Machine Learning Capstone Project
Robot Motion Planning

By: Kambiz Mir
February 20, 2018

I. Definition

Project Overview

This project is a simplified software simulation of Micromouse competitions began in 1970s, in which a robot mouse tries to find a path from the corner of a maze to its center. The robot mouse may make multiple runs in a given maze. In the first run, it tries to learn the maze map as much as possible. In the subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, using what it has learned in the first run.

There are Micromouse competitions in several countries every year. In a real maze, the specifications for the maze and robot are described exactly. Performance in recent years has improved considerably. There are also software simulators for Micromouse which try to solve it in different ways and using variety of algorithms. Here are some useful links for problem definition, simulation and a project from Udacity students:

https://en.wikipedia.org/wiki/Micromouse
https://github.com/bblodget/MicromouseSim
https://github.com/JustinHeaton/Maze-Navigation

Problem Statement

For this project, a simplified model of the world is provided by Udacity. There will be 2 runs. The goal is to write a program to explore the world in first run and then find the fastest path to the center in second run.

The maze exists on an n x n grid of squares, n even. The minimum value of n is 12, the maximum 16. The starting square will always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the center of the grid is the goal room consisting of a 2 x 2 square; the robot must make it to the center from its starting square in order to register a successful run 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. 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.

Metrics

Evaluation is determined by definition of the problem. Based on the definition of the project, 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.

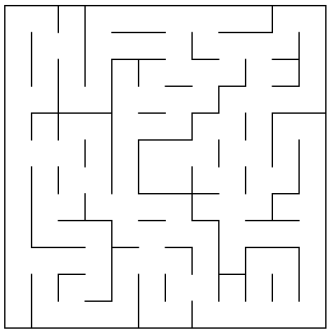
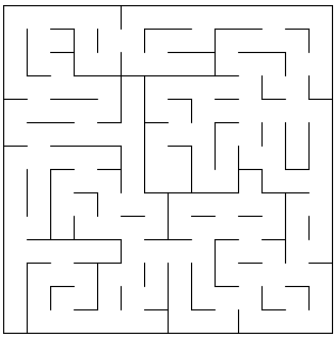
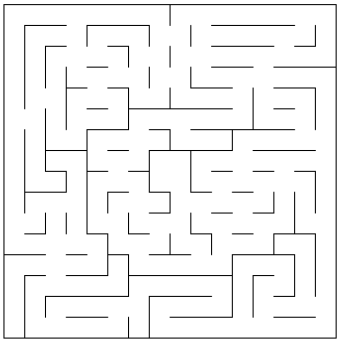
score = second_run_steps + (first_run_steps)/30

In first run, robot needs to discover the maze and will need many more steps than the second run which it really knows the maze map. So for calculating the score, it makes sense to normalize the first run steps with some factor which is dividing by 30 in this case.

II. Analysis

Data Exploration and Visualization

3 Mazes of sizes 12, 14 and 16 are provided for the project:

Here's some initial observations from the project mazes:

-All mazes have dead ends.
-All mazes have loops.
-All squares are reachable from other squares.
-It is possible to create new mazes following the file format described in project description.
-We assume that all the maze squares will be reachable for this project.

The robot can detect the number of open squares on left, front and right direction and it can move 1,2 or 3 squares in each time step. it will not move if it hits a wall. So although the target can be X squares far from the source, it may be possible to get to the target in less than X steps. For example in first maze, if we trace it manually to find a path from start to the center, on possible path is: north 2, east 1, south 2, east 3, north 2, east 1, north 1, east 2,south 3, east 1, east 3, north 3, west 3, north 2, west 1, north 1, west 1. Although there are 32 squares but the robot can go that path in 17 time steps.

Algorithms and Techniques

In first run, the robot has no knowledge of the world other than its current location and coordinates of the center. The objective in first run is to traverse the maze and gather as much data as possible by moving and sensing. The more knowledge we can collect from the environment, the higher will be the probability to find the shortest path in second run. Ideally if we can record the data for all squares in first run, we would be able to find the shortest path with guaranteed search algorithms like BFS, A* or Dijkstra. The strategy I'd like to try for first run is moving 1 square at a time in a Depth First Search (DFS) order for unknown squares and sense the walls and open edges for each square and record the observations. Also because robot can sense number of open squares in 3 directions, it is possible that we get the complete data for some of the squares without traversing all the squares.

DFS or Depth First Search is an algorithm for traversing or searching tree or graph data structures. It starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. DFS can be implemented recursively or using a stack to keep track of the nodes which have been visited. DFS guarantees to visit all nodes of a connected graph without going in a loop. DFS as the name suggests goes in each branch of the search deep until it can not proceed more because all the neighbor nodes are visited or it is a dead end, then it needs to backtrack and go to another branch.

By using DFS for 1st run, the robot starts from the initial square and goes to all squares and whenever can not go further it backtracks to previous branching square and takes another branch. This way robot can go to all the squares and discover all walls and no-walls and return to starting square.

Assuming that we find data for all squares in first run, for the second run we can use BFS, A* or Dijkstra to find the guaranteed shortest path. Before using one of these algorithms, we need to generate data structure to represent the graph of squares considering graph nodes when the robot moves 1, 2 or 3 squares and edges for the squares without a wall between them.

Having the undirected unweighted graph, all 3 searches are guaranteed to find the shortest path. There are differences in their time and implementation complexity, performance etc. which makes one better than others depending on problem at hand.

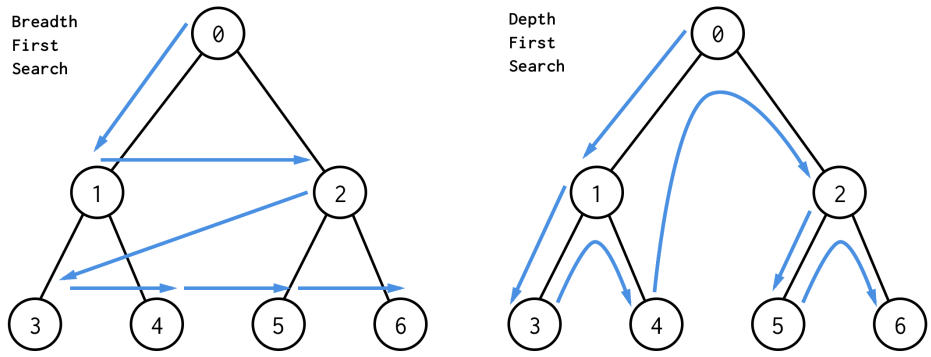
BFS traverses the graph in layers and can find all the node which are the same distance from starting node. The weights for the edges are usually considered as one. The implementation is simple but it will use large memory for large graphs because it needs to keep track of all the nodes which are the same distance from the starting node.

A* is more general than Dijkstra. It is more efficient than BFS in general because it expands on a node only if it seems promising. A* needs a admissible heuristic to guarantee finding the shortest path. In our maze case, one heuristic function can be the direct distance from the robot position to the center of the maze which always will be smaller than number of steps needed to get to the center.

Dijkstra works well for weighted graphs. When all weights are equal, it works similar to BFS.

Between A* and BFS, A* can do better in general but because BFS implementation is very simple and the graphs we have are not large I picked **BFS** for shortest path search.

The following image shows the order of traversal for DFS and BFS for a simple tree, starting from the root:



source: http://mishadoff.com/blog/dfs-on-binary-tree-array/

Benchmark

The extreme simple case can be a maze without any walls in which the first run takes 2*(n-1) steps to discover the world and n-1 steps to get to the center in second run. The total steps will be 3*(n-1) and the total score will be (n-1)+(n-1)/15.
For a complex maze the first run can have many backtracks and can get close to $2n^2$. For the second run the worst shortest path can be close to n^2. The total steps will be $3n^2$ and The approximate worst case total score will be $n^2 + n^2/15$.

For n=16 the total steps is expected to be between 45 and 768 and the total score is expected to be between 16 and 273.

Graphics

It will be difficult to test and debug the code without having visual feedback. We also will use Turtle graphics library for python to draw the discovered walls in 1st run and the shortest path on 2nd run

III. Methodology

Implementation

The main function in ***robot.py*** which is called in each time step, is named ***next_move***. The sensors data will be passed to this function and the function should update its state variables

and calculate ***rotation*** and ***movement*** variables as return values. ***location*** and ***heading*** are the state variables to keep track of location and heading of the robot at any time.

1-First run

In first run we want to search the maze using DFS. To do that, we need to keep track of ***borders*** we have passed in order not to pass them more than once, unless we want to backtrack. Between each 2 square we consider a ***border*** which can be a ***wall*** or ***no-wall***. Python dictionary ***passedBorders*** is used to keep track of passed borders.

To avoid loops, we need to make sure that robot doesn't go to squares which has been there before. We use Python dictionary ***visitedCells*** to keep track of cells which have been visited before.

When robot gets to a square which there is no next square to go, it need to backtrack. To keep track of visited cells in order, we use Python list ***cellsStack*** to keep the order of the squares traversed so far.

The description for DFS in 1st run is like this:

*If there is an available open square **and** the square is not visited before, **then***
update state variables location, heading, passedBorders, visitedCells, cellsStack
update return variables rotation and movement to get to next square
else:
get the previous square from cellsStack
update state variables location, heading, passedBorders, visitedCells, cellsStack
update return variables rotation and movement to get to previous square

return rotation, movement

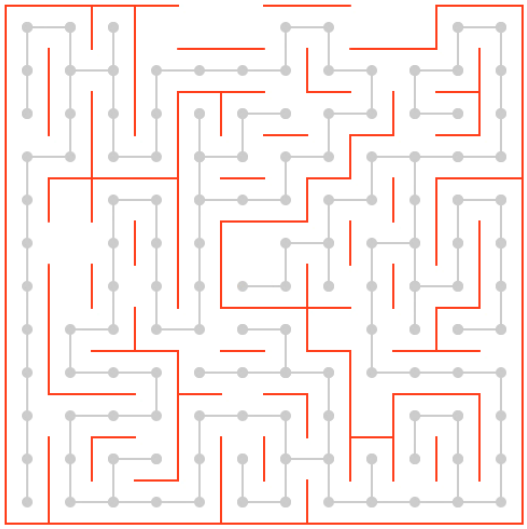
To implement in Python, each square is assigned a code which is combination of its row and column. Each border between squares is assigned a code which is a combination of codes of source square and target square.

For example square at row 1 and column 2 has the code 0102 and the border between 0102 and 0103 has the code 01020103.

With DFS implemented and moving one square at each time step, we make sure that robot will discover all the borders, wall or no-wall and also it doesn't fall into loops. If there are more than one next available moves, robot selects one randomly.

Also as the robot discovers new walls and updates the state variables to keep track of the walls and path, the walls are drawn using Turtle graphics.

Here's a snapshot of robot traversing maze 1:



Considering that the final score will use number of discovery steps divided by 30, discovering more from maze in first run increases the chance of finding the shortest path in second run. In extreme case, discovering all squares and borders in first run guarantees that we can find the shortest path using the right algorithm in second run.

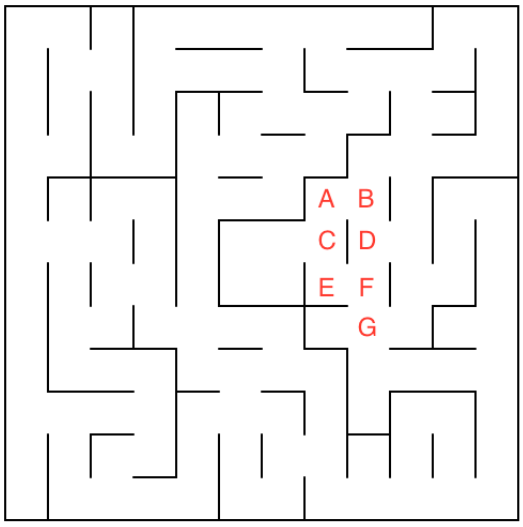
There is an easy way to stop the search earlier than robot returns to starting square. We store the code for all borders in **allBordersData** dictionary variable and whenever a border is discovered we'll remove it from this dictionary. When the dictionary gets empty, we would know all the borders are discovered although robot may not have gone to all squares and has not backtracked to initial position. This saves the robot some steps in first run.

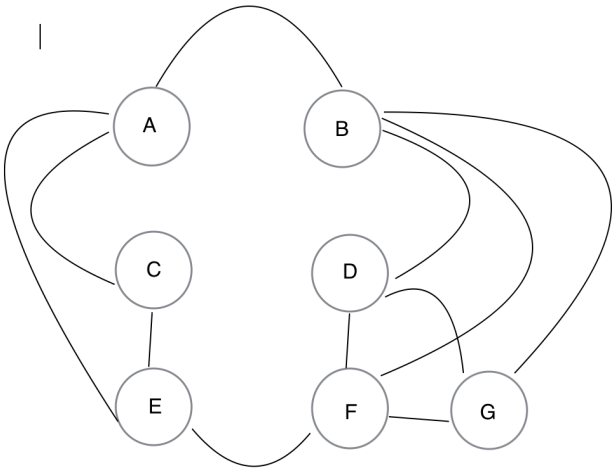
There is another way of saving some steps in first run by moving 2 or 3 steps whenever possible. This will be discussed in possible improvement section.

2-Building the graph for shortest path search

In second run we want to search for the shortest path. To do so we need to build a graph to represent the maze using the data we have discovered in first run. Tho nodes of this graph will be the squares and there will be an edge between 2 nodes if there is a possible move between the nodes. The move can be 1,2 or 3 steps.

For example for squares shown in the image below, the corresponding partial graph is drawn:





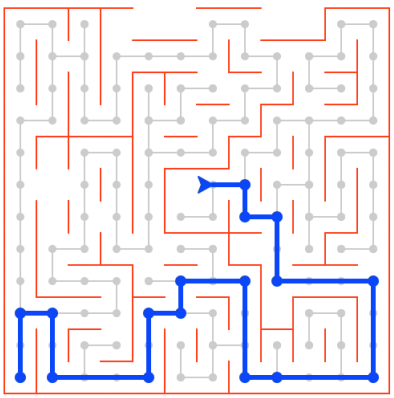
The **buildGraph** function iterates over all squares and finds out if there is a wall between squares and if they have an edge between them or not. it creates the dictionary **cellsGraph** which is the adjacency dictionary representation for the graph we want to search.

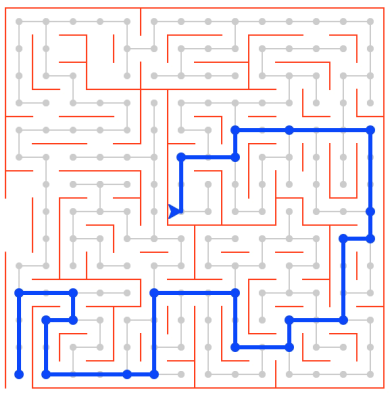
3-Search for the shortest path

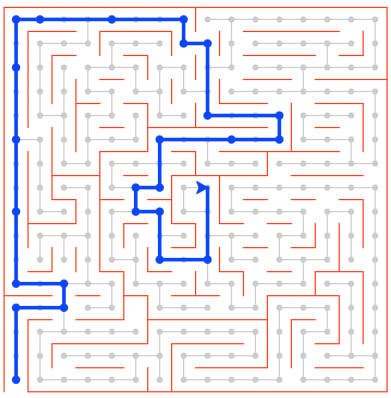
Having the graph from the previous parts, we need to find the shortest path from initial position to the center of the maze.

Function **findShortestPath** implements BFS using a queue until it finds center of the maze in the graph and then finds the reverse path from target to source and reverts it to get the shortest path.

After finding the shortest path, the shortest path will be drawn using Turtle graphics. Here's the images of the shortest paths for 3 mazes:







4-Complications

To implement DFS search and then creating the maze graph and doing BFS search on it, I needed to create a data structure to represent the squares and borders between squares. I assigned a string code to each square which is combination of row and column number. At first I created the code by combining the row and column numbers without “zero fill”. As a result for example the code for square at row 1 and column 12 had the same code as the square at row 11 and column 2. In other words the codes were not unique. It took some time to find the reason and once I used “zfill” function, it worked properly.

The other difficult task was determining next heading based on the current heading and available next moves. Although it is not complicated, but it is very error probe and confusing and one simple mistake causes the code not work properly and it is difficult to find the exact reason for the problem.

Refinement

-Originally I started coding and looking at print statements to debug my code but soon realized it would be very time consuming and difficult to continue without some graphics feedback, so I added graphics representation of the maze which although needed some extra work but it helped a lot to make the code work right.

-While coding for DFS, the conditions to detect if a move is available to the left, right or front was similar to this:

if rightOpen>0 and (not rightBorderCode in self.passedBorders) and (not rightCellCode in self.visitedCells):

Later I realized that the “(not rightBorderCode in self.passedBorders)” part of the if statement is extra, because if the right cell is not visited, then the right border cell is not passed for sure. So I refined some of the if statements in the code, removing unnecessary conditions.

-Another situation was when robot needs to backtrack and needs to rotate 180 degrees. Initially I wrote a complicated function to handle this but later simplified it to 2 subsequent 90 degrees with movement = 0 and updating state variables accordingly.

-Also initially I didn’t have any mechanism for early stop the robot once it has discovered all the walls and the robot was always returning to the first square. I added the logic for robot to early stop 1st run once it discovers all borders even though it hasn’t been to all squares.

IV. Results

Model Evaluation and Validation

Considering that we do a DFS in 1st run until we create a complete map of the maze, our BFS can always find the shortest path in 2nd run and that will be a fixed number in final score . So using this strategy, what can change the final score is the number of steps in 1st run although it will be divided by 30 and will have less effect than 2nd run steps in general. In the following tables the results for 5 times in our 3 mazes is shown:

Maze 1:

	1st run steps	2nd run steps	Final score	1st run score ratio	(1st run steps)/(total squares) ratio
1	213	17	24.100	%29	%147
2	254	17	25.467	%33	%176
3	247	17	25.233	%33	%171
4	262	17	25.733	%34	%182
5	292	17	26.733	%36	%202
AVG	253.6	17	25.453	%33	%175.6

Maze 2:

	1st run steps	2nd run steps	Final score	1st run score ratio	(1st run steps)/(total squares) ratio
1	404	22	35.467	%38	%206
2	366	22	34.200	%36	%187
3	387	22	34.900	%37	%197
4	404	22	35.467	%38	%206
5	366	22	34.200	%36	%187
AVG	385.4	22	34.84	%37	%196

Maze 3:

	1st run steps	2nd run steps	Final score	1st run score ratio	(1st run steps)/(total squares) ratio
1	526	25	42.533	%41	%205
2	466	25	40.533	%38	%182
3	463	25	40.433	%38	%180
4	515	25	42.167	%41	%201
5	521	25	42.367	%41	%203
AVG	498.2	25	41.61	%39.8	%194.2

Justification

Looking at the numbers, it can be seen that as the maze gets larger, the 1st run plays a larger role in final score which makes sense since larger maze requires more backtracks. Because the 2nd run has fixed number of states, the only way to improve the score is to decrease the 1st run steps when possible.

For all 3 mazes, the number of steps is smaller than 1000 which is required by project definition.

Also based on the benchmark, the total steps and the total score higher and lower limits for 3 mazes should be:

Maze 1 12*12: 33<total steps<432 , 11.73<total score<153.6
Maze 2 14*14: 39<total steps<588 , 13.86<total score<209.1
Maze 3 16*16: 45<total steps<768 , 16<total score<273

From the tables above the total number of steps and total score higher and lower limits for 3 mazes are:

Maze 1 12*12: 240<total steps<309 , 24.1<total score<26.73
Maze 2 14*14: 388<total steps<426 , 34.2<total score<34.467
Maze 3 16*16: 488<total steps<491 , 40.433<total score<42.533

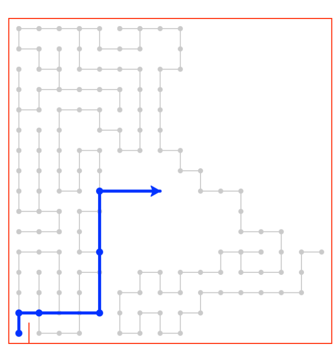
Comparing the test results and benchmark results shows that test results fall inside the benchmark results limits.

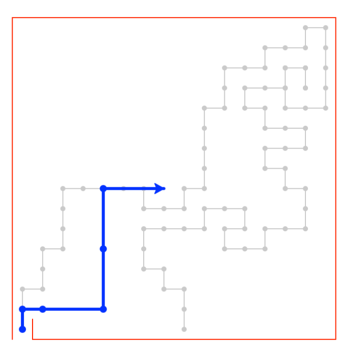
V. Conclusion

Free-Form Visualization

To verify how the program performs on other mazes with different pattern I created 2 extreme mazes and ran the program for them. The first is a maze with no walls. and the second is a maze with only one way to center and not many alternative ways to be discovered. The maze file and the result for 2 runs are shown below:

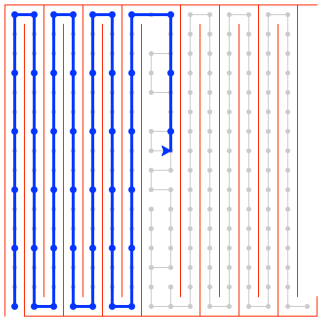
```
16
1,7,7,7,7,7,7,7,7,7,7,7,7,7,7,6
3,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
11,15,15,15,15,15,15,15,15,15,15,15,15,15,15,14
9,13,13,13,13,13,13,13,13,13,13,13,13,13,13,12
```

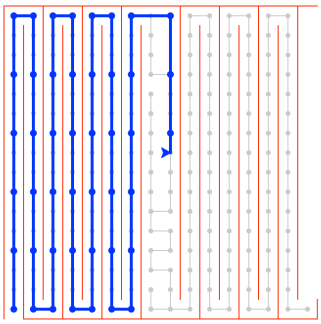




The shortest path has 6 steps. The score for 2 runs were 12.233 and 8.833. With fewer walls there is freedom in 1st run and there will be many backtracks and score for 1st run may vary a lot among different runs. Since there are fewer walls they will be discovered faster.

```
16
1,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,5,5,5,5,5,5,5,5,5,5,5,5,5,5,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,5,5,5,5,5,5,5,5,5,5,5,5,5,5,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,5,5,5,5,5,5,5,5,5,5,5,5,5,5,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,7,7,7,7,7,7,7,7,7,7,7,7,7,7,14
11,13,13,13,13,13,13,13,13,13,13,13,13,13,13,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,5,5,5,5,5,5,5,5,5,5,5,5,5,5,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,5,5,5,5,5,5,5,5,5,5,5,5,5,5,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6
3,5,5,5,5,5,5,5,5,5,5,5,5,5,5,12
9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,4
```





The shortest path has 45 steps. The score for 2 runs were 53.267 and 53.300. With many walls and fewer alternative paths, the score for 1st run does't vary much.

Reflection

The strategy I implemented is to guarantee finding the shortest path based on finding the complete map of the maze in 1st run. To discover the complete map of the maze I implemented DFS to traverse all the squares and record all the walls. I also added the logic for early stop once all the walls are found even though robot may not have been to all squares.

The next part was to extract the graph from the walls found in the previous part. I created a simple adjacency dictionary representation for the graph.The output of this part is a graph representation of the squares which will be used by next part to find the shortest path.

Given the graph representation of the maze from the 1st run, I used BFS to find the shortest path from the initial square to any of the center squares.

Although this method solves the problem, there can be other techniques to improve the score. The score for the 2nd run is fixed but the score for the 1st run can be improved with some improvements in the search.

Searching on internet, I found out this project has been done by many people using different search algorithms and some have better score result than my version. The most interesting and difficult part of the project to me was the initial design part and deciding what strategy and algorithms to choose for the project.

Improvement

To decrease the number of steps in 1st run, the DFS algorithm should be changed to take 2 or 3 steps when possible. When the direction of the next move is determined, the algorithm should check if by moving 1 step to the next square, it is going to discover any new information. If it finds out no new information will be discovered it can examine for moving 2 steps and 3 steps too and take the smallest steps in which it may find new borders information. This can save some steps in backtrack moves and even in forward moves.

Also if the program wants to be used on a real Micromouse with limited resources, the program should be optimized for memory and cpu cycles.

Videos

The links to videos for graphics implementation of sample mazes of the project for 1st and 2nd runs are provided below:

https://youtu.be/4n30CwDy8oU
https://youtu.be/c8expDQjmvE
https://youtu.be/cQ2Q7VrHhhE