



# Maze Solving Robot

Line following maze solver

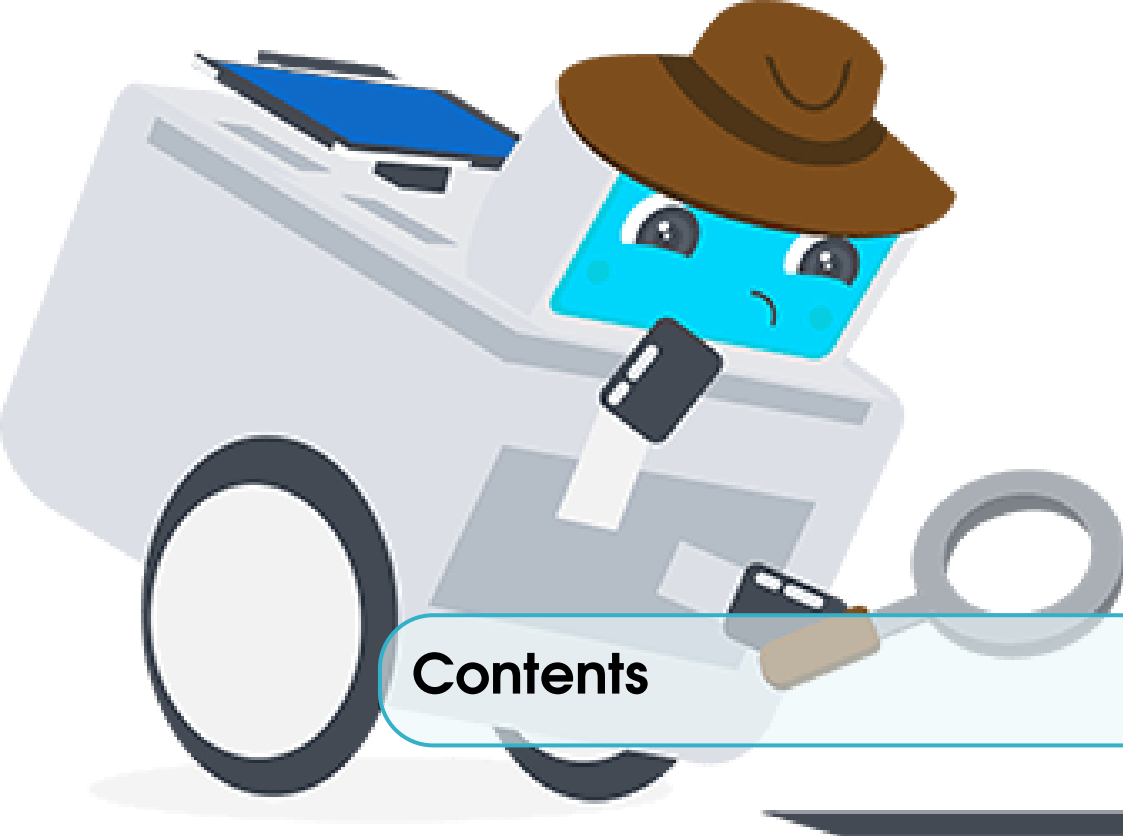
Akash Dhasade, Saket Joshi

INDIAN INSTITUTE OF TECHNOLOGY, TIRUPATI

[GITHUB.COM/](#)

The algorithm was developed as a part of preparation for the maze rover competition held at IIT Madras on January 1, 2017 by the members of team Neutrinos which stood first in the competition.

*First release, July 2017*



# Contents

<b>1</b>	<b>Introduction</b> .....	<b>5</b>
1.1	Problem Statement	5
1.2	Maze varieties and Existing Algorithms	6
1.3	How to read this book?	6
<b>2</b>	<b>Prerequisites for the Algorithm</b> .....	<b>7</b>
2.1	Getting the tools ready	7
2.2	Directions	8
2.2.1	Defining directions .....	8
2.2.2	Updating directions .....	8
2.3	The Coordinate System	9
2.4	Nodes	10
2.4.1	Node definition .....	10
2.4.2	Node characteristics .....	10
2.4.3	Matching Nodes .....	11
2.5	The Returning Variable	12
2.6	An Example	13
2.7	Conclusion	14
<b>3</b>	<b>The algorithm</b> .....	<b>15</b>
3.1	Running the algorithm	15

---

<b>3.2</b>	<b>The Optimal Path</b>	<b>23</b>
<b>3.3</b>	<b>Pseudo Code</b>	<b>23</b>
<b>4</b>	<b>Making the Bot</b> .....	<b>24</b>
<b>4.1</b>	<b>Gathering the materials</b>	<b>24</b>
<b>4.2</b>	<b>The bot design</b>	<b>24</b>
4.2.1	IR in-front design .....	24
4.2.2	IR-centered design .....	24
<b>4.3</b>	<b>Configuring the IR sensor array</b>	<b>24</b>
<b>4.4</b>	<b>Teaching bot to take turns</b>	<b>24</b>
4.4.1	LEFT-TURN and RIGHT-TURN .....	24
4.4.2	U-TURN .....	24

# 1. Introduction

## 1.1 Problem Statement

A line maze is usually a black line on a white background or it could also be a white line on a black background. But in this book, black lines on white background will be used. The problem of maze

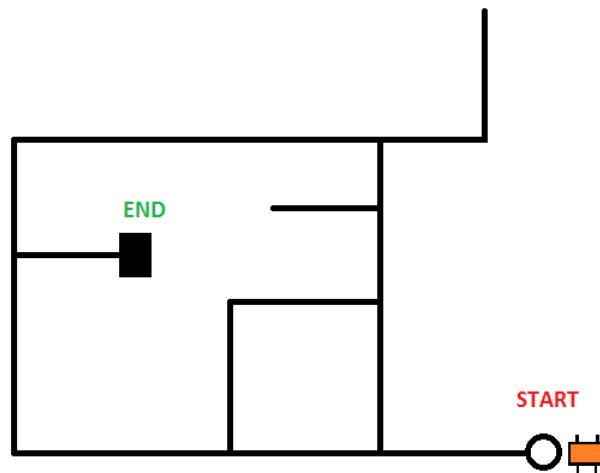


Figure 1.1: An example Maze having black lines on white background

solving comprises of two steps:

1. The first step is to drive through the maze and search for the finish where the bot starts from the marked start position. The finish is a characteristic black portion as shown in Figure 1.1.
2. In the second step, the bot is again placed at the start position, but this time the bot takes the optimal path to the finish avoiding dead ends and unnecessary paths.

## 1.2 Maze varieties and Existing Algorithms

The problem of maze solving is not as trivial as it seems and there have been many attempts to discover the best algorithms for solving mazes. But there is no such universal algorithm which guarantees the most optimal path though we have algorithms that are quite close to optimal. Graph traversing is an art in itself and we have DFS and BFS algorithms which are widely used for traversing graphs. Only difference between writing code for normal BFS or DFS and writing code for solving mazes is that in the earlier case graph is given as input whereas in the later we discover the graph as we traverse it in real time.

Though the problems are slightly different, the idea is still the same. The main crux lies in the looped and un-looped mazes. An instance of looped maze is shown in Figure 1.1. By loops we simple mean that the maze circles around some node, simply put there is a cycle in the maze. Interestingly when the mazes are not looped, we have very simple algorithms like LEFT-HAND-ON-THE-WALL which works in all cases and gives the optimal path. You can learn more about this algorithm following this link: <http://www.instructables.com/id/Maze-Solving-Robot/>. But as we said looped mazes are more complicated to solve. Our objective in this book will be to built a full fledged robot from scratch and understand and implement the algorithm which can solve looped mazes.

## 1.3 How to read this book?

Through out this book, we have made our best attempt to make you understand the algorithm. It is suggested to read the book on the order. We also provide exact code snippets alongside.

Chapter 1 introduced you to the problem of maze solving. In Chapter 2, we move to the prerequisites for the actual algorithm. We first give you a list of variables and methods that we define in our code and what each of them will be used for. We proceed with defining directions, understanding the calibration of coordinate system, then defining nodes, understanding the returning variable and finally we end with an example that illustrates each of the things defined above.

Chapter 3 puts forth the exact algorithm in a very unconditional way. Before stating the pseudo code, we do a run of algorithm on an example maze, show you how to take the optimal path and finally we state the pseudo code.

Chapter 4 is all about designing the bot. We start with the materials required and different bot designs. We then tell you about the configuration of IR sensor array and end the chapter different ways of taking turns.

With this, you will be a PRO in maze solving. We expect it to be an interesting journey, have fun!



## 2. Prerequisites for the Algorithm

### 2.1 Getting the tools ready

We define a whole bunch of variables and methods which will ease our way to the real algorithm. Here's the full list along with the explanation for each variable and function that we defined.

Declaration	Explanation
<code>cur_x</code>	Current x coordinate of the bot
<code>cur_y</code>	Current y coordinate of the bot
<code>returning</code>	Set to true when the path does not lead us to finish
<code>ptr</code>	One plus number of nodes visited and remembered till that point of time
<code>line</code>	Current position of bot on the line
<code>curr_time</code>	Holds the time returned by <code>millis()</code>
<code>prev_time</code>	Holds the time returned when <code>millis()</code> was previously called
<code>time_diff</code>	Holds difference between two time defined earlier
<code>bot_dir</code>	Current bot direction
<code>sensor[5]</code>	Holder for 5 pins of sensor
<code>turning</code>	Set to true when bot is turning, false otherwise
<code>junction_var</code>	Used to identify a junction
<code>left_path_var</code>	Used to identify the existence of a path in left direction
<code>right_path_var</code>	Used to identify the existence of a path in right direction
<code>blank_var</code>	Used to identify dead ends
<code>finish_var</code>	Used to identify finish point
<code>left_path</code>	Set to when a path in left direction exists
<code>right_path</code>	Set to when a path in right direction exists
<code>struct Node</code>	A data structure for node

Table 2.1: This table describes the methods and variables that will act as a tool kit for the algorithm. Column 1 gives declarations and Column 2 gives you the purpose of each variable or function.

Declaration	Explanation
<code>paths[4]</code>	Denotes existence of paths pertaining to a node
<code>vpaths[4]</code>	Keeps track of visited paths pertaining to a node
<code>back_path</code>	Holder for back path of a node
<code>x_cor</code>	x coordinate of node
<code>y_cor</code>	y coordinate of node
<code>kp, kd and ki</code>	Calibration variables for PID algorithm
<code>loop()</code>	Main loop of arduino code
<code>act()</code>	Method to take decisions at junctions, turns and dead ends
<code>node()</code>	Method called by <code>act()</code> to take decision when a junction is hit
<code>choose_path()</code>	Helper method to choose path at a node
<code>take_return_path()</code>	Method to turn in appropriate direction based on return path
<code>match_cur_node()</code>	Determine whether current node has been visited previously
<code>unvisited(ptr1, ptr2)</code>	Helper method to check existence of unvisited paths from ptr 1 to ptr 2
<code>get_coords()</code>	Updates <code>curr_x</code> and <code>curr_y</code>
<code>record_node</code>	Define and store a new node
<code>finish()</code>	Stops the bot when on finish point
<code>get_paths()</code>	Helper method to determine paths from bot position
<code>readLineandgetError()</code>	Updates line variable based on IR sensor values
<code>followline_pid()</code>	Implements PID algorithm
<code>brake()</code>	Defines brake for bot
<code>uturn()</code>	Defines the U-turn for the bot()
<code>turn_left()</code>	Defines the left-turn for the bot()
<code>turn_right()</code>	Defines the right-turn for the bot()
<code>stick_to_line()</code>	Helper method to correct bot position
<code>left_forward()</code>	Helper method to set rotation of left wheel in forward direction
<code>left_backward()</code>	Helper method to set rotation of left wheel in backward direction
<code>right_forward()</code>	Helper method to set rotation of right wheel in forward direction
<code>right_backward()</code>	Helper method to set rotation of right wheel in backward direction

Table 2.2: Table 2.1 continued

This is all about variables and methods that you need to know. You can always refer to this table when in doubt about what the methods does or what the variable indicates.

## 2.2 Directions

### 2.2.1 Defining directions

The direction which the bot faces when kept at the start point is referenced as direction 0 or north. The remaining directions are defined accordingly in the clockwise sense. Figure 2.1 gives two examples. These directions are absolute and remain same throughout the time. We reference the paths pertaining to other nodes according to this convention which gets defined at the start point.

### 2.2.2 Updating directions

We defined `bot_dir` in section 2.1, recall that it denotes the current direction. We must update `bot_dir` after every turn we take. Table 2.3 describes the same. Suppose the bot is pointing in direction 3, and



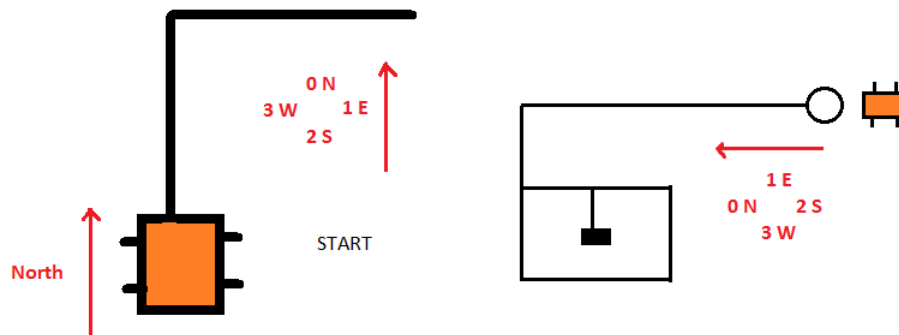


Figure 2.1: Directions are defined with respect to position of bot when placed at start point.

Turn	New direction
Right turn	$((bot\_dir+1)\%4)$
Left turn	$((bot\_dir+3)\%4)$
U turn	$((bot\_dir+2)\%4)$

Table 2.3: This table describes the new bot direction after each turn in terms of current bot direction.

takes a right turn, the new direction will be  $(3+1)\%4$  which is 0. It is easy to infer that the formulas above give the correct direction.

## 2.3 The Coordinate System

Each node is uniquely defined by its  $x$  and  $y$  coordinates. We have many ways of keeping track of  $x$  and  $y$  coordinates as we traverse the maze. One of it could be measuring the number of rotations of the wheel so that we keep track of the exact distance traveled by the bot. This is a rather easy way but comes at the cost of additional sensor in the bot design. So we adopt another way to calibrate the coordinate system. The idea will surprise you at first sight.

We use the method named `millis()`, a method provided by standard library of Arduino. The description of `millis()` goes as:

`millis()` : Returns the number of milliseconds since the Arduino board began running the current program.

bot_dir	New coordinates
0	<code>cur_y += time_diff</code>
1	<code>cur_x += time_diff</code>
2	<code>cur_y -= time_diff</code>
3	<code>cur_x -= time_diff</code>

Table 2.4: Updating  $x$  and  $y$  coordinates of the bot after every call to `millis()` based on `bot_dir`

We know that distance  $\propto$  time, meaning the distance traveled by the bot from one node to another is proportional to the time taken by the bot to do so assuming the bot travels with constant speed which is a good practical assumption. We call `millis()` every time we take a turn or hit a node and update current x and y coordinates. This can easily be accomplished by keeping track of the result of previous call to `millis()` as variable `prev_time` and current call as `curr_time`. The difference between the two times gives you the distance traveled by the bot since the previous turn was taken or a node was detected where we also updated the `cur_x` and `cur_y`. The bot coordinates that must be updated at each call can be easily decided by looking at the current bot direction. Table 2.4 illustrates the same.

- R** The bot begins facing direction 0 as we defined in section 2.2. Initially `cur_y` and `cur_x` are zero. Suppose that the bot hits a node after going straight from start point. Yes, the y coordinate of bot must be increased. Since the bot is still facing direction 0, as per table 2.4, we increase `cur_y`. Next suppose we take a left turn at this node, so bot is now facing direction 3. We go straight and hit another node. We update the coordinates once again. This time the x coordinate must be decreased since the bot took a left turn at previous node. Glance at the table and infer that we do the same!

## 2.4 Nodes

### 2.4.1 Node definition

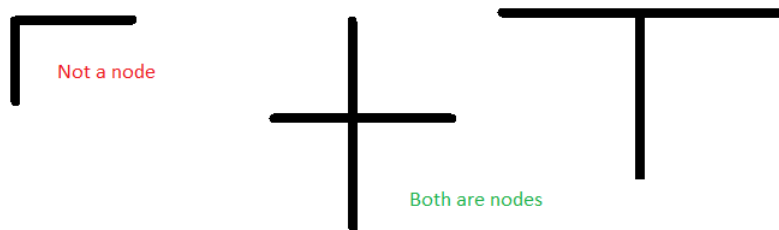


Figure 2.2: Some examples of Nodes and non Nodes.

We define any junction having strictly more than two paths as a node. This is because only those junctions with more than two paths allow us to make a decision about the path we need to choose next. Ex. For the node on the left, we have to take a right turn when we reach the corner, but for the one on the right we can choose to turn left or right.

### 2.4.2 Node characteristics

- We reach a node through some path, we call it the back path of that node which is defined as a new variable `int back_path`. We declared it as an integer so that it holds the absolute value of direction: 0,1,2 or 3.
- For each node, we could have all four paths or some of the paths may not exist. We choose to represent the available paths for a node in a `bool` array named `paths` of size 4. With reference to section 2.2 describing directions, the array indices (starting from 0) represent the directions: 0 for North, 1 for East and so on.

```
bool paths[4] = { false };
```

So if there exist paths in all four directions, we set them all to `true`. If there exists a path in direction 1 or East, we set `paths[1] = true`.

- As we explore the node, we will be choosing some path to travel while the other paths will still be unvisited. So we need another `bool` array of size 4 to allow us to remember which paths corresponding to that node have been visited and which are unvisited.

```
bool vpath[4] = { false };
```

It is obvious that only the paths that exist for a particular node will be marked as visited or unvisited. Initially there are no available paths for a node and all of them are unvisited (they do not exist actually) so everything is set to `false`.

- A node is uniquely defined by its `x` and `y` coordinates. So we have:

```
int x_cor, y_cor;
```

Hence the overall node defined as a `struct` goes as:

```
struct Node{
    bool paths[4]= { false };
    bool vpath[4]= { false };
    int x_cor,y_cor;
    int back_path;
} nodes[50];
```

Notice that we have declared an array of size 50 of type `struct Node` along with the `Node` definition. This will act as a data structure for storing all nodes that we visit assuming maximum number of nodes would not be more than 50. You can increase it depending on the size of maze. That was all that you need to know about nodes.

### 2.4.3 Matching Nodes

Nodes can be matched by matching their `x` and `y` coordinates by practically testing and finding out the epsilon value which works well such that:

```
abs(node1.x_cor - node2.x_cor) < epsilon && abs(node1.y_cor - node2.y_cor) < epsilon
```

We introduce you to ways in which a node gets matched with previously visited node. It is based on the paths that we take. Understanding these two cases is particularly important to understand the actual algorithm because our decisions at the nodes will be based on these two cases. We call them the `RETURN-KNOWN` case and `RETURN-UNKNOWN` case.

#### 1. RETURN-KNOWN

We revisit the node via a path which we took earlier. Hence the name `-KNOWN`. We set `returning = true` in this case. You can refer to section 2.1 to recall what `returning` variable denotes though it will be explained later in detail in section 2.5.

#### 2. RETURN-UNKNOWN

We revisit the node via a new path. Hence the name `-UNKNOWN`. We set `returning = false` in this case. This is because this path was not taken in first place, it so happened that the maze made us take this path looping around somewhere ahead along the path we took earlier. More in detail in the next section.

## 2.5 The Returning Variable

This might be the most difficult part to understand but we will try our best to get you through this. We define a new variable

```
bool returning = false;
```

This will be a global variable in our code. As the name suggests, we set it to true when we are returning, but returning from where? You will come to know in the next few lines. What returning variable simply tells us is that the corresponding path pertaining to this revisited node does not lead us to finish, so we can actually forget that there existed such a path pertaining to this node. In other words we can now continue the search taking other unvisited paths pertaining to the node. We consider two cases when the returning variable is true.

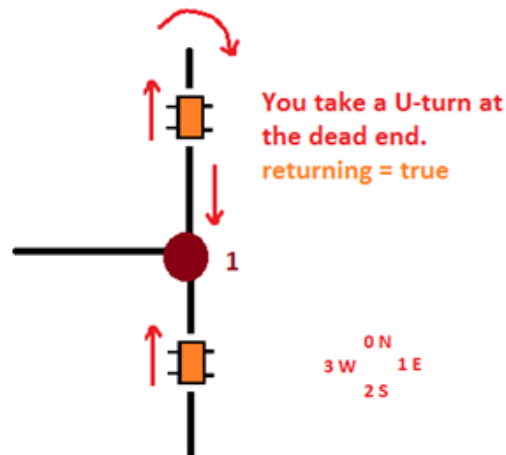


Figure 2.3: Illustration of U-TURN case.

- BACK-PATH

The algorithm is all about taking appropriate decisions at every node. Many a times we will come across a situation where all paths pertaining to the node will be visited. Simply put, even after exploring all paths pertaining to the node we couldn't find finish on any of the paths and we have hit the same node again. In this case, we choose to take back\_path of the node along with setting returning = true. Now what does this mean? When we start traveling along the back\_path of this node, we will again hit some node visited earlier. For this node, the node that we visit just next, the path we took does not contain finish. So this path becomes redundant for that node and we can simply forget that such a path existed for the node and continue search over other unvisited paths. If it's not clear, do not worry, an example at the end of this chapter will make everything clear. With this we state,

**Rule of thumb:** Whenever we take back\_path we set returning = true.

- U-TURN

Consider another simple situation given in figure 2.3. You come across a dead end while exploring the maze. You have no choice but to take a U-turn.

**Rule of thumb:** Whenever we take a U-turn, we set `returning = true`.

If it doesn't make sense to you, think about what `returning` variable tells you. It tells you that the path doesn't lead you to finish point and you can simply forget it. As we keep moving after having taken the U-turn, we come across node 1 (a known or visited node as we call it), but we know `returning = true`, so the path in direction 0 or North pertaining to node 1 doesn't contain the finish point. We can now explore remaining unvisited paths like in the direction 3 or West pertaining to node 1 without worrying about this path.

## 2.6 An Example

Figure 2.4 depicts a portion of a maze. We hit a node 1 while exploring the maze, since this is a new node for us we define its characteristics as below.

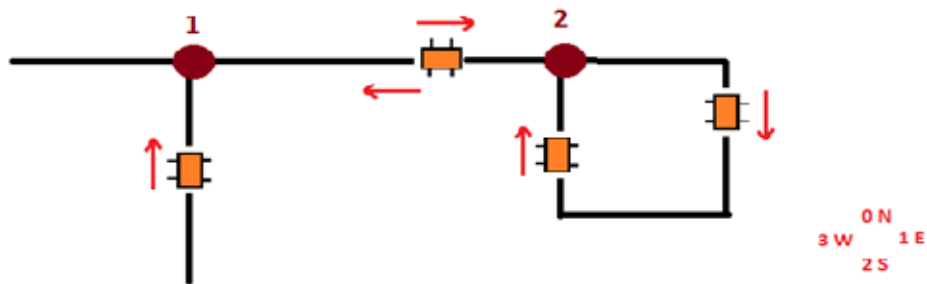


Figure 2.4: An example for understanding prerequisites.

```

\\node 1
\\These paths exist for node 1
node1.paths[2] = node1.paths[3] = node1.paths[1] = true;
\\We came by path 2
node1.back_path = 2;
\\Marking path 2 as visited
node1.vpath[2] = true;


```

Additionally we also set the `x_cor` and `y_cor` for node 1 using `cur_x` and `cur_y` which is not written above. Also we assume that you update the `bot_dir` after every turn so it is not explicitly mentioned here. We then take a right turn at node 1. The algorithm will be explained in detail in the next chapter. For now assume that these are the decisions at the nodes. After we take right turn, we mark the path as visited,

```
node1.paths[1] = true
```

and as we travel further we hit node 2. We define it's characteristics as below and move straight marking the corresponding path as visited.

```
node2.paths[1] = true
```

 Recall that the corners are not nodes since we have no decision to make there.

```
\\node 2
\\These paths exist for node 2
node1.paths[2] = node1.paths[3] = node1.paths[1] = true;
\\We came by path 3
node1.back_path = 3;
\\Marking path 3 as visited
node1.vpath[3] = true;
```

As we continue to move by following the line, we loop around and hit node 2 again. Which case is this? Recall from section 2.4.3, this is the classical instance of RETURN-UNKNOWN case as we have revisited a node via an unknown path. We won't explain what algorithm suggests to do at this point of time but assume for now that we take a U-TURN in this case. Recalling our **thumb rule** for U-TURN, we set,

```
returning = true;
```

Interestingly we again hit node 2, but this time in RETURN-KNOWN case. Hope you are able to understand the two cases and how this example actually reflects them. Now that we are at node 2, observe that all paths pertaining to node 2 have been visited. This is a classical instance where we are forced to take back\_path of node 2. So we continue to move straight, recalling our **thumb rule** for BACK-PATH, we do not change the value of returning variable.

As we continue to move along the back path of node 2, we hit node 1, classical instance of RETURN-KNOWN case. As we said in section 2.4.3, returning = true in this RETURN-KNOWN case and so is it. Do you observe that now? Look how **thumb rules** for returning variable and the RETURN-KNOWN and RETURN-UNKNOWN cases got intertwined.

## 2.7 Conclusion

This was all we had for you in this chapter. Hope you are clear with the two cases for matching nodes and the the two rules of thumb mentioned in this chapter. In the next chapter, we will directly jump onto the actual algorithm.



## 3. The algorithm

### 3.1 Running the algorithm

Without any delay, we start with an example maze and run the algorithm over it. We will explain the decision that we make at each node as we traverse the maze. So let's get started. The example maze is shown in figure 3.1.

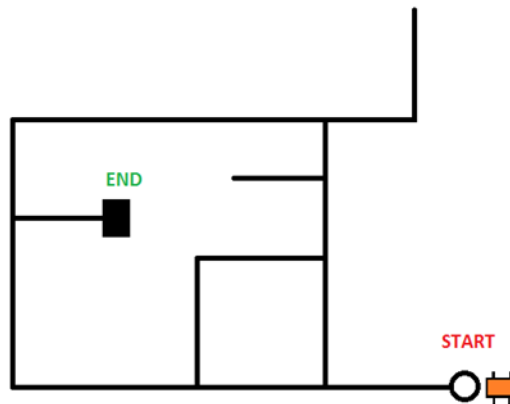


Figure 3.1: An example MAZE for algorithm illustration.

Initializing the variables: We set the current x and current y coordinates to 0. Pointer variable always points to one plus the number of nodes visited and remembered till now and bot direction is set to 0. The returning variable is set to `false` initially. You can recall the definitions of these variables from section 2.1.

```
int ptr = 1;
```

```

int cur_x = 0;
int cur_y = 0;
bot_dir = 0;
returning = false;

```

Directions will be initialized with respect to the bot in the start position. So we mark and henceforth refer to directions as in figure 3.2.

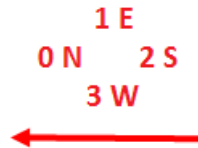


Figure 3.2: Initialization of directions.

We move in the forward direction until we hit a node. We come across our first node and hence define the characteristics of that node. The characteristics are self-explanatory. Notice that we also increment the pointer variable.

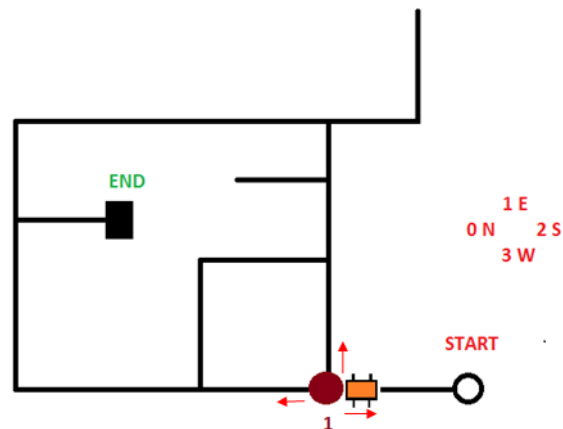


Figure 3.3: Marking node 1.

```

//node 1
//Marking the paths that exist for node 1
node[1].path[0] = node[1].path[2] = node[1].path[1] = true
// We came by path in direction 2
node[1].vpath[2] = true
//Marking the back path
node[1].back_path = 2
node[1].x_cor = cur_x
node[1].y_cor = cur_y
//Incrementing the pointer, ptr = 2 after incrementing

```



```
ptr = ptr +1
//marking the path we take at this node
node[2].vpath[0] = true
```

We won't state the marking of x and y coordinates for further nodes. It is assumed hereon that you would be marking them accordingly. We hope the directions are making sense.

- R The direction of bot and current x and y coordinates of the bot are updated at each turn. We don't mention them here explicitly to keep the things simple. You can go back to chapter 2 to recall how bot direction and coordinates of bot are updated.

Now we need to decide which path to choose to explore further. We decide the preference order as S - straight, R - right and then L - left. This simply means that we will prefer moving straight if such a path exists and is not visited, else we choose to go right or else left as the last option. You can have any order as you want. It is not possible to correctly state which order or preference will give you the most optimal path hence you can choose anything. As you would later see, choosing paths randomly will be beneficial. But for understanding purpose we go with S, R and L. From node 1 we go straight until we hit another node, we call it node 2. We also mark the path that we took as visited for node 1.

We define the characteristics of node 2 and increment the pointer variable.

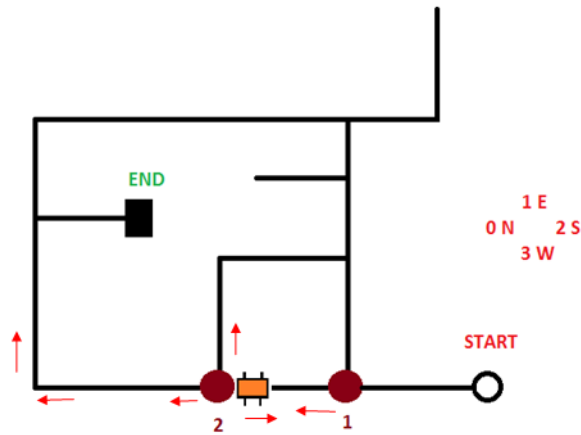


Figure 3.4: Marking node 2.

```
node[2].path[0] = node[2].path[2] = node[2].path[1] = true
node[2].vpath[2] = true
node[2].back_path = 2
ptr = ptr +1 //ptr = 3 after incrementing
//marking the path we take
node[2].vpath[0] = true
```

We again choose to move straight according to our preference order and mark this path as visited for node 2. This time we come hit a corner and are forced to take a right turn. We keep traveling

until we hit another node, we call it node 3.  
The characteristics of node 3 are set as given.

```
node[3].path[3] = node[3].path[2] = node[3].path[1] = true
node[3].vpath[3] = true
node[3].back_path = 3
ptr = ptr + 1 //ptr = 4 after incrementing
//marking the path we take
node[3].vpath[1] = true
```

We choose to move straight according to our preference and mark the path as visited. Last line of the snippet does the same. We hit the corner and take a right turn. We move straight until we come across another node, we call it node 4. We define it's characteristics as below. Next we again continue to move straight, mark the path as visited, we then hit a corner, take left turn and hit a dead end. We have no choice but to take a U-TURN. Recall our **thumb rule**, we set `returning = true` after every U-TURN. Makes sense? The path in direction 2 for node 4 does not contain the finish point, so we simply forget that such a path existed for node 2 and continue our search taking other unvisited paths of node 4.

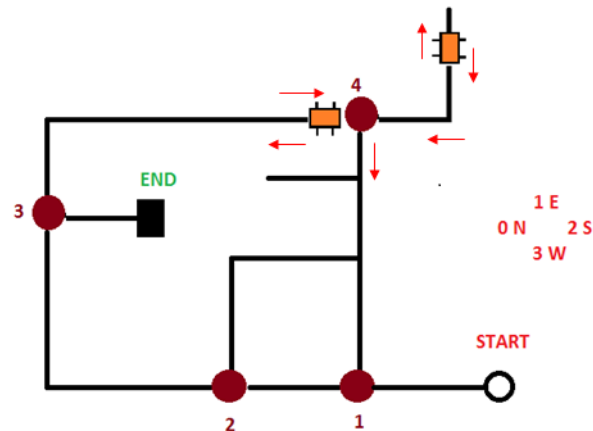


Figure 3.5: Marking node 3 and 4.

```
node[4].path[0] = node[4].path[2] = node[4].path[1] = true
node[4].vpath[0] = true
node[4].back_path = 0
ptr = ptr + 1 //ptr = 5 after incrementing
//marking the path we took
node[4].vpath[2] = true
```

We illustrated two cases when we revisit a node. Here `returning = true`, so this is the classical instance of RETURN-KNOWN case we mentioned earlier. The decision to be made is really simple in this case, we just choose the appropriate unvisited path of node 4, that's what the algorithm says in this case. If you are wondering how we came to know that we have hit node 4 particularly, then, that's what we stored `x_cor` and `y_cor` for each node. Briefly speaking,

```
abs(node[4].x_cor - cur_x) < epsilon &&
abs(node[4].y_cor - cur_y) < epsilon
```

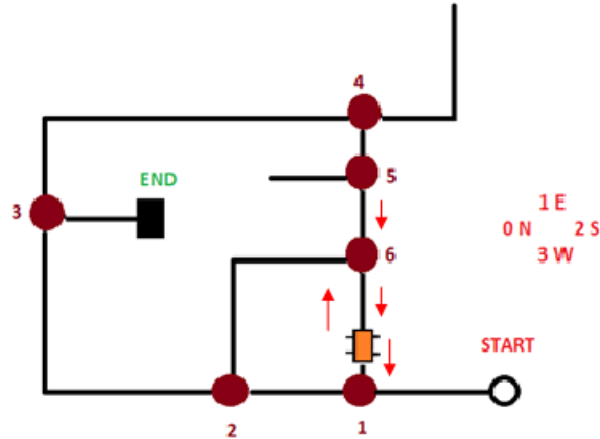


Figure 3.6: Marking node 5 and 6, node 1 revisited in RETURN-UNKNOWN case.

So we see that the path in direction 3 is the only unvisited path. We take a left turn and mark this path as visited and continue to explore further. Note that the pointer variable is still the same, we did not increment it. Hence `ptr` is still 5.

```
node[4].vpath[3] = true
```

But we must not forget to set returning back to false, as we have chosen an unvisited path!

```
returning = false
```

We hit another two nodes that is node 5 and node 6 and in both cases we choose to move straight according to the preference order we decided marking the paths chosen as visited. The node definitions of node 5 and node 6 are shown below.

```
\node 5
node[5].path[0] = node[5].path[1] = node[5].path[3] = true
node[5].vpath[1] = true
node[5].back_path = 1
ptr = ptr +1 //ptr = 6 after incrementing
//marking the path we take
node[5].vpath[3] = true
\node 6
node[6].path[0] = node[6].path[1] = node[6].path[3] = true
node[6].vpath[1] = true
node[6].back_path = 1
ptr = ptr +1 //ptr = 7 after incrementing
//marking the path we take
node[6].vpath[3] = true
```

We keep moving straight until we hit some node. Not surprisingly, we hit a node we had already visited earlier but also in this case `returning = false`. This is the classical instance of RETURN-UNKOWN case we mentioned earlier. The decision to be made is pretty different from all earlier cases. What should we do now? Before taking any decision we should keep in mind that there are still paths which we did not visit pertaining to nodes we visited earlier. Ex. path in direction 0 pertaining to node 5 and path in direction 1 pertaining to node 2. So, according to the algorithm, when we hit such a node, precisely we hit a known node in RETURN-UNKNOWN case, we take a look at all the paths starting from this known node (node 1 in this case) to the node currently pointed to by `ptr-1`.

**R** `ptr` always points to `node+1`.

Accordingly we check all paths corresponding to nodes starting from node 1 (matched node) to node 6 (`ptr-1`). This the only range of nodes we check unvisited paths for, that is from node 1 to node 6. If there exist any unvisited paths, we take a U-TURN according to our algorithm. Do they exist here? Yes, in direction 2 pertaining to node 3 and many others as we mentioned earlier. We take U-TURN, but remember what our **thumb rule** said, we set `returning = true`.

Assume if such unvisited paths did not exist, then what should we do? Yes, as you might have guessed, we need not visit those nodes again as they do not lead us to finish, we then set the `ptr` variable to currently matched node, 1, in this case and try choosing some other unvisited path pertaining to node 1. It's important to note that we took a U-TURN since we had some paths which we did not visit.

Take a close look at the maze diagram shown in figure 3.7. Notice particularly the paths which are

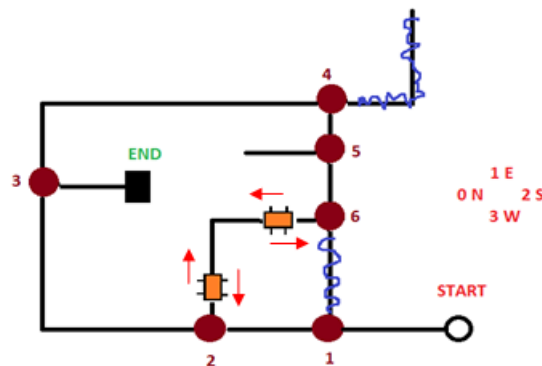


Figure 3.7: node 2 and node 6 revisited in RETURN-UNKOWN and RETURN-KNOWN case respectively.

crossed in blue! Yes, does it make sense? These are the two paths for which returning was true and we simply forgot that such paths ever existed in our maze! Why? Because they never lead us to finish. We have now hit a matched node, which case? RETURN-KNOWN case. What do we do? Take some unvisited path pertaining to node 6, so we take path in direction 0 which is the only unvisited path. Don't forget that we set `returning = false` once we take this unvisited path and also mark `node[6].vpath[0] = true`. Ahh...we again hit a node which we had already visited. Which case?

Yes, RETURN-UNKNOWN case where `returning = false`. We check if there are any unvisited paths starting from node 2 (matched node) to node 6 (`ptr-1`). We have not visited many paths yet, although one less than earlier. We take a U-TURN, set `returning = true` according to our **thumb rule**. So this path corresponding to node 6 gets crossed.

We have again hit a known node, RETURN-KNOWN case, `returning = true`. But this time, it's a bit different. Can you guess it? We are left with no unvisited paths for node 6. We take the `back_path` for node 6. Are we forgetting something? **thumb rule**: Whenever we take `back_path` we set `returning = true`. There you go, we keep moving in direction 1 from node 6 setting `returning = true`. One important thing we would like to state here is that node 6 will never be

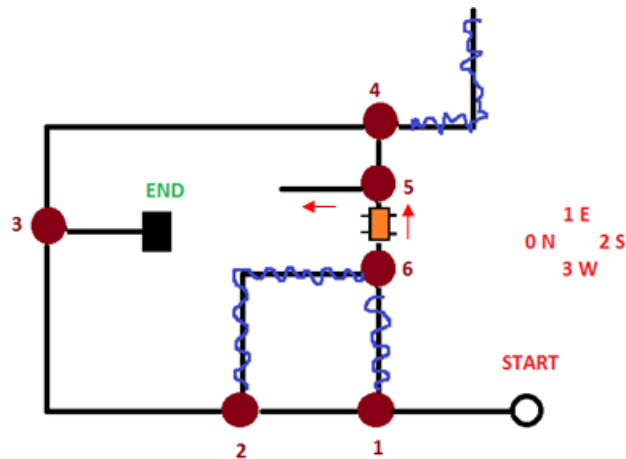


Figure 3.8: node 5 revisited

revisited again for two reasons: 1) We have already explored all paths pertaining to node 6 and 2) We took the back path for the same reason from node 6. How do we put this thing into our code? Decrement the pointer variable! So node 6 is no more a node in the maze for us, why? Look all its paths are crossed in the figure 3.8.

```
ptr = ptr - 1    // ptr = 6 after decrementing, forgot node 6
```

Next we revisit Node 5, RETURN-KNOWN case, choose unvisited path in direction 0. Set `returning = false` as we choose an unvisited path. Having taken the left turn at node 5, we hit a dead end and are forced to take a U-TURN, setting `returning = true`. We revisit node 5 in RETURN-KNOWN case, but again all paths pertaining to node 5 are already visited this time, so we take back path for node 5. `returning` variable is unchanged since we are taking back path and `returning = true` already. We can now safely cross over these two paths too, since they do not lead us to finish. We reach node 4, RETURN-KNOWN case, `returning = true`, all paths have already been visited, we take back path for node 4. We can safely cross over this path from node 5 to node 4. The figure 3.9 depicts the same. Again for node 5 like node 6, we have crossed all paths and we are not gonna visit node 5 again. So we decrement the pointer variable.

```
ptr = ptr - 1 \\ ptr = 5 after decrementing, forgot node 5
```

We hit node 3, RETURN-KNOWN case, `returning = true`, we cross over the path from node 4 to node 3, decrement `ptr` and take an unvisited path from node 3.

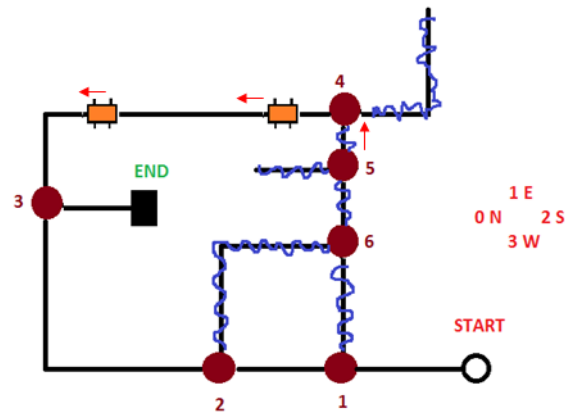


Figure 3.9: node 4 revisited

`ptr = ptr -1 \\ ptr = 4 after decrementing, forgot node 4`

Bingo! There you are.....Finished! What's important here is that the value of `ptr` is exactly equal to the number of nodes in the optimal path that lead us to finish!

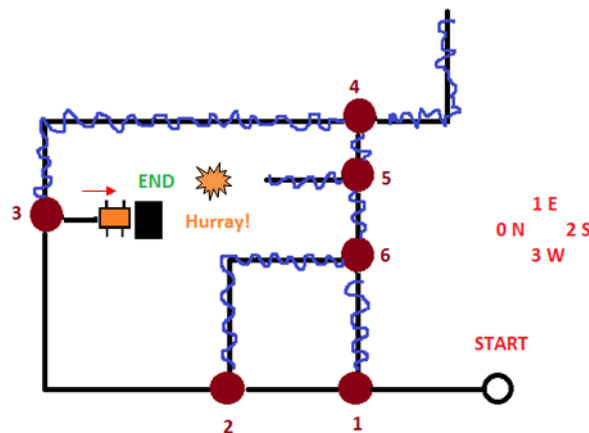


Figure 3.10: node 3 revisited

The blue crossed portion corresponds to the those paths that do not lead us to finish. Thus we have successfully explored the maze and now we know the optimal path to finish. This completes the run of our algorithm on this maze. To recall, whenever we hit a node in RETURN-KNOWN case, two things can happen, there are some unvisited paths for the node or there aren't. If there are some unvisited paths, we choose them according to our preference order and if there aren't then we take the back path for the node. Secondly, if we hit a node in RETURN-UNKOWN case, we check for all paths starting from this node that got matched and the node pointed to by `ptr-1`. If there exist any unvisited paths for any of the nodes in this range, we take a U-TURN, else we set our `ptr` variable to this node+1, and try choosing some unvisited path for this node. If no unvisited path exists, we take the back path. This summarizes our algorithm. In section 3.2, we show you how to take the optimal path as a part of round 2 of our competition. We also present the pseudo code in the section 3.3.

### **3.2 The Optimal Path**

Coming up shortly....!

### **3.3 Pseudo Code**

Coming up shortly....!



## 4. Making the Bot

### 4.1 Gathering the materials

Coming up shortly....!

### 4.2 The bot design

Explain about the two bot designs that we thought of, 1) placing the sensor array in front of the bot and 2) placing the sensor array at the center of the bot. Explain the advantages and disadvantages too.

#### 4.2.1 IR in-front design

#### 4.2.2 IR-centered design

Coming up shortly....!

### 4.3 Configuring the IR sensor array

Coming up shortly....!

### 4.4 Teaching bot to take turns

Explanation about the ways to take turn like rotating both wheels or a single wheel, etc. Best way to take a U-TURN and it's calibration.

#### 4.4.1 LEFT-TURN and RIGHT-TURN

#### 4.4.2 U-TURN

Coming up shortly....!

*Wish you all the best, Andrea Hidalgo*