# Maze Solving Algorithms for Micro Mouse

Surojit Guha
EC VI Sem JEC
surojitguha1989@gmail.com

Sonender Kumar
EC VI Sem GGCT
sonenderkumar@gmail.com

## Abstract

The problem of micro-mouse is 30 years old but its importance in the field of robotics is unparalleled, as it requires a complete analysis & proper planning to be solved. This paper covers one of the most important areas of robot, “Decision making Algorithm” or in lay-man’s language, “Robot Intelligence”. For starting in the field of micro-mouse it is very difficult to begin with highly sophisticated algorithms. This paper begins with very basic wall follower logic to solve the maze. And gradually improves the algorithm to accurately solve the maze in shortest time with some more intelligence. The Algorithm is developed up to some sophisticated level as Flood-Fill algorithm. The paper would help all the beginners in this fascinating field, as they proceed towards development of the “brain of the system”, particularly for robots concerned with path planning and navigation

## 1. Introduction

The Micromouse competition is an annual contest hosted by the Institute of Electrical and Electronics Engineers (IEEE). A small autonomous mobile robot called a micromouse, must navigate through an unknown maze and locate the center. The robot that makes the fastest time run from the start to the center of the maze is declared the winner of the competition.

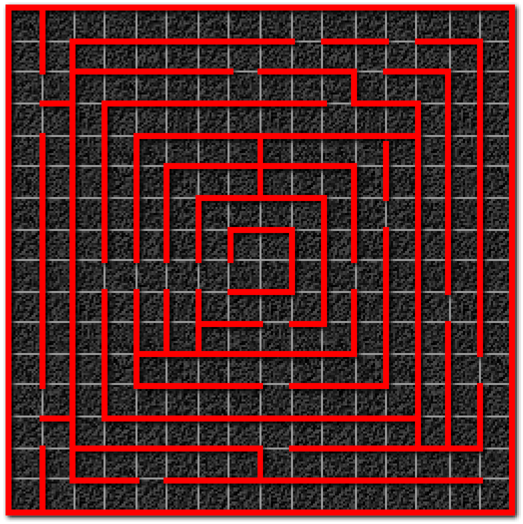
The maze is made up of a 16 by 16 grid of cells, each 180 mm square with walls 50 mm high. The mice are completely autonomous robots that must find their way from a predetermined starting position to the central area of the maze unaided. The mouse will need to keep track of where it is, discover walls as it explores, map out the maze and detect when it has reached the goal. Having reached the goal, the mouse will typically perform additional searches of the maze until it has found an optimal route from the start to the center. Once the optimal route has been found, the mouse will run that route in the shortest possible time.

Championship-level mice can make it from the start cell to the finish cell in well under 20 seconds with top speeds averaging 2 meters/sec, now that's a fast mouse!

## 2. The Wall Follower

The wall following algorithm is the simplest of the maze solving techniques. Basically, the mouse follows either the left or the right wall as a guide around the maze.

Although there are wall following competitions for the younger students, this algorithm does not work in the IEEE maze solving competitions because those mazes are specifically designed to not be solved in this way. Take a look at the maze.



***The right wall following routine:***

*Upon arriving in a cell:*

```
If there is an opening to the right
        Rotate right
Else if there is an opening ahead
        Do nothing
Else if there is an opening to the left
        Rotate left
Else
        Turn around
End If

Move forward one cell
```

You can see that if the mouse follows the left or right walls, it would only explore the perimeter of the maze without ever venturing into the middle section.

## 3. Depth First Search

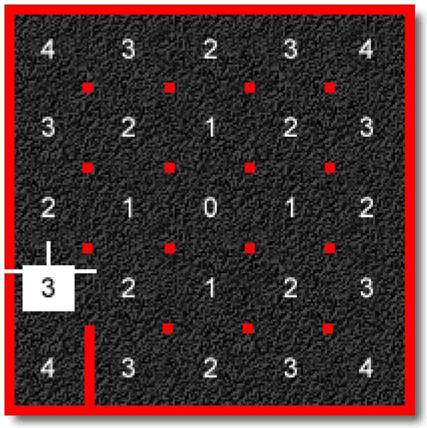
The depth first search is an intuitive method of searching a maze. Basically, the mouse simply starts moving. When it comes to an intersection in the maze, it randomly chooses one of the paths. If that path leads to a dead end, the mouse backtracks to the intersection and chooses another path. This forces the robot to explore every possible path within the maze. By exploring every cell within the maze the mouse will eventually find the center.

Obviously, exploring the entire maze is not an efficient way of solving it. Also, this method finds a route but it doesn't necessarily find the quickest or shortest route to the center. This algorithm wastes too much time exploring the entire maze.

Also this algorithm does not work for the mazes which do not contain any deep corner.

## 4. The Flood-Fill Algorithm

The flood-fill algorithm involves assigning values to each of the cells in the maze where these values represent the distance from any cell on the maze to the destination cell. The destination cell, therefore, is assigned a value of 0. If the mouse is standing in a cell with a value of 1, it is 1 cell away from the goal. If the mouse is standing in a cell with a value of 3, it is 3 cells away from the goal. Assuming the robot cannot move diagonally, the values for a 5X5 maze without walls would look like this.



Of course for a full sized maze, you would have 16 rows by 16 columns = 256 cell values. Therefore you would need 256 bytes to store the distance values for a complete maze.

When it comes time to make a move, the robot must examine all adjacent cells which are not separated by walls and choose the one with the lowest distance value. In our example above, the mouse would ignore any cell to the West because there is a wall, and it would look at the distance values of the cells to the North, East and South since those are not separated by walls. The cell to the North has a value of 2, the cell to the East has a value of 2 and the cell to the South has a value of 4. The routine sorts the values to determine which cell has the lowest distance value. It turns out that both the North and East cells have a distance value of 2. That means that the mouse can go North or East and traverse the same number of cells on its way to the destination cell. Since turning would take time, the mouse will choose to go forward to the North cell. So the decision process would be something like this

***Decide which neighboring cell has the lowest distance value:***

***Is the cell to the North separated by a wall?***
*Yes -> Ignore the North cell*
*No -> Push the North cell onto the stack to be examined*

***Is the cell to the East separated by a wall?***
*Yes -> Ignore the East cell*
*No -> Push the East cell onto the stack to be examined*

***Is the cell to the South separated by a wall?***
*Yes -> Ignore the South cell*
*No -> Push the South cell onto the stack to be examined*

***Is the cell to the West separated by a wall?***
*Yes -> Ignore the West cell*
*No -> Push the West cell onto the stack to be examined*

*Pull all of the cells from the stack (The stack is now empty)*
*Sort the cells to determine which has the lowest distance value*

***Move to the neighboring cell with the lowest distance value.***

Now the mouse has a way of getting to center in a maze with no walls. But real mazes have walls and these walls will affect the distance values in the maze so we need to keep track of them. Again, there are 256 cells in a real maze so another 256 bytes will be more than sufficient to keep track of the walls. There are 8 bits in the byte for a cell. The first 4 bits can represent the walls leaving you with another 4 bits for your own use. A typical cell byte can look like this:

Remember that every interior wall is shared by two cells so when you update the wall value for one cell you can update the wall value for its neighbor as well. The instructions for updating the wall map can look something like this

***Update the wall map:***

***Is the cell to the North separated by a wall?***
*Yes -> Turn on the "North" bit for the cell we are standing on and*
*Turn on the "South" bit for the cell to the North*
*No -> Do nothing*

***Is the cell to the East separated by a wall?***
*Yes -> Turn on the "East" bit for the cell we are standing on and*
*Turn on the "West" bit for the cell to the East*
*No -> Do nothing*

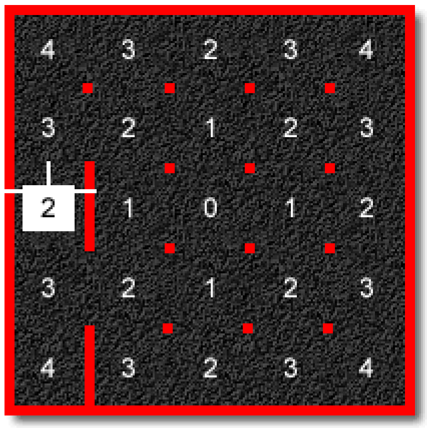
***Is the cell to the South separated by a wall?***
*Yes -> Turn on the "South" bit for the cell we are standing on and*
*Turn on the "North" bit for the cell to the South*
*No -> Do nothing*

***Is the cell to the West separated by a wall?***
*Yes -> Turn on the "West" bit for the cell we are standing on and*
*Turn on the "East" bit for the cell to the West*
*No -> Do nothing*

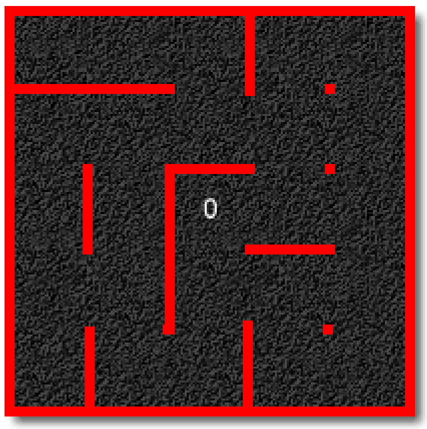
So now we have a way of keeping track of the walls the mouse finds as it moves about the maze. But as new walls are found, the distance values of the cells are affected so we need a way of updating those. Returning to our example, suppose the mouse has found a wall.

We cannot go West and we cannot go East, we can only travel North or South.

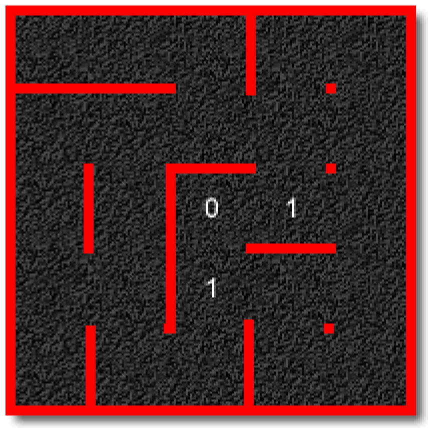
But going North or South means going up in distance values which we do not want to do. So we need to update the cell values as a result of finding this new wall. To do this we "flood" the maze with new values



As an example of flooding the maze, let's say that our mouse has wandered around and found a few more walls.

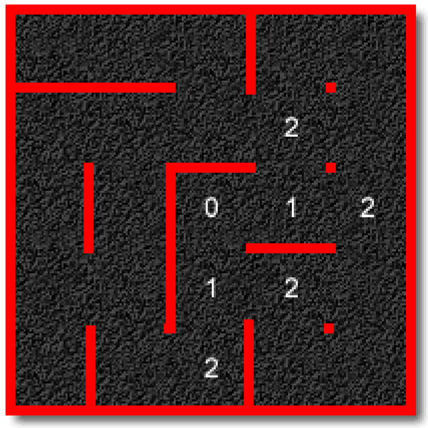


The routine then takes any open neighbors (that is, neighbors which are not separated by a wall) and assigns the next highest value, 1.

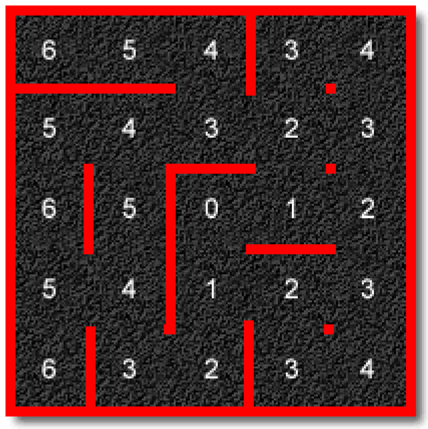


The routine would start by initializing the array holding the distance values and assigning a value of 0 to the destination cell:

The routine again finds the open neighbors and assigns the next highest value, 2:



A few more iterations:



Notice how the values lead the mouse from the start cell to the destination cell through the shortest path.

The instructions for flooding the maze with distance values could be:

```
***Flood the maze with new distance values:***

*Let variable Level = 0*
*Initialize the array DistanceValue so that all values = 255*
*Place the destination cell in an array called CurrentLevel*
*Initialize a second array called NextLevel*

***Begin:***

*Repeat the following instructions until CurrentLevel is empty:*

{
Remove a cell from CurrentLevel
If DistanceValue(cell) = 255 then

    let DistanceValue(cell) = Level and
    place all open neighbors of cell into NextLevel

End If
}

The array CurrentLevel is now empty.

Is the array NextLevel empty?
No ->

{
Level = Level +1,
Let CurrentLevel = NextLevel,
Initialize NextLevel,
Go back to "Begin:"
}

Yes -> You're done flooding the maze
```

The flood-fill algorithm is a good way of finding the shortest (if not the fastest) path from the start cell to the destination cells. You will need 512 bytes of RAM to implement the routine: one 256 byte array for the distance values and one 256 array to store the map of walls.

Every time the mouse arrives in a cell it will perform the following steps:

***(1) Update the wall map***

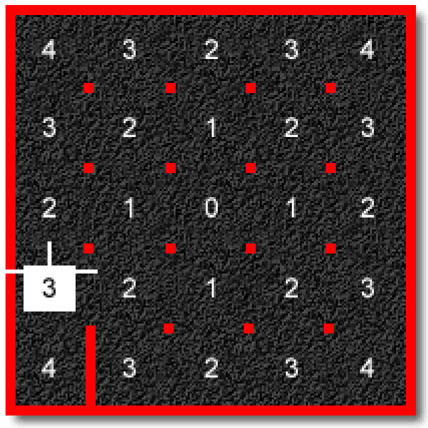
***(2) Flood the maze with new distance values***

***(3) Decide which neighboring cell has the lowest distance value***

***(4) Move to the neighboring cell with the lowest distance value***

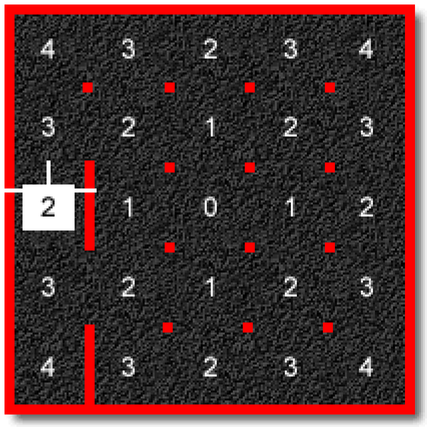
## 5. The Modified Flood-Fill Algorithm

The modified flood-fill algorithm is similar to the regular flood-fill algorithm in that the mouse uses distance values to move about the maze. The distance values, which represent how far the mouse is from the destination cell, are followed in descending order until the mouse reaches its goal.



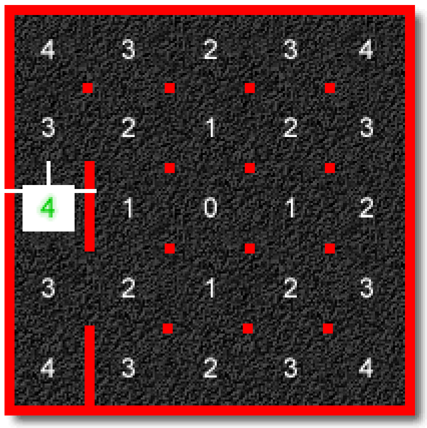
As the MicroMouse finds new walls during its exploration, the distance values need to be updated. Instead of flooding the entire maze with values, as is the case with the regular flood-fill, the modified flood-fill only changes those values which need to be changed. Let's say our mouse moves forward one cell and discovers a wall.

The robot cannot go West and it cannot go East, it can only travel North or South. But going North or South means going up in distance values which we do not want to do. So the cell values need to be updated. When we encounter this, we follow this rule:



***If a cell is not the destination cell, its value should be one plus the minimum value of its open neighbors.***

In the example above, the minimum value of its open neighbors is 3. Adding 1 to this value results in 3 + 1 = 4. The maze now looks like this:



There are times when updating a cell's value will cause its neighbors to violate the "1 + minimum value" rule and so they must be checked as well. We can see in our example above that the cells to the North and to the South have neighbors whose minimum value is 2. Adding a 1 to this value results in 2 + 1 = 3 therefore the cells to the North and to the South do not violate the rule and the updating routine is done.

Now that the cell values have been updated, the mouse can once again follow the distance values in descending order.

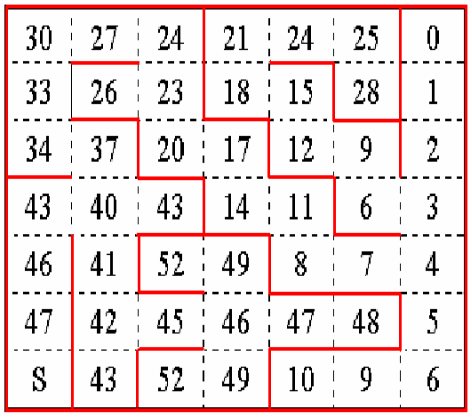
So our modified flood-fill procedure for updating the distance values is:

```
Update the distance values (if necessary)

Make sure the stack is empty

Push the current cell (the one the robot is standing on) onto the stack

Repeat the following set of instructions until the stack is empty:

{
Pull a cell from the stack
Is the distance value of this cell = 1 + the minimum value of its open neighbors?
No -> Change the cell to 1 + the minimum value of its open neighbors and
push all of the cell's open neighbors onto the stack to be checked
Yes -> Do nothing
}
```

Most of the time, the modified flood-fill is faster than the regular flood-fill. By updating only those values which need to be updated, the mouse can make its next move much quicker.

## 6. The Time Flood Algorithm

The Bellman flooding algorithm is a popular maze solver with micro mouse contestants and has been used by several world championship-winning mice.

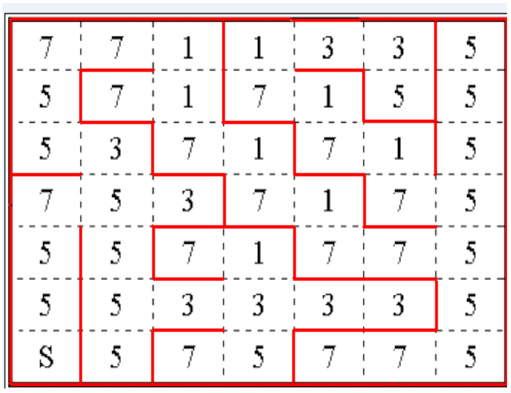


| | | | | | | |
|---|---|---|---|---|---|---|
| 30 | 27 | 24 | 21 | 24 | 25 | 0 |
| 33 | 26 | 23 | 18 | 15 | 28 | 1 |
| 34 | 37 | 20 | 17 | 12 | 9 | 2 |
| 43 | 40 | 43 | 14 | 11 | 6 | 3 |
| 46 | 41 | 52 | 49 | 8 | 7 | 4 |
| 47 | 42 | 45 | 46 | 47 | 48 | 5 |
| S | 43 | 52 | 49 | 10 | 9 | 6 |

**Bellman time flooding array**

The standard Bellman algorithm solves the maze for the shortest route, but this is not always the quickest. To find the quickest route to the centre of the maze it is necessary to use an advanced form of the Bellman flooding algorithm, which floods with time instead of distance. There might be two paths in first path robot has to make more turns than the other path in which the number of steps are few more.

For this Time Flooding Array first of all we have to create a Direction Array in which the direction of robot is stored.



| | | | | | | |
|---|---|---|---|---|---|---|
| 7 | 7 | 1 | 1 | 3 | 3 | 5 |
| 5 | 7 | 1 | 7 | 1 | 5 | 5 |
| 5 | 3 | 7 | 1 | 7 | 1 | 5 |
| 7 | 5 | 3 | 7 | 1 | 7 | 5 |
| 5 | 5 | 7 | 1 | 7 | 7 | 5 |
| 5 | 5 | 3 | 3 | 3 | 3 | 5 |
| S | 5 | 7 | 5 | 7 | 7 | 5 |

**Direction Array**

In the above direction array the current direction of the robot is stored. Now the flooding array is filled in taking direction array as reference.

## 7. Conclusion and Future Work

This paper reports the results of a feasibility study which aims to establish that in future some military purpose vehicles or we say the unmanned vehicles which can searches its own path in the battlefield avoiding the obstacles. Currently this project is undertaken by the government of U.S to design autonomous vehicles for military purpose by the year 2020.

It can be implemented in making the vehicles autonomous, DARPA Challenge in US challenges engineers to make autonomous vehicles which can cross through the hurdles placed in their way. The basic motive behind this is to make unmanned vehicles for US Army till 2020.

The algorithm applied to search the shortest path i.e. Bellman Flood-Fill may be used by the telecom industry in future to search for the shortest route even when some intermediate MSC's are fully congested.

Finally, we aim to combine Artificial Intelligence and some other sophisticated algorithms on our robots to make them more intelligent and smart slaves which reduces our work efficiently.