Webots Maze Searcher Robot

Parsa Ramezani
Amirkabir University Of Technology

May 10, 2024

Introduction

Project Objectives

- Implementation of two search algorithms :
  Depth-First Search (DFS)
  Breadth-First Search (BFS)
- Successfully navigate and map the entire maze.
- Demonstrate the chosen traversal algorithm's effectiveness in real-time simulation.
- Execute the traversal with minimal errors and without manual intervention.
- Return to the start point of the maze.

Problem-Solving Approach

Lets first define a concept which has been used to solve this problem easier.

Node

Represents every tile of the maze.
Every node is constructed and defined using its surrounding walls.
Every node has a path which relates the node to a reference node (mostly called the **root**)
Nodes are related to each other using a parent-child connection thus forming a hierarchy like a tree.

With the help of this concept the problem of searching and mapping the maze will be represented by the problem of searching and growing nodes.

The Main Process

In this part we will go throw the code part by part and give a clear explanation about each part and its functionality in the whole program.

Some Important Lines

Basics

```
 1 from controller import Robot
 2 
 3 # create the Robot instance .
 4 robot = Robot ()
 5 
 6 # get the time step of the current world .
 7 timestep = int( robot.getBasicTimeStep ())
 8 MAX_SPEED = 6.28
 9 
10 def delay(ms):
11   initTime = robot.getTime()      # Store starting time (in seconds)
12 while robot.step(timestep) != -1:
13   if (robot.getTime() - initTime) * 1000.0 > ms: # If time elapsed (converted into ms) is greater than
      value passed in
14     break
15 
16 ps = []
17 psNames = [
18 'ps0', 'ps1', 'ps2', 'ps3',
19 'ps4', 'ps5', 'ps6', 'ps7'
20 ]
21 
22 for i in range(8):
23   ps.append(robot.getDevice(psNames[i]))
24   ps[i].enable(timestep)
25 
26 leftMotor = robot.getDevice('left wheel motor')
27 rightMotor = robot.getDevice('right wheel motor')
28 leftMotor.setPosition(float('inf'))
29 rightMotor.setPosition(float('inf'))
30 leftMotor.setVelocity(0 * MAX_SPEED)
31 rightMotor.setVelocity(0 * MAX_SPEED)
32 Tile_block = 120
```

Description The first 14 lines have been given in the base code template of the project and simply does some importing and defines the delay function which is used throughout the whole program, giving the robot mobility and dynamism.
In the next lines we have listed all the available distance sensors of the robot, enabling and appending them to the ps list which holds the active sensor objects, to be used in the wall detection process.
Then, we have declared two variables `leftMotor` and `rightMotor` from the robot's object to access the corresponding motors, then their positions and velocity have been set to a safe value to ensure a fresh and correct starting point for the robot. Finally, we have set the tile block to 120 which is approximately the amount of delay needed for the robot to completely pass one tile.

Useful Functions

Reading Sensor Values

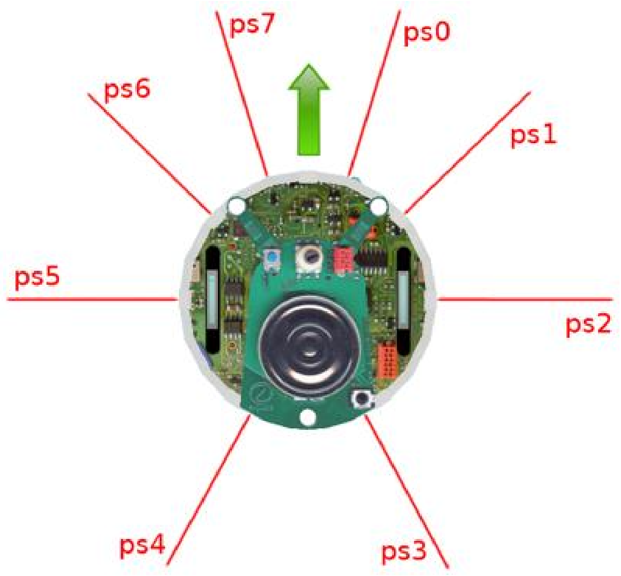
```python
def read_ps_values():
  rst = []
  for i in range(8):
  rst.append(ps[i].getValue())
  return rst
```

Description This function reads distance sensor values and returns them in form of a list

Detecting Wall

```python
def detect_wall():
  THRESHOLD = 90
  ps_values = read_ps_values()
  is_wall = ps_values[7] > THRESHOLD or ps_values[0] > THRESHOLD
  return is_wall
```

Description This function reads the sensor values and detects if any wall is in front of the robot by comparing the front-left and front-right sensor values to a predefined threshold (the location of these sensors are illustrated below). Finally, it returns True if any wall has been detected else, False.



Measuring The Wall Distance

```python
def measureTheWallDistance():
  distance_measured = 0
  is_any_wall = True
  HALF_TILE_WIDTH = 100 # half width of a tile
  while(not detect_wall()):
    if distance_measured > HALF_TILE_WIDTH:
      is_any_wall = False
      break
    leftMotor.setVelocity(0.5 * MAX_SPEED)
    rightMotor.setVelocity(0.5 * MAX_SPEED)
    delay(1)
    distance_measured += 1
  leftMotor.setVelocity(0 * MAX_SPEED)
  rightMotor.setVelocity(0 * MAX_SPEED)
  return is_any_wall, distance_measured
```

Description This function simply counts the amount of delay needed for the robot to reach the wall (without hitting it) and At the end it returns a flag and a number in form of a tuple:

flag

False if one Tile Width has been passed and no wall has been detected.

True if any wall has been detected.

distance

If any wall exists returns the wall distance from the robots position.

Otherwise returns the Tile Width which proves no wall has been detected.

Rotating The Robot

```python
# to rotate the robot dg degrees + for right and - for negative
DelayPerDegree = (750)/90
def rotate(dg) :
  if dg < 0:
    # turn left
    lVelocity = -0.5
    rVelocity = 0.5
  else:
    # turn right
    lVelocity = 0.5
    rVelocity = -0.5
  leftMotor.setVelocity(lVelocity * MAX_SPEED)
  rightMotor.setVelocity(rVelocity * MAX_SPEED)
  delay(DelayPerDegree * abs(dg))
  leftMotor.setVelocity(0 * MAX_SPEED)
  rightMotor.setVelocity(0 * MAX_SPEED)
```

Description This function rotates the robot in degrees, clockwise for positive degrees and counterclockwise for negative degrees.
Please Note that, the role of `DelayPerDegree` is crucial here. The Robot's rotation will be always done as a fraction of this unit, meaning that 90 degrees of rotation is equivalent to $90[\text{degree}] * \texttt{DelayPerDegree} = 90 * \frac{750}{90} = 750[\text{delay}]$. This delay is equivalent to approximately 90 degree of rotation when the left and right motors are set to 50% of their maximum speed.
At the end, the motor velocities were set to 0 to ensure that this function would never affect the main flow of the program.

Returning a Distance

```python
def turnBack(distance):
  leftMotor.setVelocity(-0.5 * MAX_SPEED)
  rightMotor.setVelocity(-0.5 * MAX_SPEED)
  for i in range(distance):
    delay(1)

  leftMotor.setVelocity(0 * MAX_SPEED)
  rightMotor.setVelocity(0 * MAX_SPEED)
```

Description This function reverses the motor's velocities making the robot to return the given amount of distance in units of delay.

Going A Straight Distance

```python
def goStraight(distance):
  leftMotor.setVelocity(0.5 * MAX_SPEED)
  rightMotor.setVelocity(0.5 * MAX_SPEED)
  for i in range(distance):
    delay(1)

  leftMotor.setVelocity(0 * MAX_SPEED)
  rightMotor.setVelocity(0 * MAX_SPEED)
```

Description This function does exactly the same as what `turnBack` function does yet in opposite direction.

Move Forward Based on Steps

```python
def moveForwardStepByStep(steps):
  for step in steps:
    if step == "left":
      rotate(-90)
    elif step == "right":
      rotate(90)
  goStraight(Tile_block)
```

Description This function would make the robot move based on given steps for one tile block in units of delay. For example, ”left, right, front, left” makes the robot to:

1. Rotate left and keep going for one Tile Block
2. Rotate right and keep going for one Tile Block
3. Do not rotate and keep going for one Tile Block
4. Rotate left and keep going for one Tile Block

We will see how this function makes the robot’s transitions between nodes easy by using their absolute path.

Return Backward Based on Steps

```python
def moveBackwardsStepByStep(steps):
  steps.reverse()
  for step in steps:
    turnBack(Tile_block)
    if step == "left":
      rotate(90)
    elif step == "right":
      rotate(-90)
```

Description This function works exactly same as `moveForwardStepByStep` but instead in the reverse order. The following operations are done in exactly same order:

1. Reverse the order of steps and go throw the steps, one by one
2. Return distance of exactly one Tile Block
3. Do the opposite rotation
   - If we have previously rotated left then rotate right
   - If we have previously rotated right then rotate left
   - Otherwise do not make any rotations
4. Do this for all the steps

Node Class

Here we give a complete description about the `Node` class and all its corresponding methods defined in it.

The Constructor

At first, we will define the constructor of this class.

```python
class Node:
  def __init__(self, parent = None) :
    flag_f, wall_distance_f = measureTheWallDistance()
    turnBack(wall_distance_f)
    rotate(90)
    flag_r, wall_distance_r = measureTheWallDistance()
    turnBack(wall_distance_r)
    rotate(-90)
    rotate(-90)
    flag_l, wall_distance_l = measureTheWallDistance()
    turnBack(wall_distance_l)
    rotate(90)
    self.Walls = {"front":(flag_f, wall_distance_f),"right": (flag_r, wall_distance_r), "left":(flag_l,
    wall_distance_l)}
    self.parent = parent
    self.path = []
    self.children = []
```

Description In the Node class constructor we can pass the parent node as an argument to the function otherwise, the created node won't have any parent or its parent will be set to `None` by default.
As explained in the introduction, nodes are a representation for tiles of the maze and therefore, when creating a new node object, it has to detect all the walls around that tile and put a flag in directions in which any obstacle including a wall has been detected. Furthermore, it counts the amount of distance between the robot's position and the wall in units of delay. All these data gathered from the walls will be stored in a dictionary with keys representing directions and the data of wall detection process stored in its value as a tuple in the following format `(<<flag>> , <<distance from the wall>>)`

flag Indicates if any wall exists in that direction.

distance from the wall The corresponding delay needed to get to a wall in that direction.

Challenge There is an error in the process of rotation in units of delay, meaning that 750 ms of delay is not exactly equal to 90 degree of rotation. If this error does not get canceled out, it will result in an undesirable rotation angle which is no longer the angle we wanted which might lead into incorrect wall detection.

Solution The amount of error due to this rotation process is controlled by rotating the robot in the *reverse* direction, meaning that after all the rotations are done the robot should return to its original angle and orientation as before, to ensure that this error will not exceed considerably
Please note that, there would always be an increasing amount of deviation from the correct angle and this error is proportional to the size of maze (the larger the maze is the more rotation stages are needed to follow a path) thus the great use of controller is obvious here.

The Growing Algorithm

```python
# Node class ...
def grow(self):
  walls = self.Walls
  new_nodes = []
  for key, value in walls.items():
    if not walls[key][0]: # if there is no wall in a position
      if key == "left":
        rotate(-90)
      elif key == "right":
        rotate(90)
      # lets goStraight to the new node's position
      goStraight(Tile_block)
      # create node by its surrounding walls
      new_node = Node(self) # setting this node as its parent node
      # append the history of it to all the new children
      for step in self.path:
        new_node.path.append(step)

      # adding the new node relative path from its parent(node) to its path
      new_node.path.append(key)
      # append the new node to the new nodes list:
      new_nodes.append(new_node)
      turnBack(Tile_block)
      if key == "left":
        rotate(90)
      elif key == "right":
        rotate(-90)
    else:
      new_nodes.append(None)

  self.children = new_nodes
```

Description This method simply does the steps below to grow a node and create children for that node:

1. Check if there is any available path in the node's wall dictionary if a wall exists put None in that corresponding direction of the node's children. For example if wall detection process yields to this result 'front': (False, 101), 'right': (True, 38), 'left': (True, 40) then only "front" position can be grown thus all the remaining directions should be set to None in the parent's children list. (meaning there is no child and no path there).
   - if there is no wall on left, rotate left.
   - if there is no wall on right, rotate right.
   - if there is no wall in front, do not rotate.
2. Go straight in that direction for one Tile Block.
3. Create a new node there.
4. Copy the parent's node path to its newly born child.
5. Add the new rotation to the `new node`'s path so that, the `new node`'s path is correct and valid.
6. Turn back to the previous position (parents node position).
7. Get to the parents node orientation.
8. Do the same for all the remaining directions.

At the end, the list *new nodes* contains children of the parent node and has to be set as the parent's children list.

Jumping Between Nodes

Challenge Using the Stack for DFS and Queue for BFS requires the robot to be able to change its position based on nodes so, for example in the DFS algorithm your robot should be able to jump from one node of a deeper depth to a node of a less deeper depth.

Solution The following method has been defined for Node objects to fulfill this necessity:

```
# Node class ...
def goToNode(self, destination_node):
  current_path = self.path
  destination_path = destination_node.path
  common_steps_index = min(len(destination_path), len(current_path)) - 1
  for i in range(min(len(destination_path), len(current_path))):
    if current_path[i] != destination_path[i]:
```

```
8        common_steps_index = i - 1
9        break
10   if common_steps_index == -1:
11     # one of the pathes were empty meaning root path
12     if not current_path:
13       # if the current path is root simply go straight to the destination path step by step
14       moveForwardStepByStep(destination_path)
15     else:
16       # destination is the root simply return all the steps from the current absolute path
17       moveBackwardsStepByStep(current_path)
18     return True
19   temp_list = []
20   for i in range(len(current_path)):
21     if i > common_steps_index:
22       temp_list.append(current_path[i])
23
24   moveBackwardsStepByStep(temp_list)
25
26   temp_list = []
27   for i in range(len(destination_path)):
28     if i > common_steps_index:
29       temp_list.append(destination_path[i])
30
31   moveForwardStepByStep(temp_list)
```

Description We will explain this function purpose line by line:

Line 2 The definition of the goToNode function which gets the `destination_node` apart from the `current node` (`self`)

Line 3 `current_node`'s path has been set to the `current_path` variable

Line 4 `destination_node`'s path has been set to the `destination_path` variable

Line 5 By default the minimum index of both of the two paths is set as the `common_step_index`

Line 6 - Line 9 A for loop would find the last common step in both of the absolute's paths

Line 10 - Line 18 Checks if any of the two path is empty, meaning that one of them is root path.
If the `current_path` is empty (root) simply go straight to the `destination_path` otherwise, the `destination_path` should have been empty (root) meaning that, we should simply return backward step by step to reach the root node.
In both of the situations the function should return True which shows it has done its job correctly.

Line 19 - Line 24 Creates a temporary list holding all the different steps needed to return backward where the two absolute paths differ.
And calls the `moveBackwardStepByStep` function on that temporary list.

Line 26 - 31 Does the exact same job as lines 19 - 24 yet making the robot go forward to reach the destination node.

Other Methods for Effective Functionality

```
# Node class ...
def get_children(self):
  return self.children

def console(self):
  rst = "Path from root: " + str(self.path) + "\n wall detection results: " + str(self.Walls)
  print(rst)
```

Description The `get_children` method simply returns the children list of every `node`
The `console` method makes the debugging easier by printing a well formatted message about each `node`.

Implementing Searching Algorithms

DFS

```
 1 def dfs(call_back_function):
 2   maze = list()
 3   stack = list()
 4   # detecting walls of the current location
 5   stack.insert(0,Node())
 6   while(stack):
 7     node = stack.pop(0)
 8     maze.append(node)
 9     call_back_function(node)
10     for child in node.get_children():
11       if child != None:
12         stack.insert(0, child)
13     if len(stack) > 0:
14       # checks if there exist a node which we have not already covered
15       # on the other hand checks if this is the last iteration
16       # because in the last iteration no next node exists at all!
17       next_node = stack[0]
18       node.goToNode(next_node)
19
20   return maze
```

Description First, the use of a call back function for future developments is important which gives users the ability to do whatever they want on each node based on their own needs. Now we will explain this search algorithm line by line.

Line 1 Definition of the dfs function.

Line 2 A list named `maze` has been created to get returned at the end of the function, holding all the nodes traversed during this search.

Line 3 A list named `stack` has been created to mimic the stack concept in the DFS algorithm.

Line 5 An initial node has been created to be the root node of the search.

Line 6 Opens a while loop which breaks when the stack becomes empty.

- **Line 7** One node is popped out of the stack and will be called `node`.
- **Line 8** `node` will get added to the maze list.
- **Line 9** `node` will be given to the call back function.
- **Line 10** A for loop will be created to insert all the valid children (not `None` children of `node`) to the top of stack.
- **Line 13** Checks if this iteration is not the last so that the robot's location should be changed for the next iteration by getting the next node of the stack and moving the robots position to this upcoming node.

Line 20 The search has finished so the `maze` list will be returned.

BFS

```
 1 def bfs(call_back_function):
 2   maze = list()
 3   queue = list()
 4   # detecting walls of the current location
 5   queue.append(Node())
 6   while(queue):
 7     node = queue.pop(0)
 8     call_back_function(node)
 9     maze.append(node)
10     for child in node.get_children():
11       if child != None:
```

```
12        queue.append(child)
13    if len(queue) > 0:
14      # checks if there exist a node which we have not already covered
15      # on the other hand checks if this is the last iteration
16      # because in the last iteration no next node exists at all!
17      next_node = queue[0]
18      node.goToNode(next_node)
19
20  return maze
```

Description First, the use of a call back function for future developments is important which gives users the ability to do whatever they want on each node based on their own needs. Now we will explain this search algorithm line by line.

Line 1 Definition of the bfs function.

Line 2 A list named `maze` has been created to get returned at the end of the function, holding all the nodes traversed during this search.

Line 3 A list named `queue` has been created to mimic the queue concept in the BFS algorithm.

Line 5 An initial node has been created to be the root node of the search.

Line 6 Opens a while loop which breaks when the queue becomes empty.

- **Line 7** One node is popped out of the queue and will be called `node`.
- **Line 8** `node` will get added to the maze list.
- **Line 9** `node` will be given to the call back function.
- **Line 10** A for loop will be created to insert all the valid children (not `None` children of `node`) to the beginning of the `queue`.
- **Line 13** Checks if this iteration is not the last so that the robots location should be changed for the next iteration by getting the next node of the queue and moving the robot's position to this upcoming node.

Line 20 The search has finished so the `maze` list will be returned.

Main Loop

```
1  def map(node):
2    node.grow()
3  flag = True
4  while robot.step(timestep) != -1:
5    if flag:
6      rotate(90)
7      Maze = bfs(map)
8      root = Maze[0]
9      last_node = Maze[-1]
10     last_node.goToNode(root)
11     flag = False
12     break
```

Description First, a simple function is defined that grows every node given to it. This gets used as the call back function for BFS and DFS.
Next a boolean called `flag` has been created to make sure that mapping the maze is only done once.
On line 6 we rotate the robot so that its head is to the open path, then call the dfs or bfs function with the given call back function to map the maze based on depth or breadth, respectively.
On lines 8 to 10 we make use of the newly Maze list created and change the location of the robot from the last node to the first (the root) by simply using the `goToNode()` method from the class `Node` as explained earlier.
Finally, we set the flag to `False` and break out of the loop.

Simulation Results

This chapter deals with the final results of the Maze mapping done by using the approach explained.

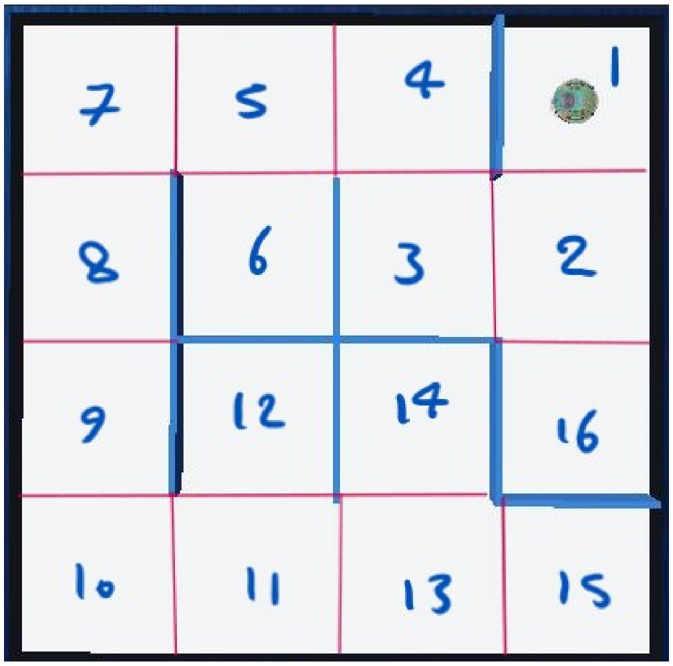
DFS Results

This is the results of printing all the nodes information traversed by the DFS algorithm.

1. Path from root: []
   wall detection results: ’front’: (False, 101), ’right’: (True, 38), ’left’: (True, 40)
2. Path from root: [’front’]
   wall detection results: ’front’: (False, 101), ’right’: (False, 101), ’left’: (True, 35)
3. Path from root: [’front’, ’right’]
   wall detection results: ’front’: (True, 47), ’right’: (False, 101), ’left’: (True, 35)
4. Path from root: [’front’, ’right’, ’right’]
   wall detection results: ’front’: (True, 45), ’right’: (True, 34), ’left’: (False, 101)
5. Path from root: [’front’, ’right’, ’right’, ’left’]
   wall detection results: ’front’: (False, 101), ’right’: (True, 43), ’left’: (False, 101)
6. Path from root: [’front’, ’right’, ’right’, ’left’, ’left’]
   wall detection results: ’front’: (True, 44), ’right’: (True, 51), ’left’: (True, 25)
7. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’]
   wall detection results: ’front’: (True, 55), ’right’: (True, 44), ’left’: (False, 101)
8. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’]
   wall detection results: ’front’: (False, 101), ’right’: (True, 54), ’left’: (True, 23)
9. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’]
   wall detection results: ’front’: (False, 101), ’right’: (True, 54), ’left’: (True, 23)
10. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’]
    wall detection results: ’front’: (True, 49), ’right’: (True, 60), ’left’: (False, 101)
11. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’]
    wall detection results: ’front’: (False, 101), ’right’: (True, 52), ’left’: (False, 101)
12. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’left’]
    wall detection results: ’front’: (True, 36), ’right’: (True, 24), ’left’: (True, 51)
13. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’front’]
    wall detection results: ’front’: (False, 101), ’right’: (True, 52), ’left’: (False, 101)
14. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’front’, ’left’]
    wall detection results: ’front’: (True, 32), ’right’: (True, 25), ’left’: (True, 50)
15. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’front’, ’front’]
    wall detection results: ’front’: (True, 30), ’right’: (True, 56), ’left’: (True, 21)

16. Path from root: [’front’, ’front’]
    wall detection results: ’front’: (True, 46), ’right’: (True, 49), ’left’: (True, 29)

The following picture may be helpful to better understand the results above:



BFS Results

1. Path from root: []
   wall detection results: ’front’: (False, 101), ’right’: (True, 38), ’left’: (True, 40)

2. Path from root: [’front’]
   wall detection results: ’front’: (False, 101), ’right’: (False, 101), ’left’: (True, 35)

3. Path from root: [’front’, ’front’]
   wall detection results: ’front’: (True, 46), ’right’: (True, 49), ’left’: (True, 29)

4. Path from root: [’front’, ’right’]
   wall detection results: ’front’: (True, 47), ’right’: (False, 101), ’left’: (True, 35)

5. Path from root: [’front’, ’right’, ’right’]
   wall detection results: ’front’: (True, 46), ’right’: (True, 35), ’left’: (False, 101)

6. Path from root: [’front’, ’right’, ’right’, ’left’]
   wall detection results: ’front’: (False, 101), ’right’: (True, 45), ’left’: (False, 101)

7. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’]
   wall detection results: ’front’: (True, 55), ’right’: (True, 45), ’left’: (False, 101)

8. Path from root: [’front’, ’right’, ’right’, ’left’, ’left’]
   wall detection results: ’front’: (True, 43), ’right’: (True, 50), ’left’: (True, 25)

9. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’]
   wall detection results: ’front’: (False, 101), ’right’: (True, 54), ’left’: (True, 23)

10. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’]
    wall detection results: ’front’: (False, 101), ’right’: (True, 54), ’left’: (True, 24)

11. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’]
wall detection results: ’front’: (True, 48), ’right’: (True, 59), ’left’: (False, 101)

12. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’]
wall detection results: ’front’: (False, 101), ’right’: (True, 50), ’left’: (False, 101)

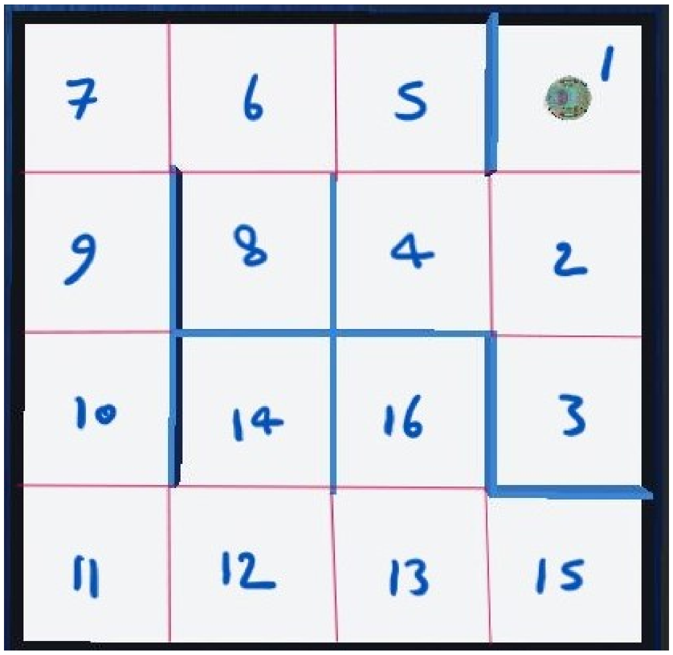
13. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’front’]
wall detection results: ’front’: (False, 101), ’right’: (True, 50), ’left’: (False, 101)

14. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’left’]
wall detection results: ’front’: (True, 38), ’right’: (True, 25), ’left’: (True, 51)

15. Path from root: [’front’, ’right’, ’right’, ’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’front’, ’front’]
wall detection results: ’front’: (True, 30), ’right’: (True, 57), ’left’: (True, 20)

16. Path from root: [’left’, ’front’, ’left’, ’front’, ’front’, ’left’, ’front’, ’left’, ’right’, ’right’, ’front’]
wall detection results: ’front’: (True, 31), ’right’: (True, 26), ’left’: (True, 49)

The following picture may be helpful to better understand the results above:


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