Plot and Navigate a Virtual Maze

Definition

Project Overview

Inspired by Micromouse competition, this project is about programming a robot which can automatically explore a virtual maze and find the fastest way to reach the destination.

Problem Statement

On each maze, the robot must complete two runs. The first run is an exploration. The robot is allowd to roam freely in the maze to not only find the goal room, but also to collect necessary information about the maze so that the robot can find a better path leading to the goal. In subsequent runs, the robot mouse is brought back to the start location. It must attempt to reach the center in the fastest time possible, using what it has previously learned.

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 are allotted to complete both runs for a single maze.

Analysis

Data Exploration and Visualization

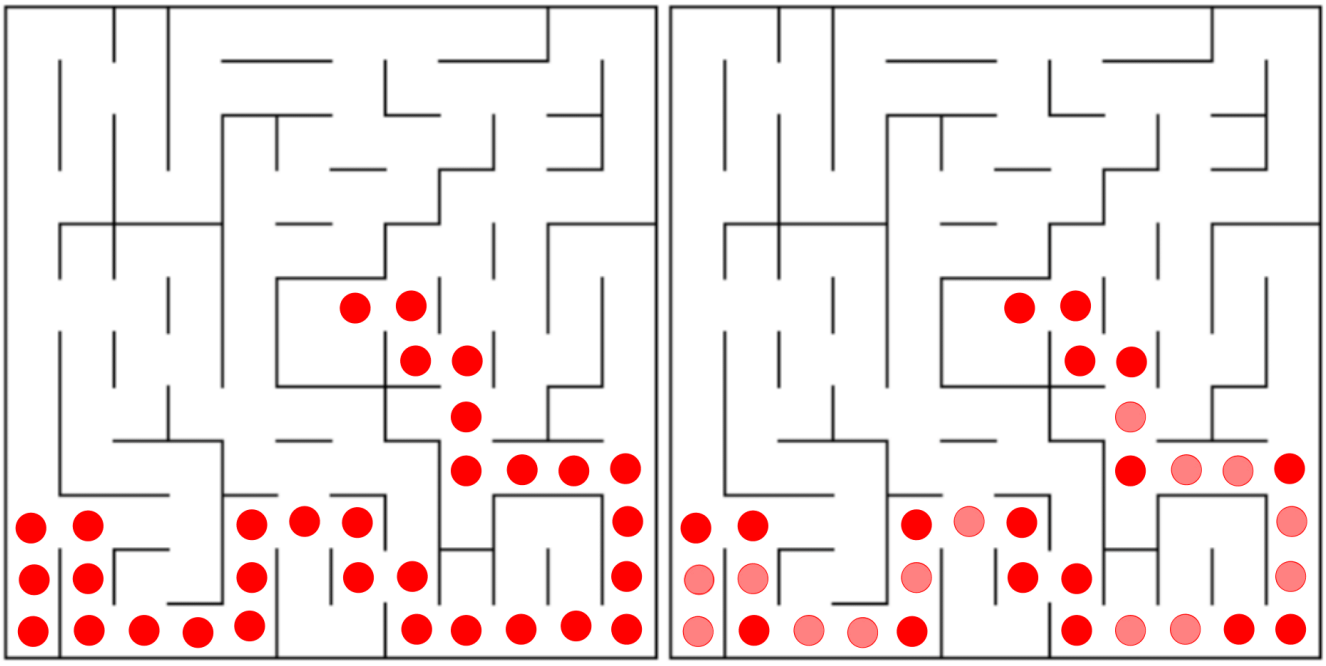
The maze exists on an n x n grid of squares, n even. The robot will start in the square in the bottom-left corner of the grid, facing upwards. The goal room, consisting of a 2 x 2 square, is located at the center of the maze.

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. The sensors will return the distance between the robot and walls in a list called 'sensors' as in [left distance, forward distance, right distance].

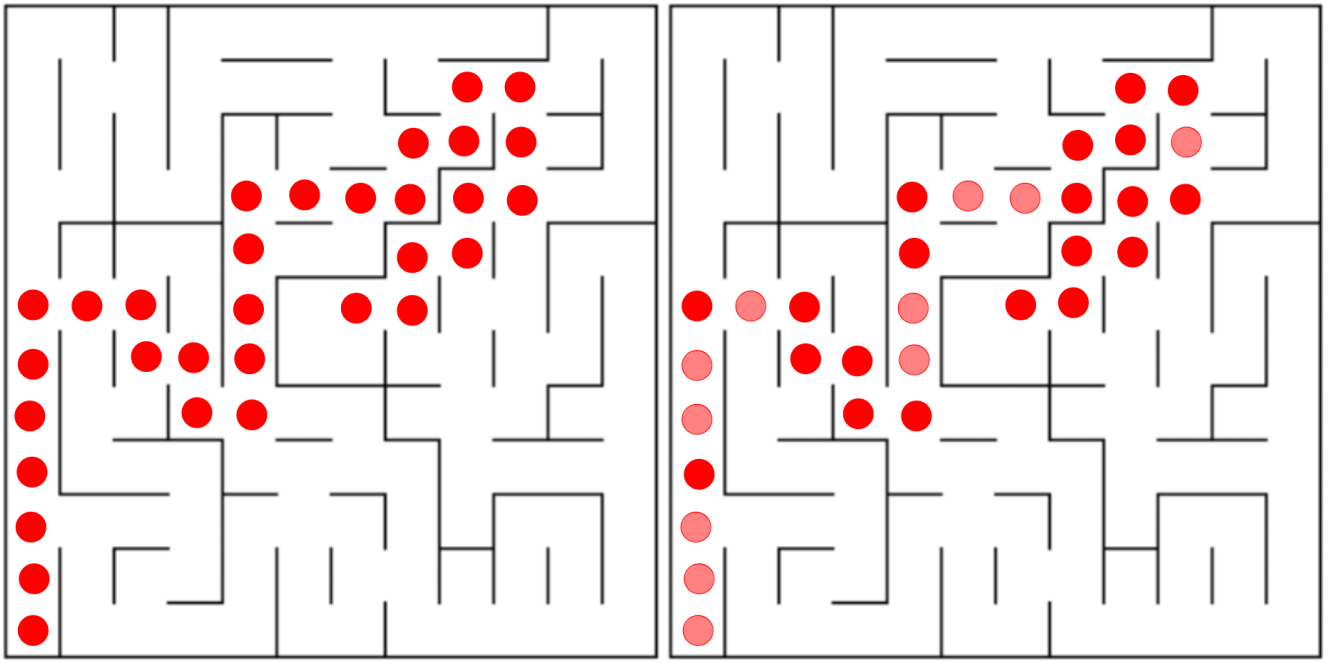
On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units. It is assumed that the robot’s turning and movement is perfect.

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. The first maze(12 x 12) is plotted using the given python script 'showmaze.py'. The path from the starting point(left-bottom cell) to the goal room is shown by a

series of red balls. There are actually four shortest paths in this maze. The robot need to cross 30 cells to arrive the destination on them. Since our robot can move up to three cells per unit time, so the robot need to try to make as less rotations as possible to achieve a short time trip.

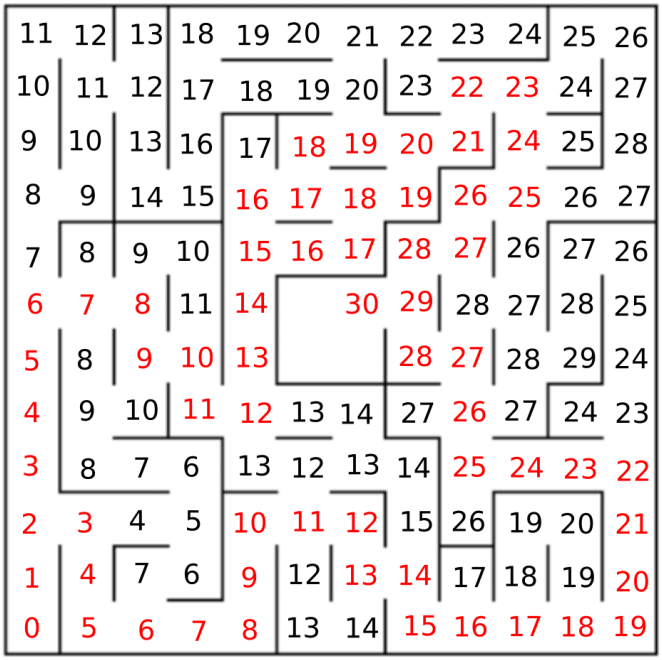


For the reason stated above, we only need to count part of the red balls. I marked those negligible balls as pink. So in total for this route, the robot need to take 17 movements to reach the goal room. This is actually the optimal route in this maze. In some other routes, the robot need to take more rotations in its trip, which severely slowdown the robot. The two pictures below show a case like that, again I use the pink ball to mark those cells included in a robot's stride.



Algorithms and Techniques

My robot is programmed based on flooding algorithm. The idea behind flooding algorithm is this: Imagine a maze has non-permeable walls and flat, level floors. If the robot turn on a hosepipe at the starting point of the maze, the water will start filling the maze. The below how shortest paths are found using this algorithm in a fully-known maze 1. The shortest route to the target destination will be taken by the first drop of water that arrives there.



By using this algorithm, the robot is able to find the shortest path given the starting point and its destination, if it has accurate information about the maze. However, even though initially the robot knows nothing about the structure of the maze. It can still adjust its routes each time it collects some more information on the maze. The robot will get closer and closer to the goal room gradually and finally it will gets there and learns a lot of information about the maze. This is how the robot can explore the maze in the first run. To make sure the robot gets enough knowledge about the maze, the first run is divided into three phases in my code.

Exploration

* Firstly the robot will roam the maze, step by step, following the left wall. This strategy will give the robot a basic knowledge on the structure of the maze. If finally the robot get to the goal room, the phase 2 is skipped. If the robot get back to the starting point by only following its left wall, then the robot enters phase 2.

* In phase 2, the robot will utilize the information collected in phase 1 to get to the goal room using flooding algorithm. During this process, the robot may touch some unknown area thus enrich its knowledge about the maze.

* In this final exploration phase, the robot will try to get a 100% map coverage on the maze. It will again use flooding algorithm to reach every untouched cell. Once

the robot remembers every inch of the maze, it can easily get to its destination on the shortest route generated by the algorithm. One problem about this strategy is that a 100% map coverage may not be necessary. In the later stage of phase 3, the robot will use lots of effort to get very tiny information about the maze, which may not be important for finding a shortest path.

Once the robot finish the exploration and the maze structure is well defined, it can easily find the shortest path leading to its destination.

Benchmark

If a robot can find the optimal path leading to the goal room by making less moves in the exploration run, it certainly should get a higher score. However, if a robot can save a significant number of moves by just making a little bit more steps in the second run, it still should get a high score.

Methodology

Data Preprocessing

The sensor specification and environment designs are already provided, therefore, no data pre-processing is required in this project.

Implementation

The variable ‘known_maze’ contains all the information the robot have learnt, it is updated each time the sensor has some feedback. It is updated by calling the function 'update_maze_info'. This function will collect the sensors' information and use it to update robot's memory. By doing some binary operation, the robot now some cells are open while others are closed.

```
def update_maze_info(self, loc, heading, sensors, maze_info):
    l = len(maze_info)
    x = loc[0]
    y = loc[1]
    N = -1
    E = -1
    S = -1
    W = -1

    if heading == 'up':
        W = sensors[0]
        N = sensors[1]
        E = sensors[2]
    elif heading == 'right':
        N = sensors[0]
        E = sensors[1]
        S = sensors[2]
    elif heading == 'down':
        E = sensors[0]
        S = sensors[1]
        W = sensors[2]
    elif heading == 'left':
        S = sensors[0]
        W = sensors[1]
        N = sensors[2]

    if N != -1:
        maze_info[x][y + N] = maze_info[x][y + N] & 0b1110
        if y + N + 1 < l:
            maze_info[x][y + N + 1] = maze_info[x][y + N + 1] & 0b1011
        for i in range(N):
            maze_info[x][y + i] = maze_info[x][y + i] | 0b0001
            maze_info[x][y + i + 1] = maze_info[x][y + i + 1] | 0b0100
    if E != -1:
        maze_info[x + E][y] = maze_info[x + E][y] & 0b1101
        if x + E + 1 < l:
            maze_info[x + E + 1][y] = maze_info[x + E + 1][y] & 0b0111
        for i in range(E):
            maze_info[x + i][y] = maze_info[x + i][y] | 0b0010
```

```
            maze_info[x + i + 1][y] = maze_info[x + i + 1][y] | 0b1000
    if S != -1:
        maze_info[x][y - S] = maze_info[x][y - S] & 0b1011
        if y - S - 1 >= 0:
            maze_info[x][y - S - 1] = maze_info[x][y - S - 1] & 0b1110
        for i in range(S):
            maze_info[x][y - i] = maze_info[x][y - i] | 0b0100
            maze_info[x][y - i - 1] = maze_info[x][y - i - 1] | 0b0001
    if W != -1:
        maze_info[x - W][y] = maze_info[x - W][y] & 0b0111
        if x - W - 1 >= 0:
            maze_info[x - W - 1][y] = maze_info[x - W - 1][y] & 0b1101
        for i in range(W):
            maze_info[x - i][y] = maze_info[x - i][y] | 0b1000
            maze_info[x - i - 1][y] = maze_info[x - i - 1][y] | 0b0010

    return maze_info
```

The variable 'coverage' is another one that is kept updated all the time. It records whether or not all the four directions of each cell have been sensed by the robot. Just like what we did on the maze information, each cell will be assigned a binary number, 0b0000 means that the robot knows nothing about this cell while ob1111 means the robot have collected all the information about this cell. This matrix will be used to realize a full map coverage on phase 3 of the 1st exploration run. It is updated by function 'update_coverage_info'.

```
def update_coverage_info(self, loc, heading, sensors):

    l = self.maze_dim
    x = loc[0]
    y = loc[1]
    N = -1
    E = -1
    S = -1
    W = -1

    if heading == 'up':
        W = sensors[0]
        N = sensors[1]
        E = sensors[2]
    elif heading == 'right':
        N = sensors[0]
        E = sensors[1]
        S = sensors[2]
    elif heading == 'down':
        E = sensors[0]
        S = sensors[1]
        W = sensors[2]
    elif heading == 'left':
        S = sensors[0]
        W = sensors[1]
        N = sensors[2]

    if N != -1:
        self.coverage[x][y + N] = self.coverage[x][y + N] | 0b0001
        if y + N + 1 < l:
            self.coverage[x][y + N + 1] = self.coverage[x][y + N + 1] | 0b0100
        for i in range(N):
            self.coverage[x][y + i] = self.coverage[x][y + i] | 0b0001
            self.coverage[x][y + i + 1] = self.coverage[x][y + i + 1] | 0b0100
    if E != -1:
        self.coverage[x + E][y] = self.coverage[x + E][y] | 0b0010
        if x + E + 1 < l:
            self.coverage[x + E + 1][y] = self.coverage[x + E + 1][y] | 0b1000
        for i in range(E):
```

```
            self.coverage[x + i][y] = self.coverage[x + i][y] | 0b0010
            self.coverage[x + i + 1][y] = self.coverage[x + i + 1][y] | 0b1000
    if S != -1:
        self.coverage[x][y - S] = self.coverage[x][y - S] | 0b0100
        if y - S - 1 >= 0:
            self.coverage[x][y - S - 1] = self.coverage[x][y - S - 1] | 0b0001
        for i in range(S):
            self.coverage[x][y - i] = self.coverage[x][y - i] | 0b0100
            self.coverage[x][y - i - 1] = self.coverage[x][y - i - 1] | 0b0001
    if W != -1:
        self.coverage[x - W][y] = self.coverage[x - W][y] | 0b1000
        if x - W - 1 >= 0:
            self.coverage[x - W - 1][y] = self.coverage[x - W - 1][y] | 0b0010
        for i in range(W):
            self.coverage[x - i][y] = self.coverage[x - i][y] | 0b1000
            self.coverage[x - i - 1][y] = self.coverage[x - i - 1][y] | 0b0010

    return self.coverage
```

The main part of this code is the flooding algorithm. There are tiny differences between applying this method to the central goal room or a unreached cell. The function 'flooding' and 'get_routes' are for the goal room, while 'flooding_to_other_cells' and 'get_routes_from_other_cells' are used when the robot try to get a full maze coverage. In the 'flooding' function, the program keeps scanning the whole maze, then will assign a new value to the boundary of the 'water' when our robot find them.

```
def flooding(self, robot_loc):

    flood = np.ones((self.maze_dim, self.maze_dim), dtype = int) * (-1)
    flood[robot_loc[0]][robot_loc[1]] = 0
    des = self.maze_dim / 2
    d = 1
    while True:
        for i in range(self.maze_dim):
            for j in range(self.maze_dim):
                if(flood[i][j] != -1):
                    continue
                n = self.get_neighbor_flood_value([i, j], self.known_maze, flood)
                if(n[0] == d - 1 or n[1] == d - 1 or n[2] == d - 1 or n[3] == d - 1):
                    flood[i][j] = d
                if(flood[des-1][des-1] != -1 or flood[des-1][des] != -1 or flood[des][des-1]
!= -1 or flood[des][des] != -1):
                    return flood
        d += 1
```

During this process, the program needs to find those connected neighbors of a specifc cell using the function 'get_neighbors', and use the function 'get_neighbor_flood_value' to obtain their 'flood value'.

```
def get_neighbors(self, loc, maze_info):

    binary_info = '000' + bin(maze_info[loc[0]][loc[1]])[2:]

    if int(binary_info[-1]):
        u = (loc[0], loc[1] + 1)
    else:
        u = None
    if int(binary_info[-2]):
        r = (loc[0] + 1, loc[1])
    else:
        r = None
    if int(binary_info[-3]):
        d = (loc[0], loc[1] - 1)
```

```
    else:
        d = None
    if int(binary_info[-4]):
        l = (loc[0] - 1, loc[1])
    else:
        l = None

    return [u, r, d, l]
```

Once the robot have the routes leading to the its destination(the central goal room or other cells it want to get to), another two functions are used to find the route with least moves. The function 'moves_refinement' can recognize those small steps that can be replaced by a large stride. Most of the routes will be shorten. 2 or 3 successive 1-step moves will be replaced by a single longer move. Out of these refined routes, we use the function 'find_shortest_moves' to find the optimal route. If there are more than one optimal routes, we make a random choice.

```
def moves_refinement(self, moves):

    for i in range(len(moves)):
        j = 0
        while j < len(moves[i]) - 1:
            if moves[i][j + 1][0] == 0 and moves[i][j][1] + moves[i][j + 1][1] <= 3
and moves[i][j][1] + moves[i][j + 1][1] >= -3:
                    moves[i][j] = (moves[i][j][0], moves[i][j][1] + moves[i][j + 1][1])
                    del moves[i][j + 1]
            else:
                j += 1

    return moves

def find_shortest_moves(self, moves):

    l = []
    for i in range(len(moves)):
        l = l + [len(moves[i])]
    mi = min(l)
    n = 0
    while n < len(moves):
        if len(moves[n]) > mi:
            del moves[n]
        else:
            n += 1

    return moves
```

Refinement

In the early version of this code, the robot was not programmed to try to get a full maze coverage in the exploration run. There was also three phases. But the difference is that in the 3rd phase, the robot did another left(or right) wall follower search instead of trying to reach every blind point in the maze. The old strategy could miss some important information in a larger maze. So the average performance for the 16 x 16 maze is not quite satisfactory. After that, I modified the 3rd phase so that the robot can learn all the maze structures.

Results

Model Evaluation and Validation

The optimal moves in each maze can be obtained by flooding algorithm if the robot learned all the structure information.

Maze 01 optimal moves

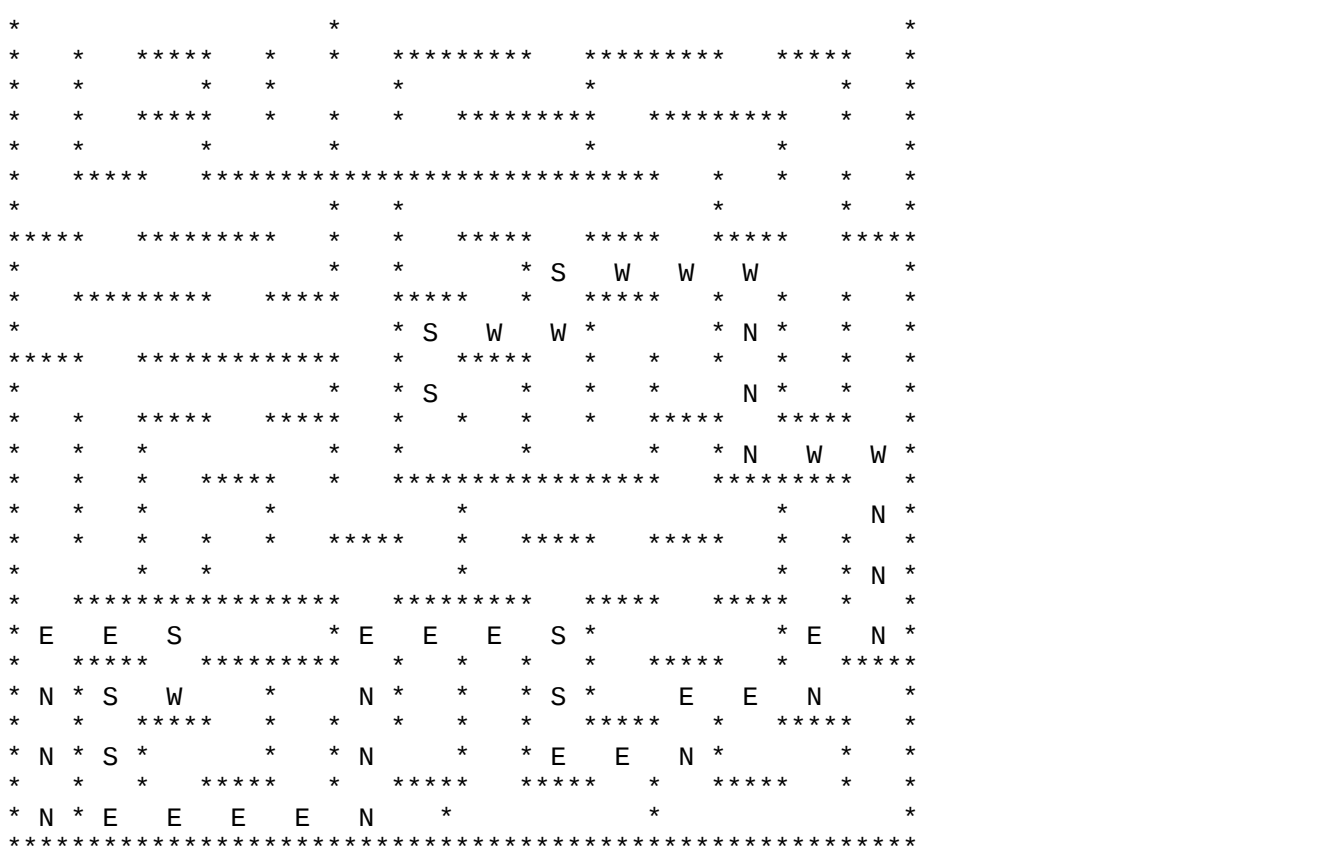
```
Path length: 30
*********************************************
*       *   *                       *       *
*   *   *   *   *********   *   *********   *   *
*   *       *               *               *   *
*   *   *   *   *********   *****   *   *****   *
*   *   *   *   *   *               *       *   *
*   *   *   *   *   *   *****   *****   *****   *
*       *       *               *               *
*   *************   *****   *****   *   *********
*   *   *       *           *       *   *       *
*   *   *   *   *   *********   *   *   *   *   *
*           *   *   *     W   W *       *   *   *
*   *   *   *   *   *   *   *   *   *   *   *   *
*   *   *       *   *       * N   W *       *   *
*   *   *   *   *   *************   *   *****   *
*   *       *               *     N     *       *
*   *   *********   *****   *****   *********   *
*   *           *               * N   W   W   W *
*   *********   *****   *****   *   *********   *
* E   S         * E   E   S *   *   *       * N *
*   *   *****   *   *   *   *   *****   *   *   *
* N * S *       * N *   * E   S *   *   *   * N *
*   *   *   *****   *   *   *   *   *   *   *   *
* N * E   E   E   N *       * E   E   E   E   N *
*********************************************

Number of moves: 17
(  0, 2)
( 90, 1)
( 90, 2)
(-90, 3)
(-90, 2)
( 90, 2)
( 90, 1)
(-90, 1)
( 90, 1)
(-90, 3)
(  0, 1)
(-90, 3)
(-90, 3)
( 90, 2)
(-90, 1)
( 90, 1)
(-90, 1)
```

Maze 02 optimal moves

There are in total four different optimal paths with exactly same number of moves in maze 2.

#1 optimal moves

Path length: 43



Number of moves: 22

```
(  0, 3)
( 90, 2)
( 90, 1)
( 90, 1)
(-90, 2)
(-90, 3)
(  0, 1)
(-90, 3)
( 90, 3)
( 90, 2)
(-90, 2)
(-90, 1)
( 90, 2)
(-90, 1)
( 90, 1)
(-90, 3)
(-90, 2)
```

```
( 90, 3)
(-90, 3)
(-90, 1)
( 90, 2)
(-90, 1)
```

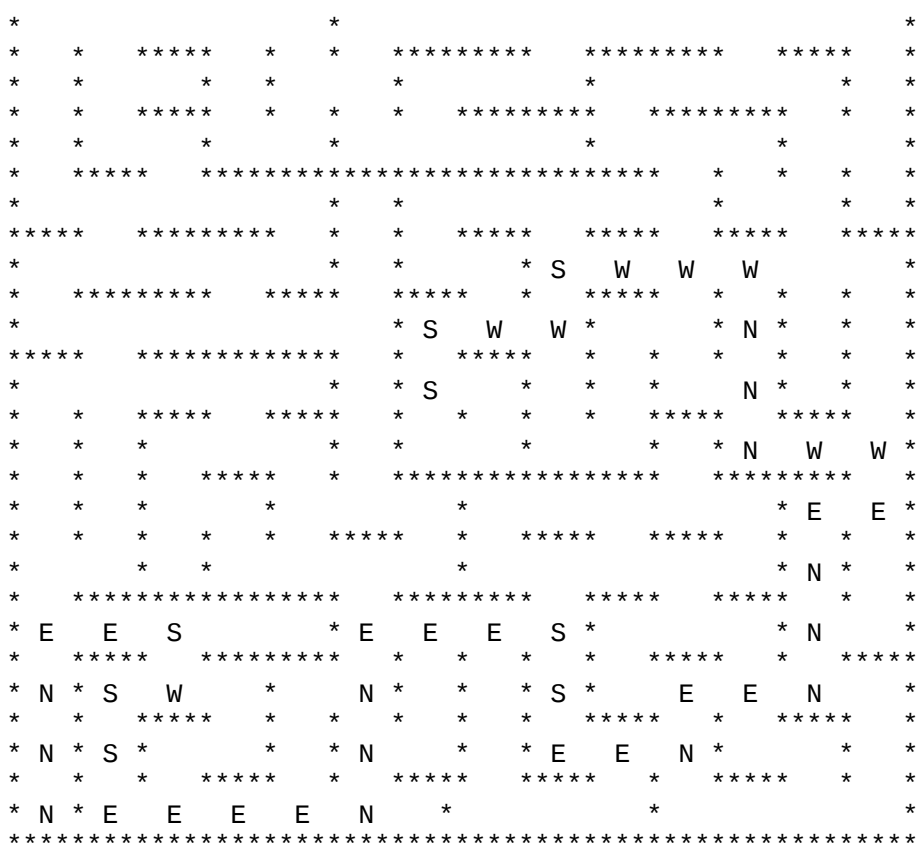
#2 optimal moves

```
Path length: 43
*********************************************************
*                   *                                   *
*   *   *****   *   *   *********   *********   *****   *
*   *       *   *       *           *               *   *
*   *   *****   *   *   *   *********   *********   *   *
*   *       *       *               *           *       *
*   *****   ******************************   *   *   *   *
*                   *   *                   *       *   *
*****   *********   *   *   *****   *****   *****   *****
*                   *   *       * S   W   W   W   W   W *
*   *********   *****   *****   *   *****   *   *   *   *
*                       * S   W   W *       *   *   * N *
*****   ************    *   *****   *   *   *   *   *   *
*                   *   * S     *   *   *       *   * N *
*   *   *****   *****   *   *   *   *   *****   *****   *
*   *   *           *   *       *       *   *         W *
*   *   *   *****   *   *****************   *********   *
*   *   *       *           *                   *     N *
*   *   *   *   *   *****   *   *****   *****   *   *   *
*       *   *               *                   *   * N *
*   *****************   *********   *****   *****   *   *
* E   E   S         * E   E   E   S *           * E   N *
*   *****   *********   *   *   *   *   *****   *   *****
* N * S   W     *     N *   *   * S *     E   E   N     *
*   *   *****   *   *   *   *   *   *****   *   *****   *
* N * S *       *   * N     *   * E   E   N *       *   *
*   *   *   *****   *   *****   *****   *   *****   *   *
* N * E   E   E   E   N   *           *               *
*********************************************************

Number of moves: 22
( 0,  3)
( 90, 2)
( 90, 1)
( 90, 1)
(-90, 2)
(-90, 3)
(  0, 1)
(-90, 3)
( 90, 3)
( 90, 2)
(-90, 2)
(-90, 1)
( 90, 2)
(-90, 1)
( 90, 1)
(-90, 3)
(  0, 3)
(-90, 3)
(  0, 2)
```

```
(-90, 1)
( 90, 2)
(-90, 1)
```

#3 optimal moves

```
Path length: 43
```

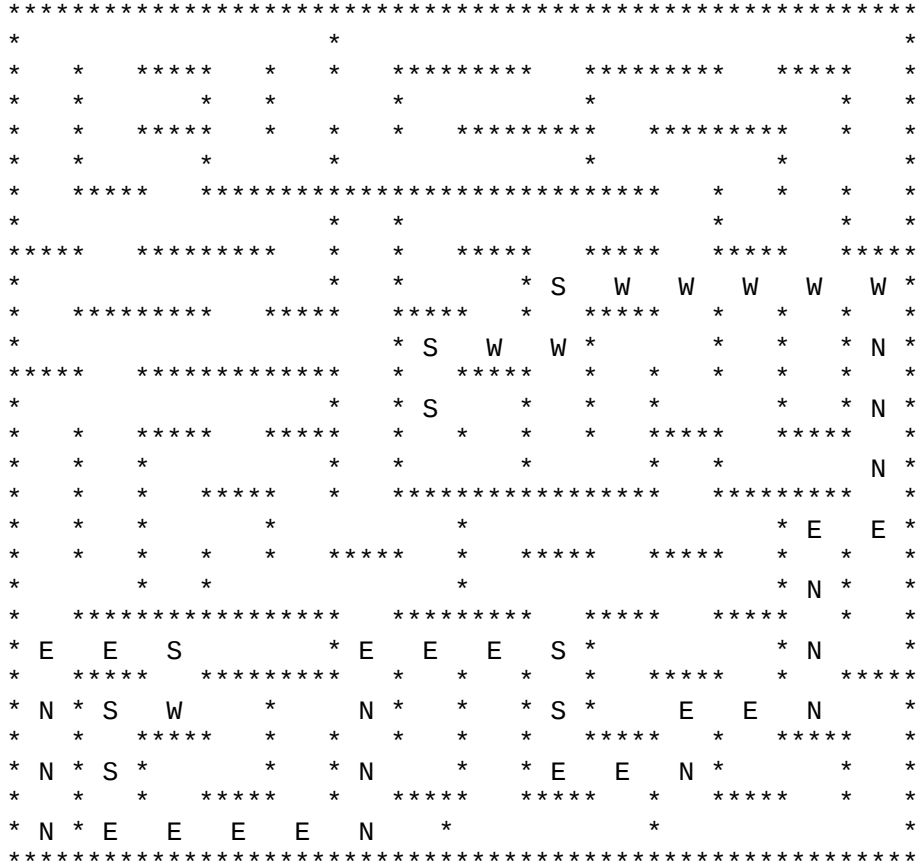


```
Number of moves: 22
(  0, 3)
( 90, 2)
( 90, 1)
( 90, 1)
(-90, 2)
(-90, 3)
(  0, 1)
(-90, 3)
( 90, 3)
( 90, 2)
(-90, 2)
(-90, 1)
( 90, 2)
(-90, 3)
( 90, 1)
(-90, 1)
(-90, 2)
( 90, 3)
(-90, 3)
(-90, 1)
( 90, 2)
```

(-90, 1)

#4 optimal moves

Path length: 43



Number of moves: 22

(0, 3)
(90, 2)
(90, 1)
(90, 1)
(-90, 2)
(-90, 3)
(0, 1)
(-90, 3)
(90, 3)
(90, 2)
(-90, 2)
(-90, 1)
(90, 2)
(-90, 3)
(90, 1)
(-90, 3)
(0, 1)
(-90, 3)
(0, 2)
(-90, 1)
(90, 2)
(-90, 1)

Maze 03 optimal moves

There are two different optimal moves for maze 3.

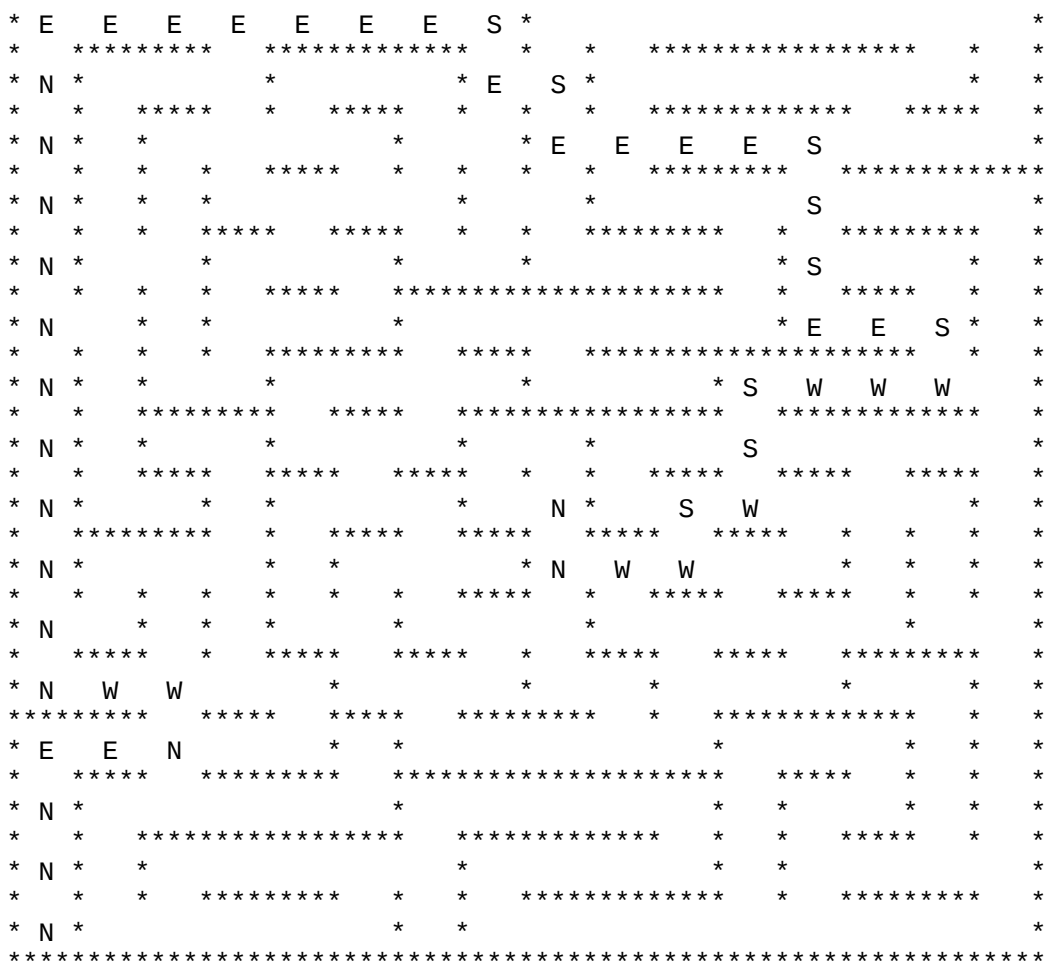
#1 optimal moves

```
Path length: 49
**********************************************************************
* E   E   E   E   E   E   E   S *                                    *
*   *********   *************   *   *   *****************   *   *
* N *           *           * E   S *                       *   *
*   *   *****   *   *****   *   *   *   *************   *****   *
* N *   *               *       * E   E   E   E   S             *
*   *   *   *   *****   *   *   *   *   *********   *************
* N *   *   *               *       *             S             *
*   *   *   *****   *****   *   *   *********   *   *********   *
* N *       *           *       *                 * E   E   S *   *
*   *   *   *   *****   ********************    *   *****   *   *
* N     *   *           *                       *         S *   *
*   *   *   *   *********   *****   ********************   *   *
* N *   *       *               *           * S   W   W   W     *
*   *   *********   *****   *****************   *************   *
* N *   *       *           *       *         S                 *
*   *   *****   *****   *****   *   *   *****   *****   *****   *
* N *       *   *           *     N *     S   W             *   *
*   *********   *   *****   *****   *****   *****   *   *   *   *
* N *           *   *           * N   W   W         *   *   *   *
*   *   *   *   *   *   *****   *   *****   *****   *   *   *
* N     *   *   *       *           *                   *       *
*   *****   *   *****   *****   *   *****   *****   *********   *
* N   W   W         *           *       *           *       *   *
*********   *****   *****   *********   *   *************   *   *
* E   E   N         *   *                   *           *   *   *
*   *****   *********   *********************   *****   *   *   *
* N *                   *                   *   *       *   *   *
*   *   *****************   *************   *   *   *****   *   *
* N *   *                   *               *   *               *
*   *   *   *********   *   *   *************   *   *********   *
* N *                   *   *                                   *
**********************************************************************

Number of moves: 25
(  0, 3)
( 90, 2)
(-90, 1)
(-90, 2)
( 90, 3)
(  0, 3)
(  0, 3)
(  0, 2)
( 90, 3)
(  0, 3)
(  0, 1)
( 90, 1)
(-90, 1)
( 90, 1)
```

```
(-90, 3)
(  0, 1)
( 90, 2)
(-90, 2)
( 90, 2)
( 90, 3)
(-90, 2)
( 90, 1)
(-90, 1)
( 90, 2)
( 90, 1)
```

#2 optimal moves

```
Path length: 49
```



```
Number of moves: 25
(  0, 3)
( 90, 2)
(-90, 1)
(-90, 2)
( 90, 3)
(  0, 3)
(  0, 3)
(  0, 2)
( 90, 3)
```

```
(  0, 3)
(  0, 1)
( 90, 1)
(-90, 1)
( 90, 1)
(-90, 3)
(  0, 1)
( 90, 3)
(-90, 2)
( 90, 1)
( 90, 3)
(-90, 2)
( 90, 1)
(-90, 1)
( 90, 2)
( 90, 1)
```

I did ten trails for each of the three mazes, recorded the number of moves in both runs, the path length in the second run and the score given by 'tester.py'. According to the implementation of my robot mentioned above, it will always get a full map knowledge and the optimal moves for each maze. The results are shown below.

Test maze 01 results

trail	move1	move2	path length	score
1	171	17	30	22.733
2	169	17	30	22.667
3	169	17	30	22.667
4	169	17	30	22.667
5	167	17	30	22.600
6	167	17	30	22.600
7	167	17	30	22.600
8	175	17	30	22.867
9	169	17	30	22.667
10	171	17	30	22.733
average:	169.4	17	30	22.680

Test maze 02 results

trail	move1	move2	path length	score
1	311	22	43	32.400
2	308	22	43	32.300
3	300	22	43	32.033
4	321	22	43	32.733
5	277	22	43	31.267
6	301	22	43	32.667
7	291	22	43	31.733
8	319	22	43	32.667
9	303	22	43	32.133
10	314	22	43	32.500
average:	304.5	22	43	32.183

Test maze 03 results

trail	move1	move2	path length	score
1	377	25	49	37.600
2	353	25	49	36.800
3	371	25	49	37.400
4	356	25	49	36.900
5	369	25	49	37.333
6	361	25	49	37.067
7	373	25	49	37.467
8	379	25	49	37.667
9	357	25	49	37.933
10	395	25	49	37.200
average:	369.1	25	49	37.337

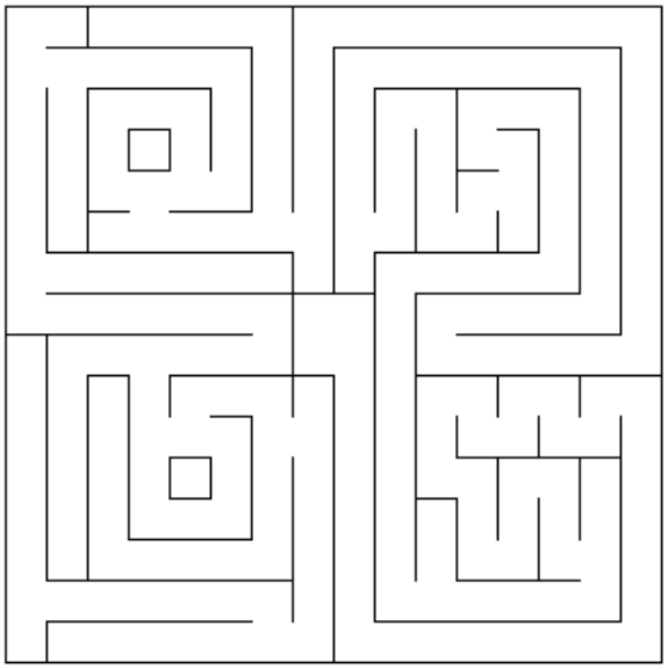
Justification

From the above three tables we can see that this robot is very stable on solving this problem. It can always find the optimal path and moves. So the only thing that affect its final score is its number of moves in the first run.

Conclusion

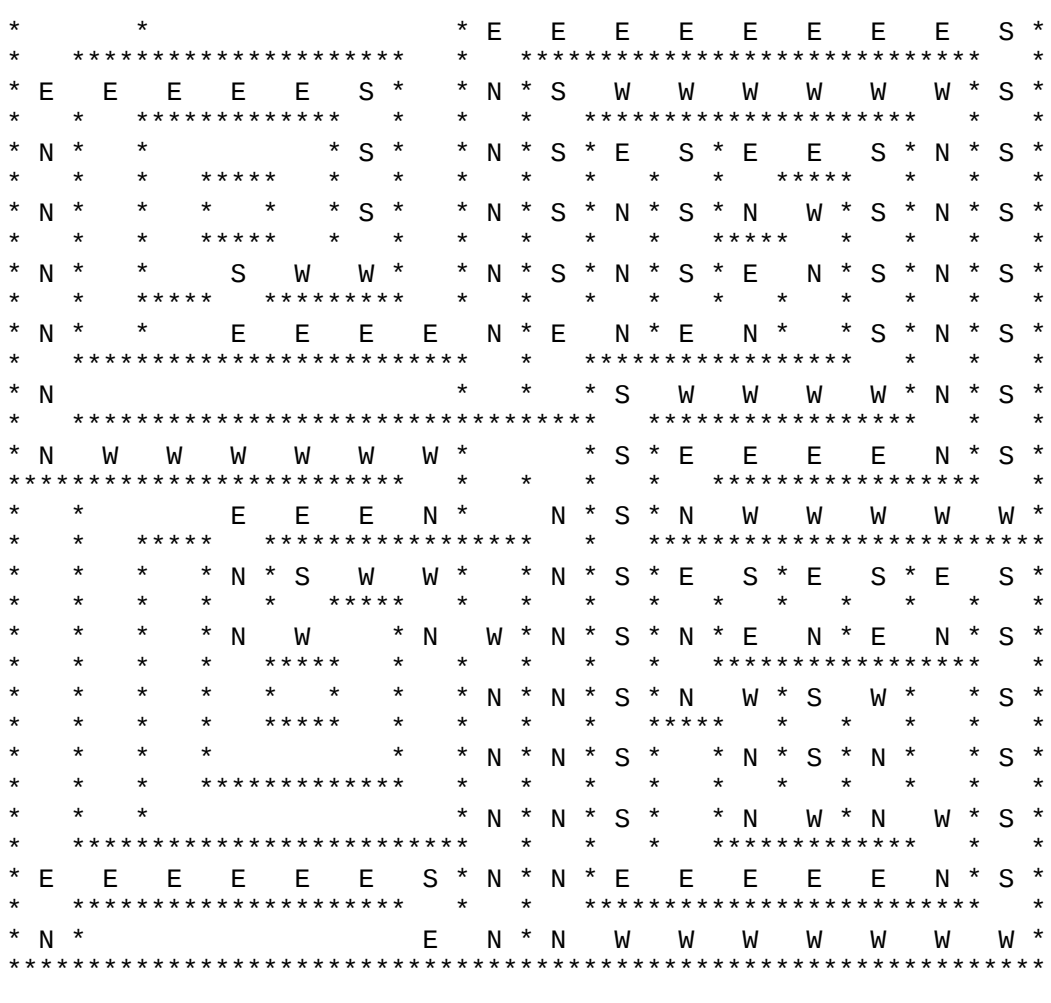
Free-Form Visualization

My maze is shown below. To get to the central goal room in maze 4, the robot need to pass most of the area of the maze. The structure is shown as below.



The optimal path and moves for this maze is shown below.

Path length: 179



Number of moves: 91

```
(  0, 1), ( 90, 3), (  0, 3), ( 90, 1), (-90, 1),
(-90, 3), (  0, 2), (-90, 1), ( 90, 1), (-90, 2),
(-90, 1), ( 90, 1), ( 90, 2), ( 90, 3), (-90, 1),
(-90, 3), (  0, 3), ( 90, 3), (  0, 3), ( 90, 3),
(  0, 2), ( 90, 3), ( 90, 2), (-90, 1), (-90, 3),
(  0, 1), (-90, 3), (  0, 2), ( 90, 3), (  0, 3),
(  0, 2), ( 90, 3), (  0, 3), (  0, 2), ( 90, 3),
(  0, 2), ( 90, 1), ( 90, 3), (  0, 1), (-90, 3),
(  0, 3), (-90, 3), (  0, 3), (-90, 3), (  0, 1),
(-90, 1), (-90, 3), ( 90, 1), ( 90, 3), (-90, 1),
(-90, 1), ( 90, 1), (-90, 1), (-90, 1), ( 90, 1),
( 90, 2), ( 90, 3), (  0, 1), ( 90, 3), (  0, 1),
(-90, 3), (  0, 3), (  0, 2), (-90, 3), (  0, 2),
(-90, 1), (-90, 1), ( 90, 2), (-90, 1), (-90, 2),
( 90, 1), ( 90, 2), (-90, 1), ( 90, 2), ( 90, 1),
( 90, 1), (-90, 1), (-90, 1), ( 90, 1), ( 90, 1),
(-90, 1), (-90, 1), ( 90, 1), ( 90, 3), (  0, 3),
( 90, 3), (  0, 3), (  0, 1), ( 90, 3), (  0, 3),
(  0, 1)
```

Test maze 04 results

trail	move1	move2	path length	score
1	355	91	179	102.867

Unlike the other three mazes, the robot can get to the goal room by just using a left-wall-follower strategy in the first run. So the phase two is automatically skipped by the robot. It directly enter phase three to get a 100% map coverage.

Reflection

The first challenge of this project to divide the whole problem into many small pieces. Unlike other projects before, in this capstone project, there is no clear thought about what I need to do or what steps I can take to finally solve the problem at the first place. The whole problem consists of many different aspects, like, how to explore the maze as fast as possible, how to find the best route, how to record what the robot have learned from the sensors. I must think all of them through and step by step build the robot. And I have to make sure all my codes are correct before including some new features in the program, otherwise the debugging will be a nightmare.

Another challenge is to implement 'known_maze, coverage' update mechanisms. Since all the other strategies or algorithms are based these two essential variables, so if there are some bugs in these parts, the robot will have some very strange behaviors. And that was what happened during the early stage of this project. Besides, the way we use binary number to record wall information also needs to be carefully dealt with.

Improvement

In a real Micromouse competition, everything is in a continuous domain rather than a perfect virtual world. There everything has errors. The sensors mounted on the robot won't know exactly how many cells are open. Instead, it will only measure a distance value to any object ahead of the sensors. Additionally, the rotation and movement of the robot is no longer perfect. They are both continuous values and the robot can make mistakes. The accumulation of small errors can result in robot's losing its accuracy on exploring the maze. The robot speed also needs to be controlled. A fast movement can give the robot a higher score, but it can also increase the probability to make mistakes. In real world we don't deal with unit time, but real time. To get a higher score, challengers must try to shorten the time needed for every action the robot take. Of course, to build a real Micromouse that can solve a maze, it is far more complicated than a virtual one. A real robot needs to be built. Chips, motors, sensors, memory, wheels, all of these must be tweaked to have a nice performance. There is just so many things to consider and we can't cover all of them here.