open upwards and downwards.

The robot is designed to perfectly take up only one cell. There are 3 sensors embedded inside the robot indicating the number of open cells in front of the left, right and front side of the robot. For each time, result of sensors will tell the robot opened ways, and then the robot will choose to turn left or right if there is a wall in front of it, or move forward if there are walls on the two sides. Each movement could range from 1 to 3 between two sensor readings. Robot might turn exactly 90 degree clockwise or counterclockwise when walls block it. If the robot tries to go into the wall, it will just stay at the same location. The robot will start by exploring the maze. It will travel to somewhere and collect information regarding the maze, especially marking spaces too far away from the goal.

So generally, the robot will receive 3 numbers from sensors and pass them along to the "next_move" function. The "next_move" function returns two values, rotation(-90, 0, 90 representing turning left, forward and right) and movements(-3 to 3, where negative values represent backward and positive values represent forward). Once the robot enters into the center area, it will be regarded as a successful run. We should design an algorithm to have it reach the goal in the shortest way. Potential algorithms have been covered in Domain Background above.

1.3. Evaluation Metrics

There are two important steps for this project, exploration and optimization, and weights of them are different on the measurement score. Weight for exploration runs should be much lower than following optimization runs. So evaluation metric is the aggregate value of following scores,

1. The total number of steps taken in exploration process divided by 30. The exploration process might be run again if the robot did not find the target within the upper limits of steps, like 500, or 1000.

2. The total number of steps takin in optimization processes. This process will not stop until the robot thinks it finds the optimal path and reach the goal.

The lower the score, the better.

2. Analysis

2.1. Datasets and visualization

The nature of the project does not require any data inputs, but in the process of learning and optimization, some important data are collected and updated.

Input datasets:

Location of robot: normally an element of an array, like [0,0] representing the bottom-left cell.

Heading direction of robot: one of four directions, up, down, left and right, representing to which direction the robot heads at a specific cell.

Distance to wall for each cell: a list with three values representing steps to the nearest wall for each direction except backward.

The shortest steps from current location to starting point: this is an array, where each cell has a score recording its minimum number of steps to starting point.

The Manhattan distance from target area to current location: this is an array, where each cell has a score recording its Manhattan distance to central area.

Maze dimension: the length and width of the maze(square).

Output datasets:

Movement: the number of steps the robot should go

Rotation: which direction should the robot turn given heading direction and current location.

Shortest path: the shortest path from starting point to the target area

Map: an array with scores in each cell. In the A* algorithm, the score is the aggregation of the shortest steps to starting point and Manhattan distance to target area. In the Dijkstra algorithm, the score is only the shortest steps which might include consecutive steps into consideration. Cells that were not reached by A* algorithm are marked as 1000, which will also never be reached in Dijkstra algorithm.

Mazes:

Three mazes of different dimensions are showed below.

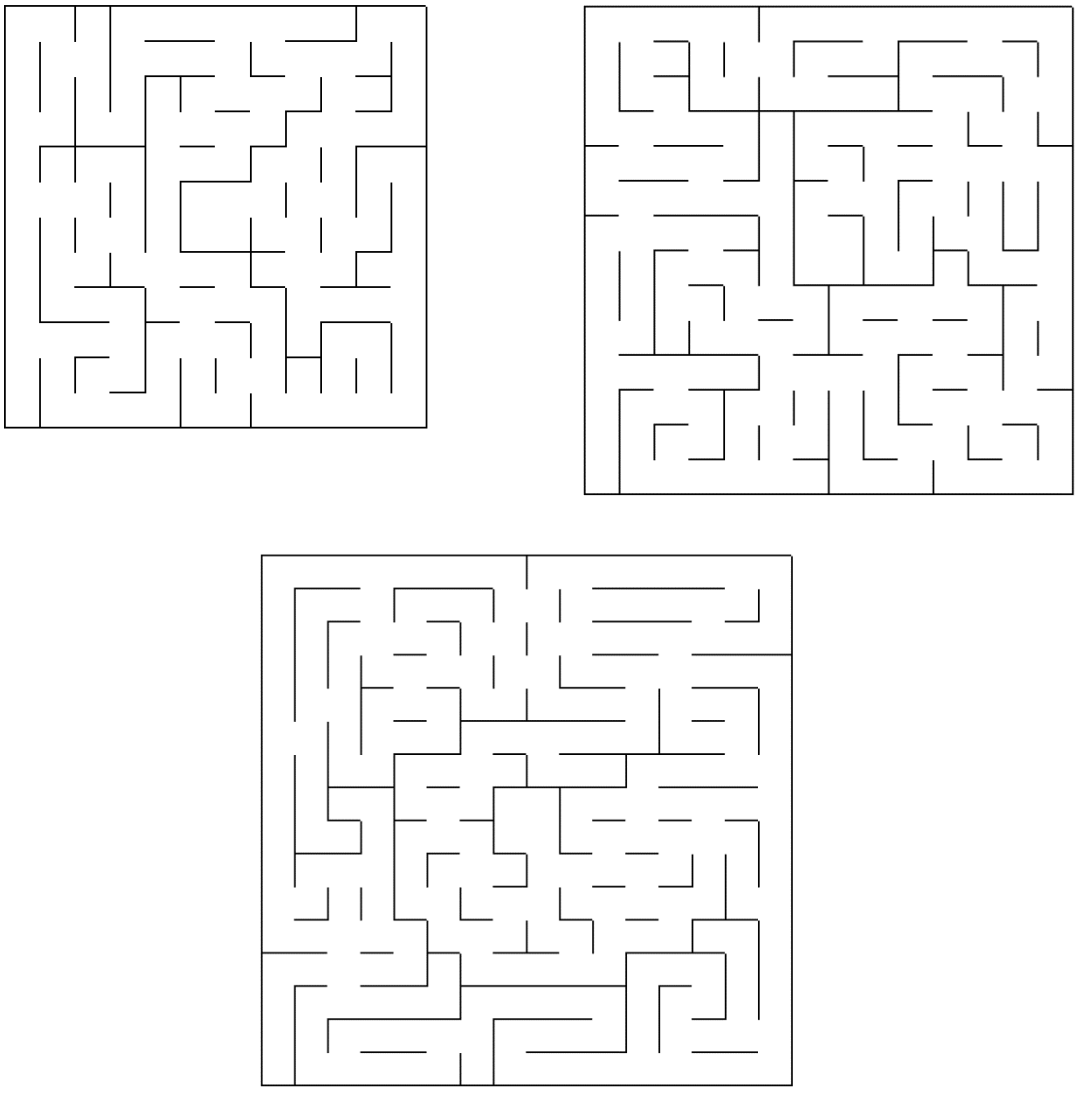

*Figure 1 Graphs of three mazes. Top left is the 12*12 maze. Top right is the 14*14 maze. Bottom is the 16*16 maze.*

2.2. Algorithms and Techniques

As stated in previous parts, the project contains two steps, exploration and optimization. There are two algorithms designed correspondingly for these two steps.

For exploration step, A* algorithm is used. The reason why I use A* algorithm is that the distance from the robot place to target area are taken into consideration in the exploration process so that the robot would not deviate from its “correct” path too much. It could save exploration time and help the next run ignore those unnecessary parts too far away from its target.

Generally A* algorithm assigns each box a score, let say, it is T(x), where x is the location, and it follows the formula below,

$T(x) = G(x) + H(x)$,

where G(x) is the shortest steps from starting point to the location, and H(x) is the distance from location to target are. There are three optional choices for calculating H(x), Euclidean distance, Chebyshev distance and Manhattan distance. In this case, I selected Manhattan distance since the robot is just moving like cars moving in cities.

As the robot explored the maze, a cell got a score T(x). Exploration stopped when the robot reached the goal. Cells that have never been reached got a score of 1000, while cells that have been reached got much lower score

After the exploration process, Dilkstra algorithm was used in determining the shortest path. The only difference between Dilkstra algorithm and A* algorithm is that Dilkstra only takes the shortest path from starting point to current location into consideration but leave Manhattan distances to target area out. The reason why I use Dilkstra algorithm in this step is that, we only care about the speed to reach the goal given the score map generated by the exploration step. The exploration step has limited the search area, and Dilkstra is used to find the shortest path.

2.3. Benchmark Model

The benchmark model I choose is the random model. The robot will avoid running into a wall according to sensors information but it chooses direction and movement steps randomly if necessary.

In the exploration run, the random model marks dead ends. Cells with walls at three direction and cells with only two accessible directions connected to them are all marked as 1000. If the robot runs into dead end, it will move backward until it gets out of it and mark them as previously stated. So in the second run, the robot will avoid running into these cells. Samples dead ends are listed as follows.

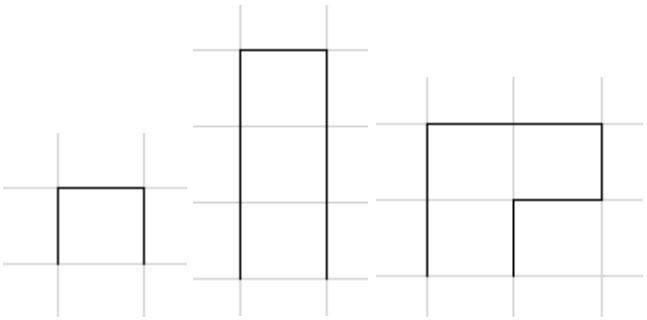

Figure 2 Some sample dead ends

The benchmark model can be evaluated by the metrics. Once the robot find the target for the first time, it will then move forward to optimization process. We can set an upper limit of attempts to let it stop.

3. Methodology

3.1. Prepocessing

The nature of this project does not require any data preprocessing.

3.2. Implementation

The project will be built under Python 2.7 with Numpy package. There are several starter codes provided here in an archive:

https://drive.google.com/file/d/0B9Yf01UaIbUgQ2tjRHhKZGlHSzQ/view

Starter codes include:

Robot.py: py script that establishes the robot class.

Maze.py: script that constructs the maze and checks for walls upon robot movement or sensing

Tester.py: script that test robot's performance on navigating mazes.

Showmaze.py: plot a map of the maze

Test_maze_##.txt: mazes of different dimensions to test robot.

In the implementation process, I mainly thought about following problems.

- What is the general logic of the project

- What kind of information should be stored in the exploration step in order to maximize efficiency for the second run and how they are stored
- Realization of A* star algorithm and Dijkstra's Algorithm
- When should the first run stopped
- How to reconstruct path and determine rotation and movement according the path
- When encountering bugs, what could I do to fix them

What is the general logic of the project

There are two steps, exploration and optimization. For each step, the general idea is similar.

For the step of exploration, the robot will start off without any knowledge towards the maze. It tries to reach the goal according to sensors reading by traversing a great number of locations with A* algorithm. After reaching the goal, a score map would be gained, and an optimal path would be reconstructed. Then the next_step function would generated rotation and movement based on the optimal path, heading direction, and its location. Notice that movements are always 1 in this process, since when exploring, the robot should navigate carefully and look through all neighbors to find something. If it moves two or three cells at one time, it would definitely miss something.

Then it moves forward to optimization process. At this time, the robot got a map from the exploration process, which might be smaller than the original maze since the exploration process has already blocked cells too far away from target area. So given a smaller but more precise map, the robot could move more aggressively. It move at most 3 steps for one time as long as there are no walls on the way. Dilkstra's Algorithm is applied to find the optimal path. After having the optimal path, the next_move does the same thing to calculate rotation and movements.

What kind of information should be stored in the exploration step in order to maximize efficiency for the second run and how they are stored

The starting code has already mentioned several inputs that should be included, the location, heading direction, and the maze dimension. Meanwhile, sensors reading have also been included into the next_move function. They are basic information that we need for sure. But when I tried starting coding with them, I found they are not enough.

First of all, a map with scores generated by Dijkstra's and A* algorithm is needed. This map was initialed as a numpy array with exact the same dimension as the maze has, and filled with 1000 for each cell. The algorithm traversed most cells and each time it reached a cell, the cell was assigned a score calculated as I stated in Algorithms and Technology

section. Cells whose children have all been traversed would be placed into close set, meaning that scores in these cells were determined and would not change again. Cells that have already been reached but whose children have not been completely traversed were placed into active set. Children of cells in the active set would be traversed one by one ordered by the current score the cell has, and cells score in active set might change based on future exploration until it was placed into close set. So in this case, maps stored values which are extremely important to search out the optimal path.

The second variable I need is called *temp*, which records robot locations but is different from *location*. The *location* is used in search process while the *temp* is used in determining rotation and movement based on path we got. The *location* has nothing to do with robot location in tester file, but *temp* follows exactly the same as robot location in tester file.

The third additional variable is a *flag*. It is the flag to mark the round of run. It was initialed as 0, and after the completion of exploration step, it was changed to 1 meaning that the second run starts.

Last but not least, a *path* should be stored. This path was a list generated by searching process and it should be reconstructed in the dictionary format.

Realization of A* star algorithm and Dijkstra's Algorithm

This is not a big deal compared to other issues. The only tricky thing is to understand all sorts of conditions and actions to each condition. Generally I thought about three conditions where a location would be.

If a newly reachable location was in close set, ignore it and try another directions. Otherwise, calculate a score for this location.

Then, if the new location was not in active set, it has not been reached before so the score calculated must be the minimum score for it. Append the location into the active set. Set the current location as the parent of the new location.

Then, if the new location was in active set, and the new score calculated is smaller than existing score stored at the location, replace the location with the new score. Also replace the parent of the new location.

After traversing all children for a location, it will be placed into close set and removed from active set.

After reaching target area, the optimal path are stored in parent.

Some functions have been created to make it more readable and let it go more smoothly. The *create_dict* function turns the path, a list with tuples in it, into a dictionary format, with parent nodes as the keys and children nodes as the values. The *direction* function aims at calculating rotation based on starting point, ending point and heading direction. The *heading2* function is calculating relative direction of ending point to starting point. The *move* function is to have the robot do actual move according to heading direction, current location and the number of steps. The *ReachTarget* function returns a Boolean value, telling us whether a specific location is within target area. The *DistToTarget* function calculates the Manhattan distance between a location and the target area. The *InitialMap* function creates a numpy array filled with 1000 copying the maze.

When should the first run stop

Theoretically, in the first run, the robot could stop at any time after reaching goals. But I did not let it continue to explore the maze further after it reaches the goal. I only allowed the robot to move one step more. The reason is due to distance cost.

It is possible that the robot could move faster if it is able to move 2 or 3 steps at one time and there are less turnings in a path, so the shortest path calculated by A* algorithm and Dijkstra's Algorithm might be different(Remember that I only allow 1 step in searching by A* but more by Dijkstra's algorithm). But it is also possible that the fastest path calculated by Dijkstra's Algorithm might have a longer distance. In the real-world applications, we always consider time and distance synchronously while choosing an optimal path. Sometimes a shorter path might cost you more time due to traffic jam or too many traffic lights, but you may not be willing to select a longer path and drive 10 miles more to save only 10 minutes. I could let the robot explore by more steps, like 3 or 4, after reaching goals, but I would like to have the policy more strict, that when one path reaches the goal, I only allow 1 more chance for other path to extend no matter whether it would be able to reach the goal within 1 step. If so, that is better because we may have more candidates in Dijkstra's algorithm when searching for an optimal path.

How to reconstruct path and determine rotation and movement according the path

After each run, a dictionary was generated, containing each cell on the shortest path, as well as their relationship (parent and child). Children are the keys and parents are values. Each cell has exactly one parent cell. So we can start reconstructing the path from the target node to the starting point reversely, storing tuples in a list.

After getting the path list, we then calculated rotation and movement between two adjacent elements. Rotations are calculated by *direction* function based on two locations and heading direction. Movements are the aggregation of differences between x and y

coordinator values of two nodes. Two adjacent nodes are at the same row or the same column all the time. For each location the robot goes through, we have a rotation and movement for it to take action, and at the same time, its heading direction is updated.

When encountering bugs, what could I do to fix them

In the process of writing and testing codes, I often encounter problems, like robot walking through the wall, not turning the right direction, etc. I usually checked some key information to find out the wrong sections.

First of all, I printed the score map in the *next_move* function to see if the A* and Dijkstra's algorithm output correct score maps.

If the score map was incorrect, the problem should be in A* algorithm or the Dijkstra's algorithm. The map would show obviously which scores are wrong. I then printed *location* in these two functions to see which specific location were the first to go wrong. If I found the place, I would then print sensors reading and heading direction. If they were correct, problems should be in value allocation process and I would carefully read through all possible conditions and the robot's response to each condition. If they were incorrect, I would check whether sensors reading and heading direction were updated correctly.

If the score map was correct, the problem should lay in *next_move* function. I would print *rob_location* in tester file and *temp* in *next_move* function to see whether two locations matches for each action and found the first location that did not match. Then check if the movement and rotation functions were used appropriately at that location. Usually I did found some tiny errors, like forgetting to calculate absolute but actual values between two points, which were sometimes negative.

3.3. Refinement

The initial solution: at the very beginning, I thought to use random model to explore the maze and then Dijkstra's algorithm to find the optimal path. In the process of random exploration, the robot would move for 1 step at one time and turn randomly based on accessibility of its neighbors. The robot recorded wall information and accessibility to its neighbors for each cell until it found the goal. However, I found it hard to reach the goal in a short time and the exploration process lasted for too long. Meanwhile, the movement is always 1 for Dijkstra's algorithm.

Intermediate solution: I changed the exploration algorithm from random model to A* algorithm by searching references online, as I learned that A* algorithm is widely used in map search since the algorithm approaches the goal gradually and would not deviate from correct path too much so as to reduce searching cost. The exploration result

improved a lot as it could find the goal much faster. Then I blocked cells that had not been reached in the exploration process so that robot would never reach them in optimization process. This action saved time for optimization process.

Final solution: I further optimize the algorithm by allowing robot to move multiple cells at one time in the optimization process. I noticed that the robot could move for up to three steps at one time. By applying this method, the total steps spent decreased from 31 to 17 steps and the optimal path it found also changed. The scores became much better.

4. Results

4.1. Model evaluation and validation

According to Metrics section, a score would be calculated to measure its performance. The score and the number of moving steps for each maze are listed below.

	Maze 1(12*12)	Maze 2(14*14)	Maze 3(16*16)
Score	18.133	23.467	27.733
Step#	17	22	27

Table 1 Results for the final model. Scores are calculated with standard formula. The step number represents the number of steps on the shortest path from starting point to ending point.

From the results above, we could know that the robot took 17, 22, 27 steps from the starting point to the goal, respectively, for mazes of three dimensions. The following maps will show optimal paths. Final scores are quite low, especially when comparing them to that of the benchmark model, which are 18.133, 23.467, 27.733. It is not surprising that the score would increase as the dimension of a maze increases as the robot would take more time to reach the goal.

The model has a very good robustness, since no matter how many attempts I did for tester file, scores and the number of steps for the shortest path are always the same. Variances of scores and step number for minimum path are 0.

An interesting phenomenon is that score maps of three mazes do not have any cells filled with values of 1000 except cells in target area. I don't know whether mazes were deliberately designed like this, but it appears to me that the target areas for three mazes happened to have the longest distance from their starting points, so that A* algorithm would reach nearly all cells within the maze before reaching goals. So in these three cases,

the function of A* algorithm that limits the searching area for Dijkstra's Algorithm did not work well. Perhaps this function would be better reflected in a larger maze.

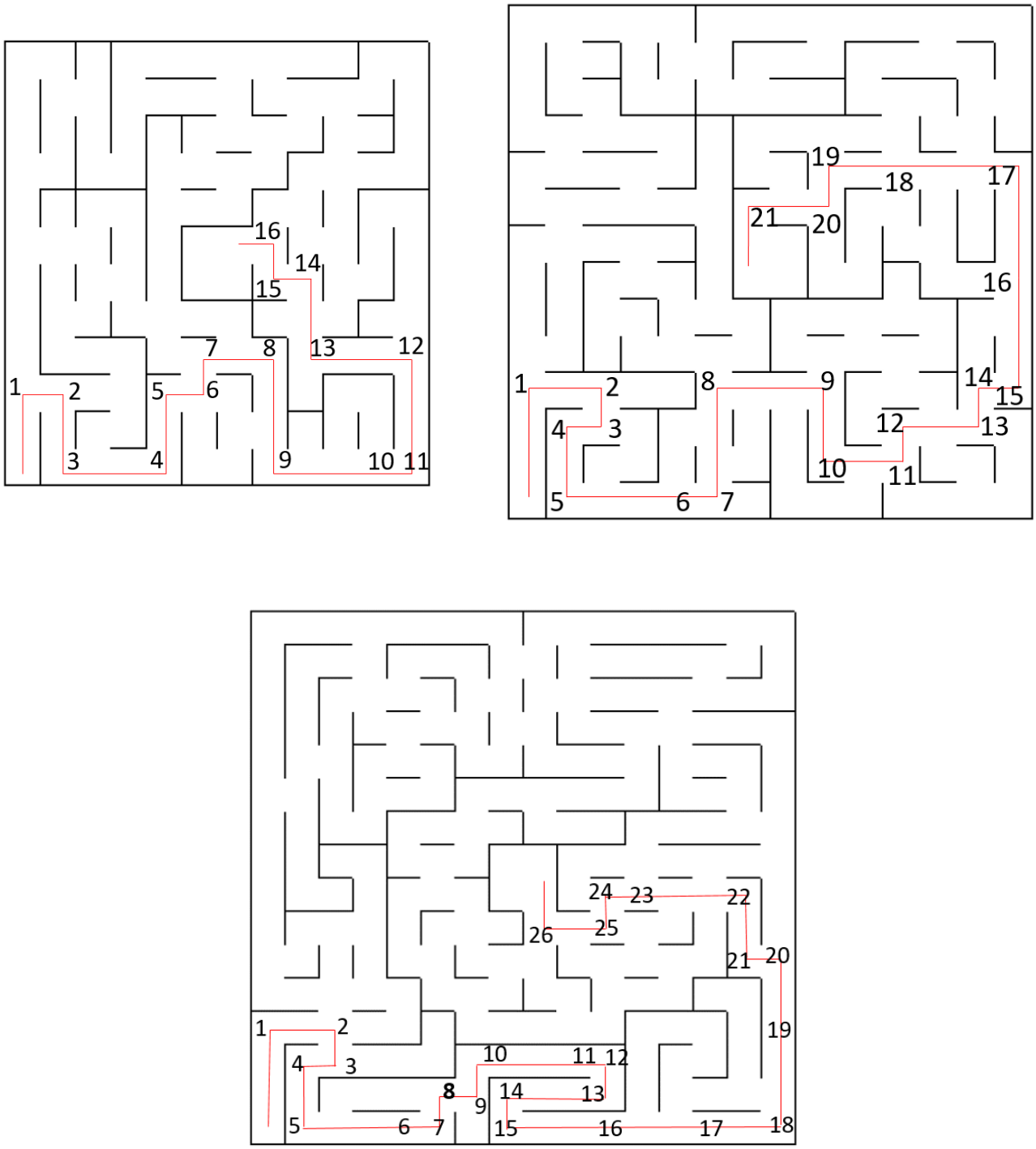

*Figure 3 Optimal paths for three mazes. Top left is the 12*12 maze. Top right is the 14*14 maze. Bottom is the 16*16 maze. Red lines are the shortest path. Numbers reflects actions the robot took*

4.2. Justification

The performance of the benchmark model is far worse than the final model, for it is unlikely to reach the goal in a limited steps. I ran the benchmark model for 30 times for each maze and relevant results are tabled as below.

	Maze 1(12*12)	Maze 2(14*14)	Maze 3(16*16)
Success rate for exploration	90%	53.33%	26.67%
Success rate both optimization and exploration process	26.7%	10%	3.33%
Lowest score	58.067	156.2	370.87
Highest score	350.933	216.8	370.87

Figure 4 Statistical analysis for the benchmark model.

The result shows that it is really hard for the robot to find the goal, not mentioning the optimal maze. The success rate of completion is only 26.7% for a 12*12 maze and it is even smaller for mazes of larger size. The larger the dimension of a maze is, the lower the success rate is, giving a maximum trials number of 1000. The variance of scores is larger when a maze is smaller. Scores are much too high, which are usually over 100.

By comparison, the final model could always find the goal. And the results are stable since the shortest path is always the same no matter how many attempts I did. Scores are pretty lower. Obviously, the final model is much better than the benchmark model from every perspective.

5. Conclusion

5.1. Free-Form Visualization

The figure 3 shows the map result of Dijkstra's algorithm for 12*12 maze. The map shows that how step increases from the starting point at the bottom left corner. As we can see, robot could move at most 3 steps for one action. Three of four target cells are marked as 1000 since they have not been reached in A* algorithm. The map shows that there are two ways from the starting point to the goal with exactly the same number of steps, upper way and lower way. However, it is clear that the lower way has a closer distance so the robot chose to run through the lower way even though two ways have the same steps.

4	5	7	9	10	10	10	11	11	11	14	15
4	5	6	9	10	10	10	12	13	13	13	16
3	5	7	9	9	11	11	11	12	14	15	16
3	4	7	8	9	10	10	10	15	14	15	15
3	4	4	5	8	9	9	16	15	15	15	14
2	3	3	6	8	1000	17	16	14	15	16	13
2	4	4	5	8	1000	1000	15	14	15	16	13
2	4	5	6	7	7	7	15	14	15	14	13
1	4	5	3	8	7	8	8	13	13	13	12
1	2	2	2	5	6	6	9	14	11	11	12
1	3	4	3	5	7	7	8	11	11	11	12
0	3	4	4	4	7	7	9	10	10	10	11

Figure 5 Score maps generated by Dijkstra's Algorithm

To illustrate the reason why there are few cells filled with 1000 except cells in target area, the figure 4 has been showed. This map is the g score map generated by A* algorithm. Remember that g score for a single cell represents the minimum number of steps from starting point to this cell and it allows only one more chance for the robot to continue to explore after reaching goals. As you can see, the target area has the highest number of steps towards starting point among all cells within the maze. So it is impossible for the robot to not reach all cells.

5.2. Reflection

To sum up, the problem of this project is to find the optimal path from a starting point to goals within a maze, which is not known by the robot at the beginning. General solution contains two parts, exploration and optimization.

- Several new variables are set to meet requirements.

11	12	13	18	19	20	21	22	23	24	27	28
10	11	12	17	18	19	20	23	24	25	26	29
9	10	13	16	19	20	21	22	27	26	27	30
8	9	14	15	18	19	20	21	28	27	28	29
7	8	9	10	17	18	19	30	29	28	31	30
6	7	8	11	16	1000	32	31	30	29	32	29
5	8	9	12	15	1000	1000	32	31	30	33	28
4	9	10	13	14	15	16	31	30	31	28	27
3	8	7	6	15	14	15	16	29	28	27	26
2	3	4	5	12	13	14	17	30	23	24	25
1	4	7	6	11	14	15	18	21	22	23	24
0	5	8	9	10	15	16	19	20	21	22	23

Figure 6 G score map generated by A algorithm*

- Read preset variables, like sensors readings.
- Start exploring the maze until the next step after it reaching the goal with A* algorithm. Areas that have not been reached are marked as blocked.
- A path with minimum steps is created and reconstructed. Rotation and movement are generated based on the path and real-time location and heading direction so that tester file could be processed smoothly.
- Score map is passed along to the next step, optimization.
- Start the second run until it finds the goal with Dijkstra's Algorithm.
- A path with minimum steps is created and reconstructed. Rotation and movement are created again based on the path and real-time location and heading direction.
- Reach the goal and calculate scores for the model.

The most difficult aspect was to apply A* algorithm and Dijkstra's algorithm to the real world use. The basic logic of two algorithms are not difficult, but when thinking about condition combinations, I always found them tricky to fix as the program showed errors. I had to print out the actual actions the robot implemented, including its real-time location, sensors readings at this location, heading direction, directions it turns, steps it moved, etc, when I faced with errors.

5.3. Improvement

The biggest problem of this simulation could be that it is quite ideal and standard, which might not reflect the actual situation. For example, the robot could not take only 1 unit time to run for 3 unit cells while it should take 1 unit time to move 1 unit cell. Another example is that the robot could not move horizontally or vertically precisely. There should be some radian when moving and turning. So, many details, like the turnings, sensors detections, and rotations should be carefully thought.

Several potential solutions could be considered into the project.

- PID controller. It is a controller system that could be used to control speed. It has a feedback system and includes the error term which is calculated by subtracting desired values and real values so as to continuously make adjustments. When moving on a straight way, the robot could move faster since there are no barriers. When approaching walls, the robot would steadily reduce speed for the purpose of turning. So there should be no 3 movement limit here.
- Simultaneous localization and mapping(SLAM). It is a good tool for the robot to navigate and map. Probability is a base for this solution. Current location and turning choice are used to map and the map is used to navigate and make turning choices. A bundle of algorithms are used when we implement this solution.