Capstone Project of Machine Learning Engineer Nanodegree: Plot and Navigate a Virtual Maze

Yi-Yuan Chiang

February 6, 2017

1 Definition

There are three subsections in this section. They are: *Project Overview*, *Problem Statement*, and *Metrics*.

1.1 Project Overview

The purpose of this project is to design a virtual robot to navigate a given maze. The task of the robot is to plot a path from a corner of the maze to the center. Two runs are allowed. In the first run, the robot has to find the goal and the best path from corner to center. In the second run, the robot attempts to reach the center in the fastest time possible, using what it has learned in the first run. This rule is inspired from the APEC Micromouse Contest [1].

1.2 Problem Statement

As mentioned in the previous section, two runs are allowed for the robot. 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.

1.3 Metrics

The robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. A maximum of one thousand time steps is allotted to complete both runs for a single maze.

2 Analysis

There are three subsections in this section. They are: *Data Exploration and Visualization*, *Algorithms and Techniques*, and *Benchmark*.

2.1 Data Exploration and Visualization

The given mazes are specified and coded in the files with text format. On the first line of the text file is a number, n, describing the number of squares on each dimension of the maze.

On the following n lines are n comma-delimited four-bit coding numbers describing which edges of the square are open to movement. The coding method is described as below. 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.

Figure 1 is a portion of an example of four-bit coding maze, in the bottom left corner, the number 1 means the upper edge of this square is open, 6 means both the right edge and the bottom edge are open.

From this coding method, we are easily to translate the given four-bit coding maze into human understandable mazes specified by walls. Figure 2a is the four-bit coding edition of Maze01; and Figure 2b is the normal one.

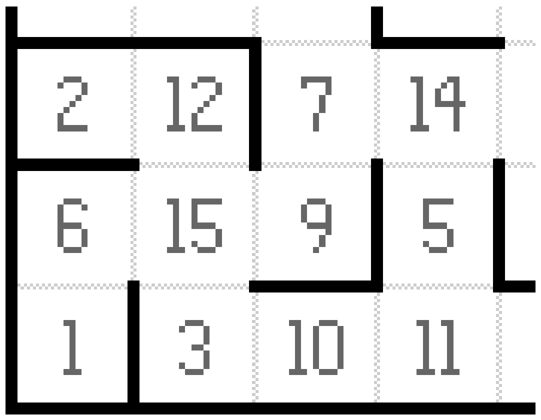
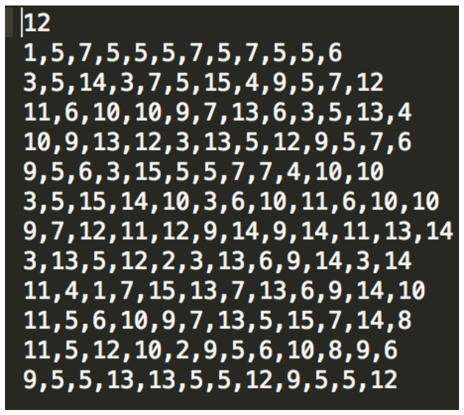
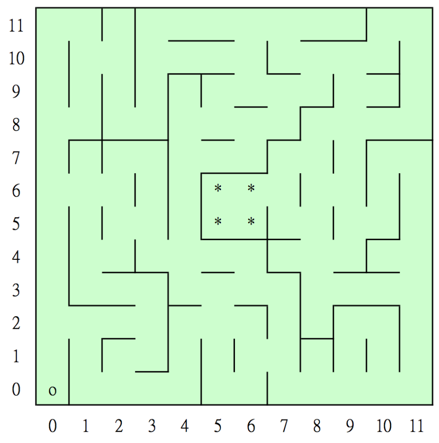


Figure 1: A portion of an example of four-bit coding maze.



```
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
```

(a) Four-bit coding edition of Maze01. (The given format)



(b) Normal edition of Maze01. The start is labeled as ‘o’ and the goal is labels as ‘*’.

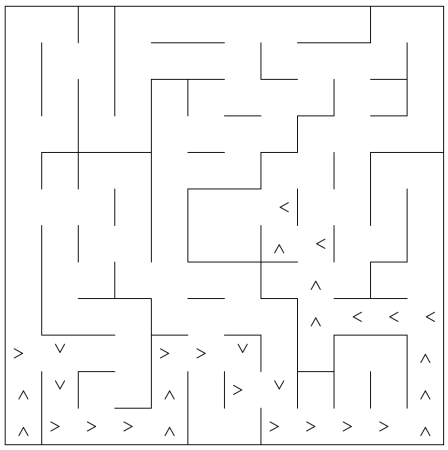
Figure 2: The four-bit coding representation and the normal wall representation of Maze01.

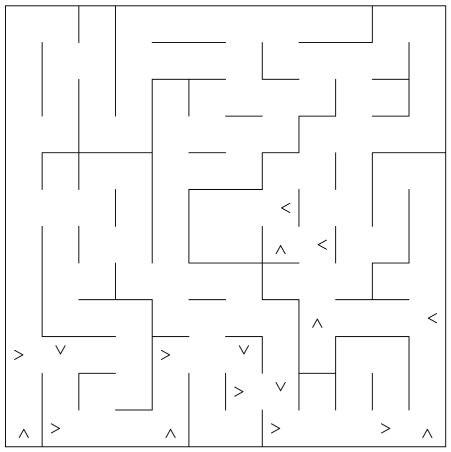
As for robot, it has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor; for example, in its starting position, the robot's left and right sensors will state that there are no open squares in those directions and at least one square towards its front. On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units. It is assumed that the robot's turning and movement is perfect. If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robot's new location and/or orientation to start the next time unit.

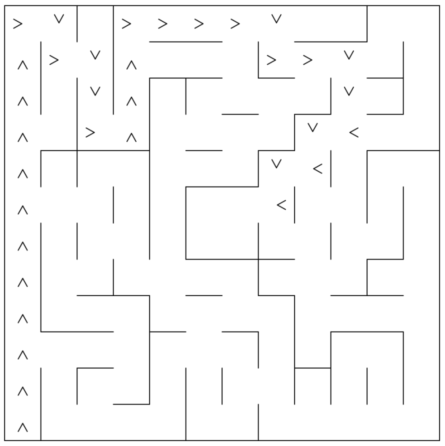
After understand the sensing method of robot, let's take a look at maze01 more detail. The start location is $(0, 0)$, and the goal is a square with four positions, they are $(5, 5)$, $(5, 6)$, $(6, 5)$, $(6, 6)$. When navigating such a maze, a robot may encounters at least two kinds of hazards. I named them as: *dead-end hazard* and *loop hazard*. When a robot encounters a dead-end hazard, it has no way for any direction except moves backwards. For example, when the robot get into the position $(1, 7)$, $(4, 9)$, $(8, 1)$, or $(10, 9)$, it encounters a dead-end hazard. The other hazard is loop hazard, which is caused by isolated walls. For example, the right-hand-side wall of position $(0, 9)$ and $(0, 10)$ may causes loop hazard. Our robots should avoid these hazards in the second run.

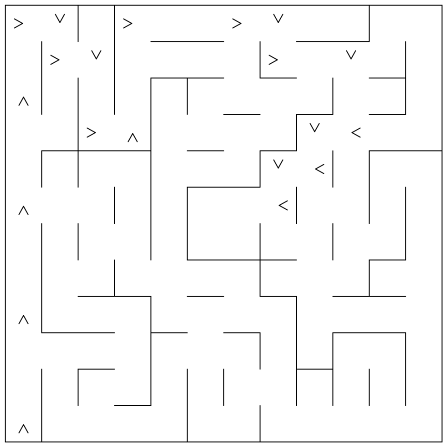
There are many paths for a robot to reach the goal. I just list three of them as shown in Figure 3a, 3c, 3e. Because the robot can move up to three units, the moving steps can be reduced. When the steps of a path are reduced, I call this path as an optimized path (Figure 3b, 3d, 3f).

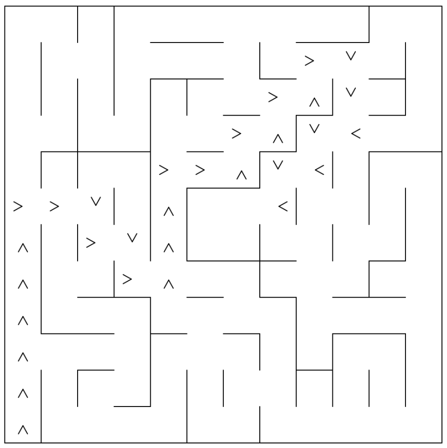
The shortest path of maze01 is path-1 (Figure 3a); it takes 30 steps. After optimization, only 17 steps are required (Figure 3b).


(a) Maze01: path-1 (30 steps)


(b) Maze01: path-1 optimized (17 steps)


(c) Maze01: path-2 (34 steps)


(d) Maze01: path-2 optimized (20 steps)


(e) Maze01: path-3 (30 steps)

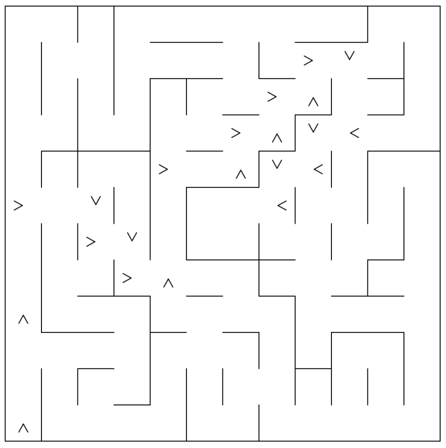

(f) Maze01: path-3 optimized (21 steps)

Figure 3: Example Paths of Maze01

2.2 Algorithms and Techniques

A robot finding a path from start to goal could be viewed as a search problem [5]. There are many search algorithms could be used to solve a robot path-planning problem, including *best-first search* (BFS), *uniform cost search* (UCS), *depth-first search* (DFS), and *depth-limited search* (DLS), etc [3]. These algorithms belong to *blind searches*, because during the search procedure, there is no additional information obtained. On the other hand, some kind of search algorithms called *heuristic searches*, because during the search procedure, there are additional information could be obtained. A well-known heuristic search algorithm is the A-star search [4], which will be used in this project. Except A-star search algorithm, DFS will also be implemented for the purpose of benchmarking.

A DFS could be implemented as a recursive or a non-recursive procedure. I used the non-recursive one. Notably, a maze could be viewed as a graph; all positions within the maze are interpreted as vertices or nodes of the graph. Besides, for any position, its *adjacent vertices* are the positions that can be arrived for a robot within one step. *Adjacent edges* of a node link the node and its adjacent vertices. The following is the pseudo code of DFS [7].

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

As for A-star search, the following is the pseudo code [2].

```
initialize the open list
initialize the closed list
put the starting node on the open list (you can leave its f at zero)

while the open list is not empty
    find the node with the least f on the open list, call it "q"
    pop q off the open list
    generate q's successors and set their parents to q
    for each successor
      if successor is the goal, stop the search
         successor.g = q.g + distance between successor and q
         successor.h = distance from goal to successor
         successor.f = successor.g + successor.h

         if a node with the same position as successor is in the OPEN list \
             which has a lower f than successor, skip this successor
         if a node with the same position as successor is in the CLOSED list \
             which has a lower f than successor, skip this successor
         otherwise, add the node to the open list
      end
      push q on the closed list
end
```

2.3 Benchmark

In this subsection, I will discuss what is a reasonable benchmark score I feel. I will use the three examples stated in Section 2.1 to demonstrate what the optimal scores they could be along with the reasonable score range they perhaps will be.

As mentioned in Section 1.3, 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. Therefore, the scores for the paths stated in Section 2.1 should be as shown in the Table 1.

Table 1: The scores of path-1, path-2, and path-3.

	Best scores	Reasonable scores
path-1	18	$22 \sim 27$
path-2	21.13	$25.67 \sim 31.33$
path-3	22	$26 \sim 31$

3 Methodology

In this section, there are three subsections; they are: *Data Preprocessing*, *Implementation*, and *Refinement*.

3.1 Data Preprocessing

Because the sensor specification and environment designs are provided, there is no data preprocessing needed in this project.

3.2 Implementation

The python scripts along with test mazes for this project includes the following files:

- robot.py: This script establishes the robot class. This is the only script I have modified.

- maze.py: The provided script contains functions for constructing the maze and for checking for walls upon robot movement or sensing.
- tester.py: The provided script runs to test the robot's ability to navigate mazes.
- showmaze.py: The provided script is used to create a visual demonstration of what a maze looks like.
- test_maze_01.txt ∼ test_maze_03.txt: The three provided sample files specify mazes upon which to test my robot.
- test_maze_04.txt: The 12x12 maze which is designed by myself for testing the robustness of my robot.

In order to understand the architecture of the robot environments, I plot a diagram as shown in Figure 4. The file `tester.py` is the tester of the whole environment. Tester can call the `Maze` class (`maze.py`) to obtain the given maze. Then, the tester can initialize a robot (`robot.py`), and send the only parameter, the dimension of the maze, to the robot. While a robot has been initialized, its location and heading is always $(0, 0)$ and 'up', respectively.

For the purpose of clarity, I summarize some of the characteristics of the robot as below.

- The robot is always initialized by the tester. While the robot is initialized, its location is $(0, 0)$ and heading is 'up'.
- The robot does not know any information about the given maze except the dimension. Of course, the maze is always a square.
- The robot has to maintain the map of the maze when it visiting the maze.
- The robot has three sensors mounted on the front of the robot, its right side, and its left side. These sensors can detect the distances from the robot to walls.
- I implemented only one method (function), `next_move`, for the `Robot` class. (Except the constructor, of course.)

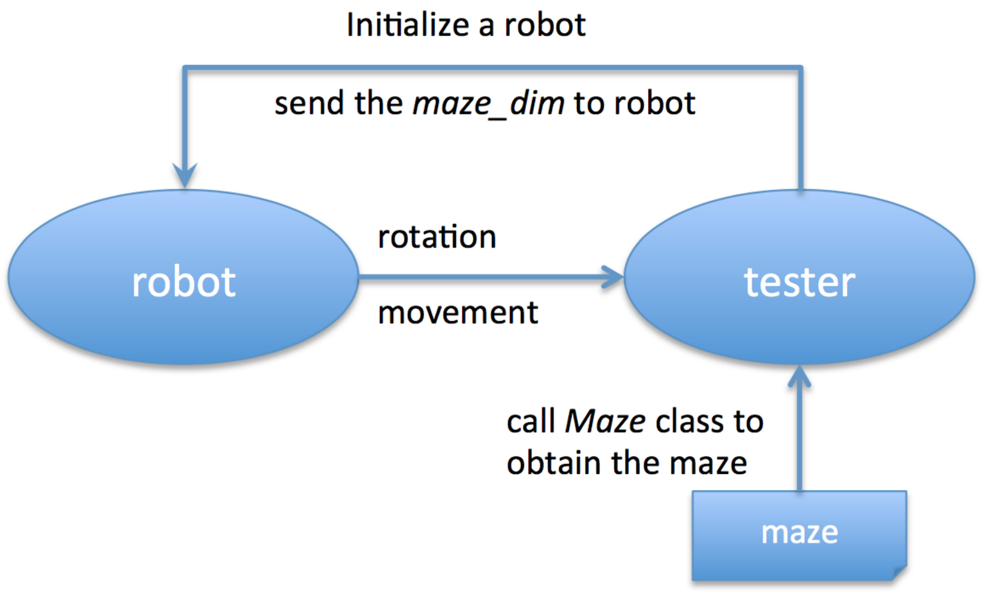

Initialize a robot
send the *maze_dim* to robot
robot
rotation
movement
tester
call *Maze* class to obtain the maze
maze

Figure 4: The architecture of the robot and the test environment.

- The `next_move` function must then return two values indicating the robot's rotation and movement on that time-step. Rotation is expected to be an integer taking one of three values: -90, 90, or 0, indicating a counterclockwise, clockwise, or no rotation, respectively. Movement follows rotation, and is expected to be an integer in the range $[-3, 3]$ inclusive.

- The robot will attempt to move that many squares forward (positive) or backwards (negative), stopping movement if it encounters a wall.

The search algorithms implemented for the robot are DFS and A-star. In some sense, my DFS could be viewed as an A-star using heuristic matrix with 0 in each entry. For an A-star search, the heuristic matrix plays a very important role in getting a good performance. I particularly mention my design of the heuristic matrix here.

For a $n \times n$ maze, my heuristic matrix, $H(i, j)$, is defined as below:

$$\begin{aligned} H(i,j) = \min(&\|(i,j) - (\frac{n}{2}-1, \frac{n}{2}-1)\|, \\ &\|(i,j) - (\frac{n}{2}-1, \frac{n}{2})\|, \\ &\|(i,j) - (\frac{n}{2}, \frac{n}{2}-1)\|, \\ &\|(i,j) - (\frac{n}{2}, \frac{n}{2})\|), \end{aligned}$$

where $\|\cdot\|$ is 1-norm, which is defined as:

$$\|\mathbf{v}\| = \sum_{i=0}^{k} |v_i|,$$

where $\mathbf{v} = (v_0, v_1, v_2, \ldots, v_k)$.

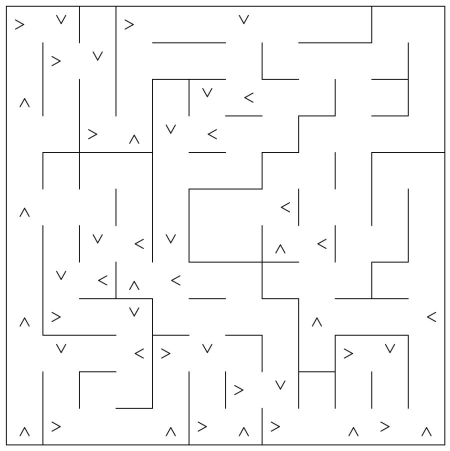
In particular, the heuristic matrix for 12×12 maze is defined as:

$$\begin{bmatrix} 10 & 9 & 8 & 7 & 6 & 5 & 5 & 6 & 7 & 8 & 9 & 10 \\ 9 & 8 & 7 & 6 & 5 & 4 & 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 7 & 6 & 5 & 4 & 3 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 6 & 5 & 4 & 3 & 2 & 2 & 3 & 4 & 5 & 6 & 7 \\ 6 & 5 & 4 & 3 & 2 & 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 4 & 3 & 2 & 1 & 0 & 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 3 & 2 & 1 & 0 & 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 5 & 4 & 3 & 2 & 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 6 & 5 & 4 & 3 & 2 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 7 & 6 & 5 & 4 & 3 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 8 & 7 & 6 & 5 & 4 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 9 & 8 & 7 & 6 & 5 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix}$$

Now, we apply the heuristic matrix to the three given mazes, a preliminary result could be obtained. The following is the result while Maze01 is tested.

```
******************************************************************************
Starting run 0.
[(0, 0), 'up'],
[(0, 1), 'up'],
[(0, 2), 'up'],
[(0, 3), 'up'],
[(0, 4), 'up'],
[(0, 5), 'up'],
[(0, 6), 'right'],
[(1, 6), 'right'],
[(2, 6), 'down'],
[(2, 5), 'right'],
[(3, 5), 'up'],
[(3, 6), 'up'],
[(3, 7), 'left'],
[(2, 7), 'left'],
[(3, 7), 'up'],
[(3, 7), 'up'],
[(3, 6), 'up'],
[(3, 6), 'up'],
[(3, 5), 'right'],
[(3, 5), 'down'],
[(3, 4), 'right'],
[(4, 4), 'up'],
[(4, 5), 'up'],
[(4, 6), 'up'],
[(4, 7), 'right'],
[(5, 7), 'right'],
[(6, 7), 'up'],
[(6, 8), 'left'],
[(5, 8), 'left'],
[(4, 8), 'up'],
[(4, 9), 'up'],
[(4, 8), 'left'],
[(4, 8), 'left'],
[(5, 8), 'left'],
[(5, 8), 'up'],
[(5, 9), 'right'],
[(6, 9), 'up'],
[(6, 10), 'left'],
[(5, 10), 'left'],
[(4, 10), 'left'],
[(3, 10), 'down'],
[(3, 9), 'down'],
[(3, 8), 'left'],
[(2, 8), 'up'],
[(2, 9), 'up'],
[(2, 10), 'left'],
[(1, 10), 'down'],
[(1, 9), 'down'],
[(1, 8), 'left'],
[(0, 8), 'down'],
[(0, 7), 'down'],
[(0, 8), 'left'],
[(0, 8), 'up'],
[(0, 9), 'up'],
[(0, 10), 'up'],
[(0, 11), 'right'],
[(1, 11), 'right'],
[(0, 11), 'up'],
[(0, 11), 'up'],
[(0, 10), 'up'],
[(0, 10), 'up'],
[(0, 9), 'up'],
[(0, 9), 'up'],
[(0, 8), 'left'],
[(0, 8), 'left'],
[(1, 8), 'down'],
[(1, 8), 'down'],
[(1, 9), 'down'],
[(1, 9), 'down'],
[(1, 10), 'left'],
[(1, 10), 'left'],
[(2, 10), 'up'],
[(2, 10), 'up'],
[(2, 11), 'up'],
[(2, 10), 'up'],
[(2, 10), 'up'],
[(2, 9), 'up'],
[(2, 9), 'up'],
[(2, 8), 'left'],
[(2, 8), 'left'],
[(3, 8), 'down'],
[(3, 8), 'down'],
[(3, 9), 'down'],
[(3, 9), 'down'],
[(3, 10), 'left'],
[(3, 10), 'up'],
[(3, 11), 'right'],
[(4, 11), 'right'],
[(5, 11), 'right'],
[(6, 11), 'right'],
[(7, 11), 'down'],
[(7, 10), 'right'],
[(8, 10), 'down'],
[(8, 9), 'left'],
[(7, 9), 'down'],
[(7, 8), 'down'],
[(7, 9), 'left'],
[(7, 9), 'left'],
[(8, 9), 'down'],
[(8, 9), 'down'],
[(8, 10), 'right'],
[(8, 10), 'right'],
[(9, 10), 'down'],
[(9, 9), 'down'],
[(9, 8), 'left'],
[(8, 8), 'down'],
[(8, 7), 'left'],
[(7, 7), 'down'],
[(7, 6), 'left'],
[(6, 6), 'up'],
Ending first run. Starting next run.
Starting run 1.
[(0, 0), 'up'],
[(0, 3), 'up'],
[(0, 6), 'right'],
[(2, 6), 'down'],
[(2, 5), 'right'],
[(3, 5), 'down'],
[(3, 4), 'right'],
[(4, 4), 'up'],
[(4, 7), 'right'],
[(6, 7), 'up'],
[(6, 8), 'left'],
[(5, 8), 'up'],
[(5, 9), 'right'],
[(6, 9), 'up'],
[(6, 10), 'left'],
[(3, 10), 'up'],
[(3, 11), 'right'],
[(6, 11), 'right'],
[(7, 11), 'down'],
[(7, 10), 'right'],
[(9, 10), 'down'],
[(9, 8), 'left'],
[(8, 8), 'down'],
[(8, 7), 'left'],
[(7, 7), 'down'],
[(7, 6), 'left'],
Goal found; run 1 completed!
Task complete! Score: 29.667
********************************************************************************
```

I plot the run-2 path on the maze. Figure 5a and 5b are the simulation results on Maze_01; Figure 6a and 6b are the simulation results on Maze_02; Figure 7a and 7b are


(a) Using DFS. (46 steps)

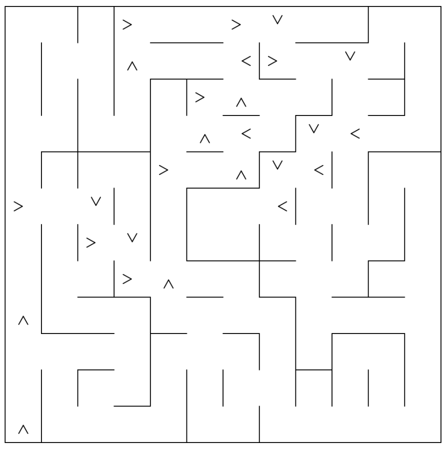

(b) Using A-star. (26 steps)

Figure 5: The simulation results on Maze-01. (Run-2)

Table 2: The simulation results on the three given mazes.

		Maze-01	Maze-02	Maze-03
DFS	score	49.576	71.233	73.667
	run-2 steps	46	65	59
A-star	score	29.667	44.267	43.100
	run-2 steps	26	41	39

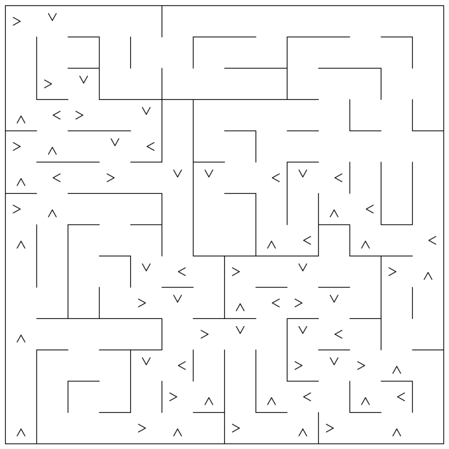
the simulation results on Maze_03. The scores and run-2 steps are shown in Table 2.

3.3 Refinement

I this subsection, I would like to know whether it is possible to improve the performance further by using other heuristic matrix. Notable, using other heuristic matrix sounds not make sense in general situations if we (or the robot) cannot get further information about the maze. I did this experiment just because I want to know the influences of the heuristic matrix. Let's start the experiment from Maze_01.

Let's named the previously defined heuristic matrix as H_1 (page 11); the all zeros heuristic matrix as H_0. Then,

$$H_2(i,j) = j \qquad (j = 0, 1, 2, \ldots, n-1),$$


(a) Using DFS. (65 steps)

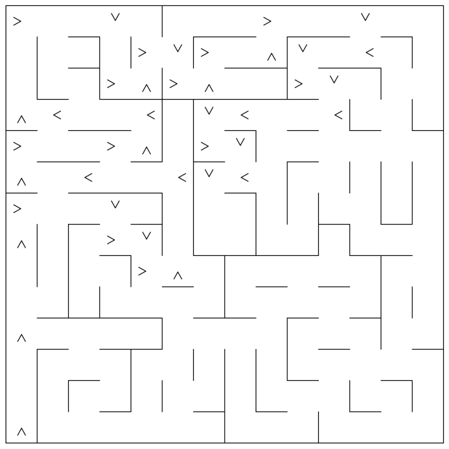
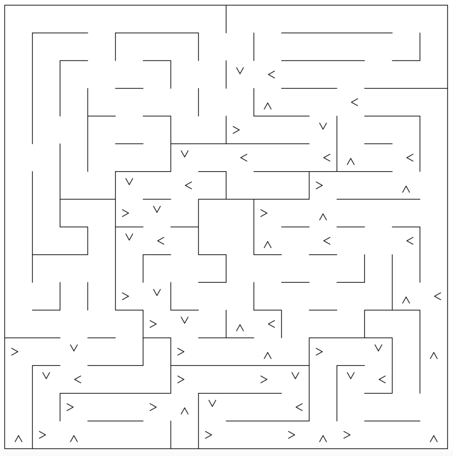

(b) Using A-star. (41 steps)

Figure 6: The simulation results on Maze-02. (Run-2)


(a) Using DFS. (59 steps)

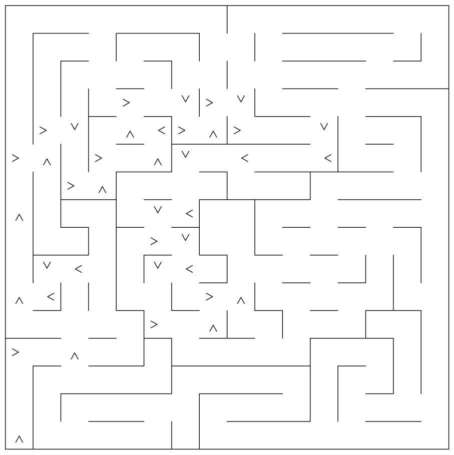

(b) Using A-star. (39 steps)

Figure 7: The simulation results on Maze-03. (Run-2)

Table 3: Using four kinds of heuristic matrix to do simulation on the given mazes.

		Maze-01	Maze-02	Maze-03
H_0	score	49.576	71.233	73.667
	run-2 steps	46	65	59
H_1	score	29.667	44.267	43.100
	run-2 steps	26	41	39
H_2	score	25.233	63.100	59.167
	run-2 steps	23	51	39
H_3	score	49.567	71.233	73.677
	run-2 steps	46	65	59

and

$$H_3(i,j) = i \qquad (i = 0, 1, 2, \ldots, n-1).$$

Using the four kinds of heuristic matrix to do simulation, the results could be obtained as shown in Table 3. In average, H_1 is a good design of heuristic matrix. The robot get better performance while choosing H_2 on Maze_01, that is because the optimal path of Maze_01 is near the bottom of the Maze. Figure 8 shows the paths of Maze_01 while choosing H_1 and H_2.

From this experiment, we know that H_1 is a good choice in general situation. The design of heuristic matrix will influence the performance. If we know some information about the maze, we indeed can design a better heuristic matrix to obtain better performance.

4 Results

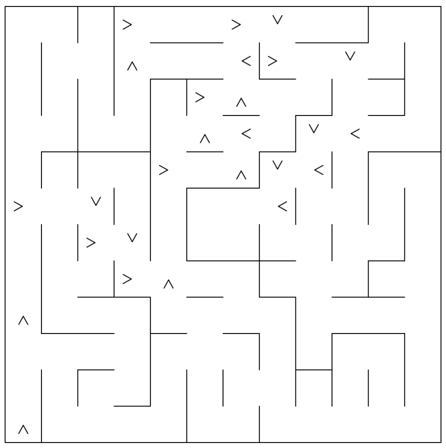
This section includes two subsections: *Model Evaluation and Validation* and *Justification*.

4.1 Model Evaluation and Validation

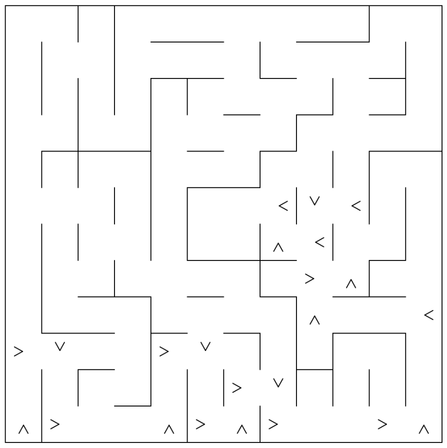
4.2 Justification

5 Conclusions

Three subsections are included in this section. They are: *Free-Form Visualization*, *Reflection*, and *Improvement*.



(a) Choosing H_1 as the heuristic matrix.



(b) Choosing H_2 as the heuristic matrix.

Figure 8: The paths of Maze-01 while choosing different heuristic matrix.

5.1 Free-Form Visualization

(Use this section to come up with your own maze. Your maze should have the same dimensions (12x12, 14x14, or 16x16) and have the goal and starting positions in the same locations as the three example mazes (you can use test_maze_01.txt as a template). Try to make a design that you feel may either reflect the robustness of your robot's algorithm, or amplify a potential issue with the approach you used in your robot implementation. Provide a small discussion of the maze as well.)

5.2 Reflection

5.3 Improvement

(Consider if the scenario took place in a continuous domain. For example, each square has a unit length, walls are 0.1 units thick, and the robot is a circle of diameter 0.4 units. What modifications might be necessary to your robotâĂŹs code to handle the added complexity? Are there types of mazes in the continuous domain that could not be solved in the discrete domain? If you have ideas for other extensions to the current project, describe and discuss them here.)

References

[1] *APEC Micromouse Contest Rules 2015*, http://www.apec-conf.org/wp-content/uploads/2013/10/APEC_2015_Micromouse_Contest_Rules.pdf

[2] Rajiv Eranki, *Pathfinding using A* (A-Star)*, http://web.mit.edu/eranki/www/tutorials/search/, 2002.

[3] *G5AIAI– Introduction to Artificial Intelligence: Blind Searches*, http://www.cs.nott.ac.uk/ pszgxk/courses/g5aiai/003blindsearches/blind_searches.htm

[4] Nils J. Nilsson, *Artificial Intelligence: A New Synthesis*, 1st edition, Morgan Kaufmann Publishers, Inc., 1998.

[5] S. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 3rd edition, Pearson Education, Inc., publishing as Prentice Hall, 2010.

[6] Naoki Shibuya, “Plot and Navigate a Virtual Maze”, *Udacity Machine Learning Nanodegree Capstone Project Sample Report*, https://github.com/udacity/machine-learning/blob/master/projects/capstone/report-example-3.pdf, March 20, 2016.

[7] Wikipedia, *Depth-first search,* https://en.wikipedia.org/wiki/Depth-first_search