VOLUME I

1. PROJECT OBJECTIVE

The objective of the project is to solve the classic problem of maze solving by using distributed robotics. Two micromice need to be designed to negotiate a path to the center of a maze in the optimal amount of time. For this purpose an algorithm needs to be developed which allows the two of them to work in parallel and reduce redundant mapping by communicating and sharing information with each other and aiding each other to reach the centre. The design must integrate a microcontroller, sensors, motors and communication modules.

This paper presents the current design and interface of the different modules as well as preliminary data related to the performance of the micromouse. Currently, one MicroMouse has been designed and made fully functional, ie capable of solving the maze on its own. Future challenges lie in implementing the communication modules and improvising the algorithm to allow efficient parallel coordinated operation of both mice for maze navigation and solving with increased efficiency.

The project objectives can be summarized as follows:

- To solve the maze using multiple mice.
- Helpful in solving bigger Mazes.
- Does so by communicating with each other.
- Thus individual intelligence of the Bots combine to form a single intelligence.
- Each Robot can take decisions based on the success or failure of the other.

2. HISTORY OF MICROMOUSE

The micromouse competition has been running since the late 1970s around the world. The modern form of the competition originates in 1980 or so.

Essentially, you have a wooden maze made up of a 16 by 16 grid of cells. Mice 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. As if that was not enough, the winner is the mouse that manages this the fastest. There are many versions of the full rules on-line and there are a number of minor variations on how the score of the mouse is determined.

Although modern micromice are relatively sophisticated beasts, this is an extremely challenging undertaking. One of the earliest mice, now about 20 years old is still regularly entered in competitions and puts up a very respectable show.

IEEE Spectrum magazine introduced the concept of the micromouse. In May 1977, Spectrum announced the 'Amazing Micromouse Competition' which would be held in 1979 in New York. There were 15 competitors running out of around 6000 initial entries. This competition involved mice finding their way out of a 10' by 10' maze.

The second UK micromouse contest was held at Wembly. Dave Woodfield's Thumper was the winner with a best time of 47 seconds. Nick Smith's Sterling Mouse was placed second with a best time of 1min 37sec. Alan Dibley took third place with Thezius which did two traversals with a best time of 2min 27sec. Thezius was especially interesting in that the processing power was provided by the relatively new ZX80 personal computer.

The 'First World Micromouse Competiton' was held in Tsukuba, Japan. Mazes were sent to a number of countries around the world in order to encourage entries. A wide range of mice from all over the world competed. The world champion was Noriko-1 from Japan. The top six places were taken by Japanese entries. Seventh was Dave Woodfield from England with Enterprise.

The IEE UK Championship held during July in London was won by members of a Singapore team that took 6 of the 8 places. Dave Woodfield's Enterprise came in 5th while Dave Otten's Mitee Mouse III was placed second. All three of the top mice were within a half second of each other.

The seventh annual IEE micromouse competiton was held in London. Nine mice ran. The winner was Mitee Mouse III with the best overall score although the runner up, Mouse Mobile II by Louis Geoffrey from Canada, made the fastest run. Third prize went to Enterprise. Derek Hall's Motor Mouse 2 managed a good best run but picked up some penalty points as did Andrew Gattell's Mars 1.

Royal Holloway was the venue for what was, arguably, the first competition of the new millenium. Royal Holloway has taken on the responsibility for the UK micromouse competition following the decision by IEE to drop the 3D contest. All the usual suspects were present along with a few hopefuls and new entrants.

In April 2001, micromouse builders from around the world gathered at Royal Holloway College for the first conference of its kind.

3. COMPETITION RULES AND SPECIFICATIONS

Objective

1. In this contest the contestant or team of contestants design and build small self-contained robots (micromice) to negotiate a maze in the shortest possible time.

Rules for the MicroMouse

1. A MicroMouse shall be self-contained (no remote controls). A MicroMouse shall not use an energy source employing a combustion process.
2. A MicroMouse shall not leave any part of its body behind while negotiating the maze.
3. A MicroMouse shall not jump over, fly over, climb, scratch, cut, burn, mark, damage, or destroy the walls of the maze.
4. A MicroMouse shall not be larger either in length or in width, than 25 centimeters. The dimensions of a MicroMouse that changes its geometry during a run shall not be greater than 25 cm x 25 cm. There are no restrictions on the height of a MicroMouse.

Rules for the Maze

1. The maze is composed of multiples of an 18 cm x 18 cm unit square. The maze comprises 16 x 16 unit squares. The walls of the maze are 5 cm high and 1.2 cm thick (**assume 5% tolerance for mazes**). The outside wall encloses the entire maze.
2. The sides of the maze walls are white, the tops of the walls are red, and the floor is black. The maze is made of wood, finished with non-gloss paint.
3. The start of the maze is located at one of the four corners. The start square is bounded on three sides by walls. The start line is located between the first and second squares. That is, as the mouse exits the corner square, the time starts. The destination goal is the four cells at the center of the maze. At the center of this

zone is a post, 20 cm high and each side 2.5 cm. The destination square has only one entrance.

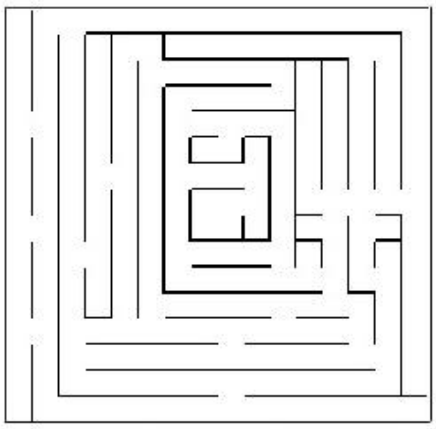
4. Small square zones (posts), each 1.2 cm x 1.2 cm, at the four corners of each unit square are called lattice points. The maze is so constituted that there is at least one wall at each lattice point.
5. Multiple paths to the destination square are allowed and are to be expected. The destination square will be positioned so that a wall-hugging mouse will NOT be able to find it.

Rules for the Contest

1. Each contesting MicroMouse is allocated a total of 10 minutes of access to the maze from the moment the contest administrator acknowledges the contestant(s) and grants access to the maze. Any time used to adjust a mouse between runs is included in the 10 minutes.
2. Each run shall be made from the starting square. The operator may abort a run at any time. If an operator touches the MicroMouse during a run, it is deemed aborted, and the mouse must be removed from the maze. If a mouse has already crossed the finish line, it may be removed at any time without affecting the run time of that run. If a mouse is placed back in the maze for another run, a one-time penalty of **30 seconds** will be added to the mouse's best time.
3. After the maze is disclosed, the operator shall not feed information on the maze into the MicroMouse however, switch positions may be changed.
4. The illumination, temperature, and humidity of the room shall be those of an ambient environment
5. The run timer will start when front edge of the mouse crosses the start line and stops when the front edge of the mouse crosses the finish line
6. Every time the mouse leaves the start square, a new run begins. If the mouse has not entered the destination square, the previous run is aborted.
7. The mouse may, after reaching the destination square, continue to navigate the maze, for as long as their total maze time allows.

8. If a mouse continues to navigate the maze after reaching the destination square, the time taken will not count toward any run. Of course, the 10-minute timer continues to run. When the mouse next leaves the start square, a new run will start. Thus, a mouse may and should make several runs without being touched by the operator. It should make its own way back to the beginning to do so.
9. A contestant may not feed information on the maze to the MicroMouse. Therefore, changing ROMs or downloading programs is NOT allowed once the maze is revealed. **However, contestants are allowed to**:
10. Change switch settings (e.g. to select algorithms)
11. Replace batteries between runs
12. Adjust sensors
13. Change speed settings
14. Make repairs
15. However, a contestant may not alter a mouse in a manner that alters its weight
16. If requested, a break will be provided for a mouse after any run if another mouse is waiting to compete. The 10-minute timer will stop. When the mouse is re-entered, the 10-minute timer will continue. The judges shall arbitrate on the granting of such breaks.

Sample Maze



4. LITERATURE REVIEW

The best source of information regarding micromouse is the internet. The competion details and how to go about making a micromouse is explained in detail in the following websites:

- **http://www.micromouseinfo.com/** - This site provides insight into the various problems faced while attempting to make a micromouse in addition to complete competition specifications and proposed algorithms.
- **http://www.micromouseonline.com/** - this site has an active online forum discussing the most modern approaches for micromouse.

In addition to the micromouse specific websites mentioned above, the wikipedia (**http://en.wikipedia.org/**) and **www.robocet.com** has provided us information about the various shortest path algorithms and sensors.

The Bellman Ford algorithm was studied in details from **On a Routing Problem**, in *Quarterly of Applied Mathematics* published by Richard Bellman in 1958. Other references for the algorithms are **An algorithm for Finding Shortest Routes from all Source Nodes to a Given Destination in General Network**, in *Quarterly of Applied Mathematics*, 1970 by Jin Y. Yen and **A Simple and Fast Label Correcting Algorithm for Shortest Paths** in *Networks magazine*, 1993

There have been many IEEE publications in the recent years regarding the micromouse and its modern approaches. Some of the papers worth mentioning are:

- **An updated micromouse competition** by Ning Chen of *California State University* in *IEEE Frontiers in Education Conference*, 1996.
- **Maze Solving Algorithms for Micrjo Mouse** by Swati Mishra and Panka Bande in *Signal Image Technology and Internet Based Systems*, 2008.
- **Research relevance of mobile robot competitions** by Braunl, T in *Robotics & Automation Magazine, IEEE*, Dec 1999.

5. ALGORITHMS

The software implementation of the micromouse is important for the proper and efficient working of the micromouse, as much as its hardware. As the rules of the competition suggests it is not just solving the maze that counts. The micromice should solve the maze in the shortest time possible giving it the edge over the other competitors participating in the competition. So a proper selection and implementation and even improvisation of algorithm is important for obtaining expected results.

Most of the algorithms used nowadays in maze solving competitions are based on logical synthesis design rules. The algorithms which were used in the beginning years of this competition were simple logical algorithms which became inefficient as the maze itself became complex. The Algorithms used for maze solving competition are

- Wall follower Algorithm
- Depth first search Algorithm
- Flood fill Algorithm
- Modified Flood fill Algorithm

5.1 Wall follower Algorithm

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. And if the mouse encounters an opening in any of the walls picked up by its sensors, the mouse will stop and take a turn in that direction and then move forward sensing the walls gain. Thus keeping the walls as a guide the micromouse hopes to solve maze rather than actually solving it. The steps involved in following the right wall is given below

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

Although this Algorithm was efficient in solving the maze of the beginning years it is not used nowadays. This is because the maze used in competitions nowadays are constructed in such a way that the wall follower algorithm will never solve it. Such mazes have bottleneck regions which will cause the algorithm to fail.

5.2 Depth First Search Algorithm

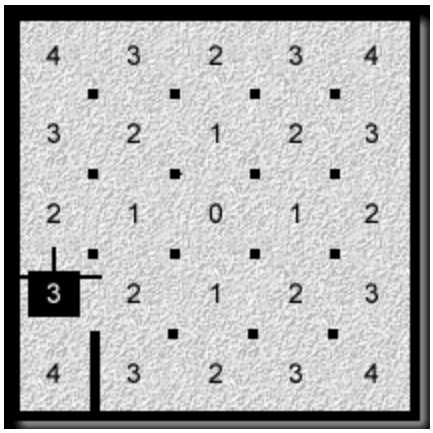
The depth first search is an intuitive method of searching a maze. Basically, the mouse simply starts moving. When it comes to an intersection in the maze, it randomly chooses one of the paths. If that path leads to a dead end, the mouse backtracks to the intersection and chooses another path. This forces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

Even though this algorithm succeeds in solving the maze completely it need not necessarily be the shortest path. In addition to that the mice explores the entire maze for solving the maze leading to long latency. Cases have occurred in competitions where the competitors using this algorithm had to change their batteries in the middle as the mice took too much time in solving the maze. Hence algorithm is also not used in modern competitions.

5.3 Flood Fill Algorithm

The introduction of flood fill algorithm in maze solving methods paved new ways by which modern complex mazes can be solved without any bottlenecks. This algorithm is derived from the Bellman Ford Algorithm coming under the field of logic synthesis techniques. Other algorithms in this field are Djikstra's Algorithm, Johnson's Algorithm etc. These algorithms find applications in various fields like Communication, design softwares etc.

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:

Bit No.	7	6	5	4	3	2	1	0
Wall					W	S	E	N

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**:*

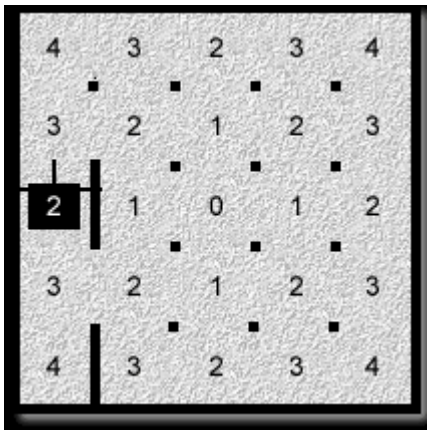
Is the cell to the North separated by a wall?
Yes -> Turn on the "North" bit for the cell we are standing on and
Turn on the "South" bit for the cell to the North
No -> Do nothing

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

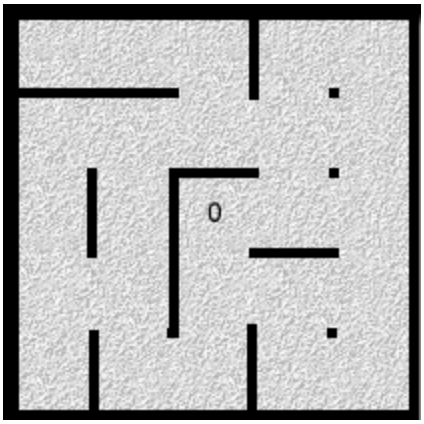
Is the cell to the West separated by a wall?
Yes -> Turn on the "West" bit for the cell we are standing on and
Turn on the "East" bit for the cell to the West
No -> Do nothing

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.

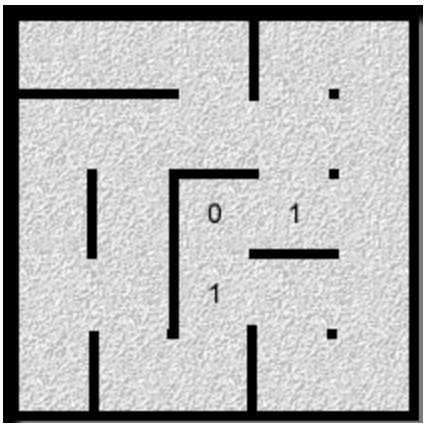


We cannot go West and we cannot go East, we can only travel North or South. But going North or South means going up in distance values which we do not want to do. So we need to update the cell values as a result of finding this new wall. To do this we "flood" the maze with new values.

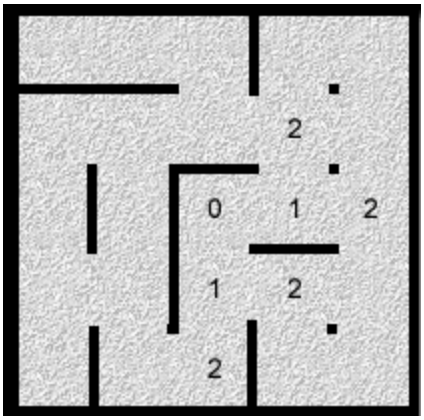
As an example of flooding the maze, let's say that our mouse has wandered around and found a few more walls. The routine would start by initializing the array holding the distance values and assigning a value of 0 to the destination cell. This means that the objective of the mice is to get to the cell which has the cost value with value 0. During the process of solving the maze the cost values of all the cell may change except for the centre cell. It will always be assigned the value 0.



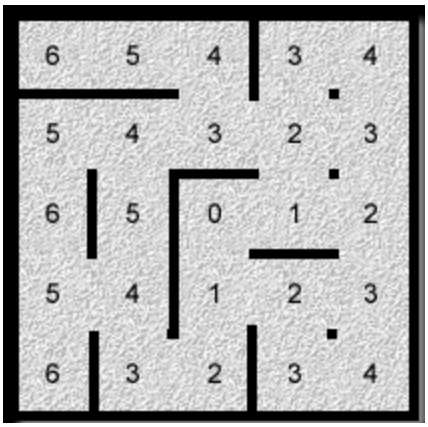
The routine then takes any open neighbors (that is, neighbors which are not seperated by a wall) and assigns the next highest value, 1:



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



This is repeated as many times as necessary until all of the cells have a value:



It can be seen 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

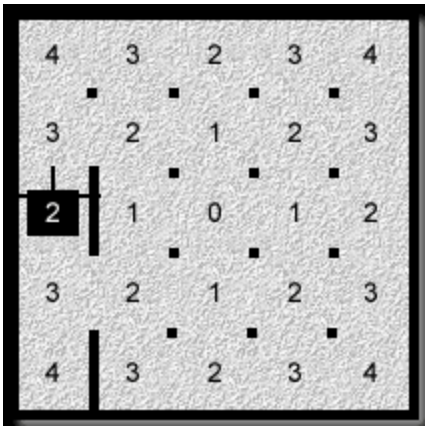
The flood-fill algorithm is a good way of finding the shortest (if not the fastest) path from the start cell to the destination cells. You will need 512 bytes of RAM to implement the routine: one 256 byte array for the distance values and one 256 array to store the map of walls. Every time the mouse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

- **Update the wall map**
- **Flood the maze with new distance values**
- **Decide which neighboring cell has the lowest distance value**
- **Move to the neighboring cell with the lowest distance value**

5.4 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. The basic structure of the algorithm remains the same with some modifications to optimise the working of the flood fill algorithm. This modification helps the mice to solve the maze in lesser time than if it were using a simple flood fill algorithm.

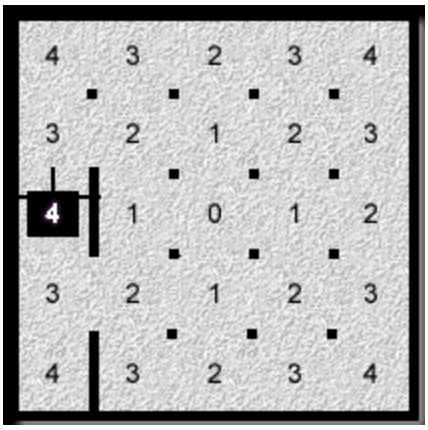
As the Micromouse finds new walls during its exploration, the distance values need to be updated. Instead of flooding the entire maze with values, as is the case with the regular flood-fill, the modified flood-fill only changes those values which need to be changed. Let's say our mouse moves forward 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. MODIFIED DJIKSTRA'S ALGORITHM

As an initial stage, we developed an algorithm that aims to bridge the Djikstra's algorithm with Flood-Fill algorithm. The algorithm description is given below.

It is assumed that the cell from which the mouse is to start and its orientation is already specified. Two bits are used to represent the 4 possible orientations of the mouse. At each cell the mouse will read the sensor inputs to detect the presence of walls. Two matrices of memory locations are maintained to map the maze, both are 16*16. The first cell is given a value zero. All the cells whose value are not yet determined are flushed with the value 254. Each successive cell traversed is given the subsequent value.

After reading the sensor inputs, it will check how many open cells are available into which it can move other than the one from which it came. If it is just one it will move to that cell, else it will compare the central tendencies of the possible cells and will move to the one which lies closer to the centre of the maze. For all the sub-cases that arise, appropriate actions are performed by using a switch function so that each orientation is treated uniquely and the orientation variable is also kept track of. Some of the sub-cases can be resolved into another sub-case. For example, take the case of a read in which all three options are open, ie left, right and straight. If the centre of the maze lies towards the current cell's left, then the cell to the right need not be considered at all as that would only take the mouse further away from the centre of the maze. So this case reduces to the one where the central tendency of just the left and straight moves needs to be resolved.

Another additional check is also performed to avoid repetitive tracking of the same area. Once the sensor readings are taken, it is compared with the maze map we have generated till then to see if any of those open options are cells we already covered. If yes, they will be masked so that it appears internally as if there is a block there so that path will not be taken.

It is on encountering a dead end that the second matrix comes into use. Whenever we come to a junction and prefer one turn over another, we start storing values into a second matrix from that cell position onwards. We give zero for the branch cell and increment the count for

each successive cell traversed. When we finally encounter a dead end after branching, all we have to do is do a 180 degree turn and keep moving to the next lesser value cell according to the second matrix till we reach the original branch cell ie the zero value. During this retrace to the branch cell, the sensors need not be used as we have already chartered the path. After that, the path not taken can be traversed. The resolving of central tendency to choose branch and the fast return to the branch cell on encountering a dead end both ensure effective and efficient operation.

The Modified Djikstra's algorithm is just an algorithm that has central tendency. It does not take into account the previous knowledge of walls and hence find the shortest. This algorithm is efficient in getting to the centre in less number of steps but does not serve our puspose in mapping the maze and finding the shortest path. Hence the algorithm was replaced by the Distributed Flood Fill algorithm.

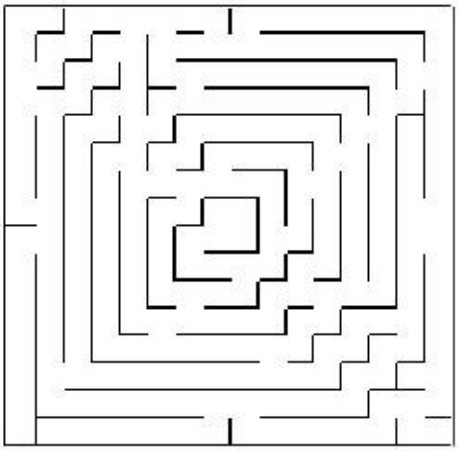
7. DISTRIBUTED FLOOD FILL ALGORITHM

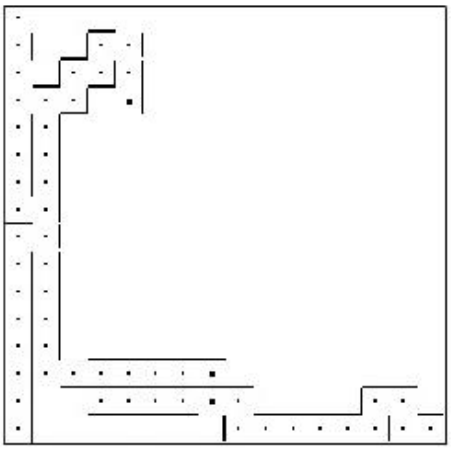
The final algorithm is a modified form of the conventional flood fill algorithm customized to optimize the advantage of using distributed robotics. In this we have multiple bots starting from different points of the maze and trying to solve the maze. Each of the bots updates the wall map in its vicinity, thus the wall map has more than one updates every turn. After the updates are done, flood filling is done. Depending on the flooded values each of the bots in its next turn moves in the direction it believes the shortest path lies. Again the wall map is updated at multiple points after this movement. The process is carried out till all the bots reach the centre during the course of which they explore a good portion of the maze area. It is seen that the explored maze area increases with the number of bots used. Increase in the explored area means better chances of finding an optimum path to the centre.

In conventional flood fill, each successive traversal of the maze by a bot makes use of the information gathered in the previous run. In this case, the tendency is to move to unexplored area as that area is marked as open and devoid of obstacles and hence it is led to believe that a shorter path lies in that direction. This eventually leads to further scouting of the maze rather than finding the shortest path. So, we mark off the unexplored area as inaccessible i.e. surrounded by walls on all sides. This modification is performed on the final wall map obtained after maze traversal by the multiple bots. Flood fill is performed on this modified wall map which yields the shortest path to the centre from whichever starting position is chosen.

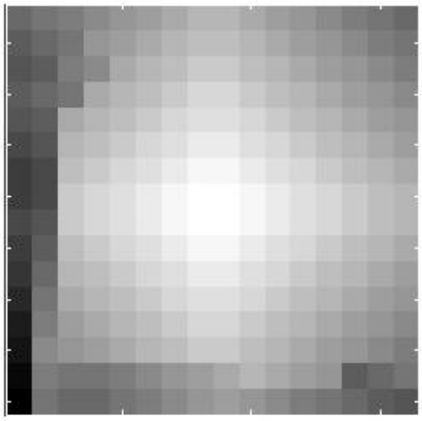
Some mazes have their shortest paths from the original intended starting position itself to the centre without having to visit the neighborhood of the other starting positions of our scouting bots, in which case the advantage of using distributed robotics is severely restricted. In all other cases it is observed that increase in the number of bots from 1 to 2 and 2 to 3 in the initial scouting section leads to a more optimal solution. Interestingly, increase in the number of scouting bots beyond a particular limit fails to yield further improvement. The steps involved in Distributed flood fill algorithm are shown in the upcoming pages with illustrations.

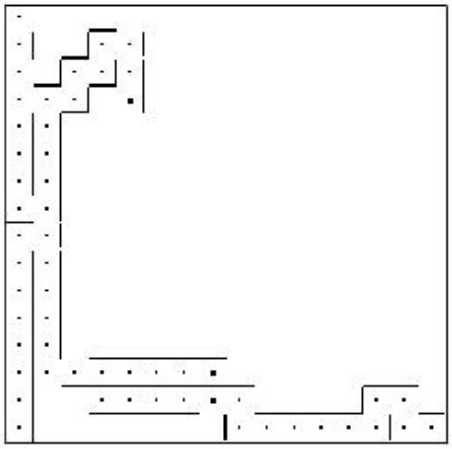
Step 1: Scout using multiple bots
(shown by dotted path)

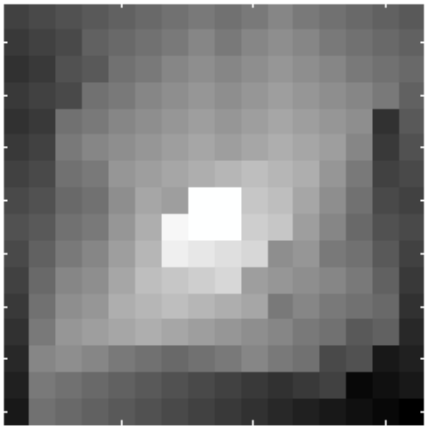


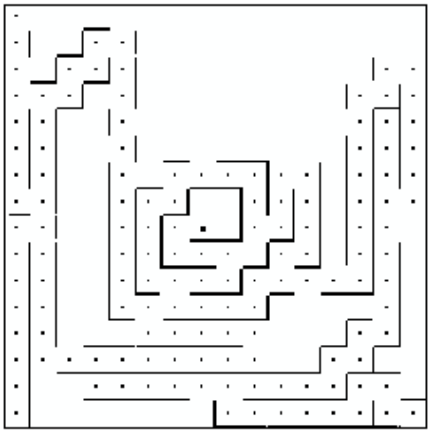


Step 2: Update wall maps and Flood fill till centre is reached





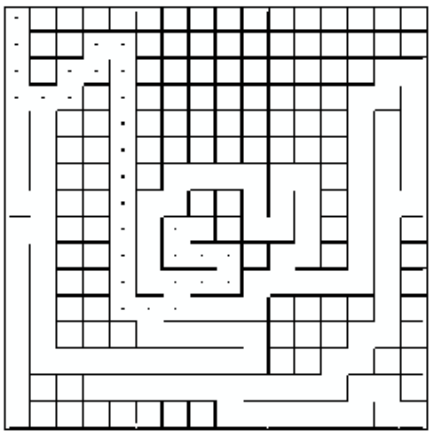




White = min distance from centre
Black = max distance from centre

Lighter dots = path taken
Dark dots = current location of bot

Step 3: Find shortest path along the explored area after blocking off unexplored areas.



8. ROBOT DESCRIPTION:

Every robot has a mechanical part, sensors and intelligence. This project requires a communication channel in addition to those.

8.1 Mechanical Assembly

As with any engineering project, there are countless ways to build a MicroMouse. One has to make design decisions based on the time available, the cost and specifications of the project. The following are the main hardware components of a MicroMouse and things you should consider when making the robot.

8.1.1 Propulsion

There are many ways to make a robot move but in the case of a MicroMouse, there are a few logical choices to choose from. When considering motors, one can choose from continuous DC motors, stepper motors, and servos such as those found in remote-controlled cars and airplanes.

Continuous DC motors have been successfully used in many MicroMouse robots. Motors of every shape and size can be found in the surplus market, in toys and in old electronic gadgets. Dc motors tend to spin too quickly and do not have enough torque to drive the robot wheels directly so some kind of gear reduction will have to be used. Another necessity when using DC motors is an encoder disk on the robot's drive shaft. This shaft encoder provides a pulse train that can be counted and allows the microcontroller to determine how far the robot has traveled and how fast the wheels are spinning.

Hobby servos, such as those found in remote-controlled vehicles, are essentially DC motors with a gear box and control circuitry already built-in. In order to use a servo as a drive motor it must be modified for continuous rotation. The result, however, is a small and strong motor that is good for driving small robots. Another advantage is that these motors come with several attachments that will aid in mounting a wheel. Servos have three connections: +5 volts, ground, and signal. The signal is a pulse between 1 and 2 milliseconds long, at a rate of about 50

pulses per second. By controlling the width of the pulses to the servo, the microcontroller will determine whether the wheel spins forward or backward and how fast it spins

Though DC motors and servo motors have their own advantages, for this project, we have chosen Stepper motors. Further details on Stepper motors are given in the following section.

Stepper Motor

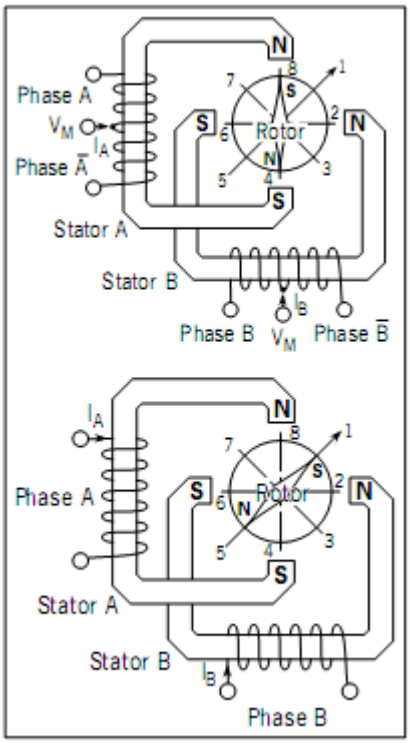
A stepper motor is an electromechanical device which converts electrical pulses into discrete mechanical movements. The shaft or spindle of a stepper motor rotates in discrete step increments when electrical command pulses are applied to it in the proper sequence. The motors rotation has several direct relationships to these applied input pulses. The sequence of the applied pulses is directly related to the direction of motor shafts rotation. The speed of the motor shafts rotation is directly related to the frequency of the input pulses and the length of rotation is directly related to the number of input pulses applied.

Stepper Motor Advantages

1. The rotation angle of the motor is proportional to the input pulse.
2. The motor has full torque at stand-still (if the windings are energized)
3. Precise positioning and repeatability of movement since good stepper motors have an accuracy of 3 – 5% of a step and this error is non cumulative from one step to the next.
4. Excellent response to starting stopping/reversing.
5. Very reliable since there are no contact brushes in the motor. Therefore the life of the motor is simply dependant on the life of the bearing.
6. The motors response to digital input pulses provides open-loop control, making the motor simpler and less costly to control.
7. It is possible to achieve very low speed synchronous rotation with a load that is directly coupled to the shaft.
8. A wide range of rotational speeds can be realized as the speed is proportional to the frequency of the input pulses.

When to Use a Stepper Motor

A stepper motor can be a good choice whenever controlled movement is required. They can be used to advantage in applications where you need to control rotation angle, speed, position and synchronism. Because of the inherent advantages listed previously, stepper motors have found their place in many different applications. Some of these include printers, plotters, highend office equipment, hard disk drives, medical equipment, fax machines, automotive and many more.



Unipolar motors

A unipolar stepper motor has logically two windings per phase, one for each direction of current. Since in this arrangement a magnetic pole can be reversed without switching the direction of current, the commutation circuit can be made very simple (eg. a single transistor) for each winding. Typically, given a phase, one end of each winding is made common: giving three leads per phase and six leads for a typical two phase motor. Often, these two phase commons are internally joined, so the motor has only five leads.

A microcontroller or stepper motor controller can be used to activate the drive transistors in the right order, and this ease of operation makes unipolar motors popular with hobbyists; they are probably the cheapest way to get precise angular movements.

Bipolar motor

Bipolar motors have logically a single winding per phase. The current in a winding needs to be reversed in order to reverse a magnetic pole, so the driving circuit must be more complicated, typically with an H-bridge arrangement. There are two leads per phase, none are common.

Static friction effects using an H-bridge have been observed with certain drive topologies. Because windings are better utilised, they are more powerful than a unipolar motor of the same weight.

Motor Specifications

- 6V 500mA, 12V 1Amp in bipolar mode.
- Can be used in Bipolar mode or Unipolar Mode.
- Weight : 200 gm
- Dimensions : 39 x 39 x 36 mm 5 mm
- Standard Shaft 7 mm length
- 6 Wire interface

Stepper Motor with wheel

8.1.2 Chassis and Wheels

In order for a mouse to operate reliably, its parts have to be mounted to some type of chassis. This does not have to be an elaborate design. Some mice are composed of parts stuck to balsa wood using double-stick foam tape. Fancier designs have parts screwed to an aluminum chassis.

Our chassis is made of a substance known as ‘Foam sheet’. It’s an organic polymer sheet having qualities appropriate for our need. The material is very light hence not adding to the load that motors have to pull. The material property depends upon its thickness. Half a centimeter thickness ensures enough inflexibility yet can be easily cut using a knife. The motors are attached to frame which is bolted on to the base frame.

Motors are no good without wheels. Decisions need to be made as to how many wheels are needed and where they should be located. The wheels chosen by us are made of aluminium. It has a diameter of 60mm with rubber grip. It can be fixed on shaft with a screw. In addition to the driving wheels, two free-moving castor wheels are provided for stability.

8.2 Electronics and Sensors

8.2.1 Battery

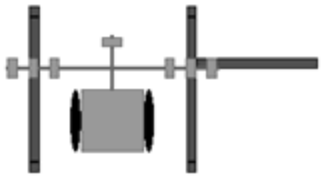
The robot should be self contained in every sense. Therefore it needs a battery to power the sensors, motor and other electronics. Due to its high current driving capability, we have chosen Lead-Acid batteries. The major disadvantage of Lead-Acid battery is that it is heavy. For this reason it may be substituted by Ni-MH or Li-ion batteries.

8.2.2 Sensors

Mice need a way of detecting the walls within the maze and most use near infrared sensors to do this. These IR sensors can either be simple proximity sensors or they can be distance sensors.

Proximity Sensors

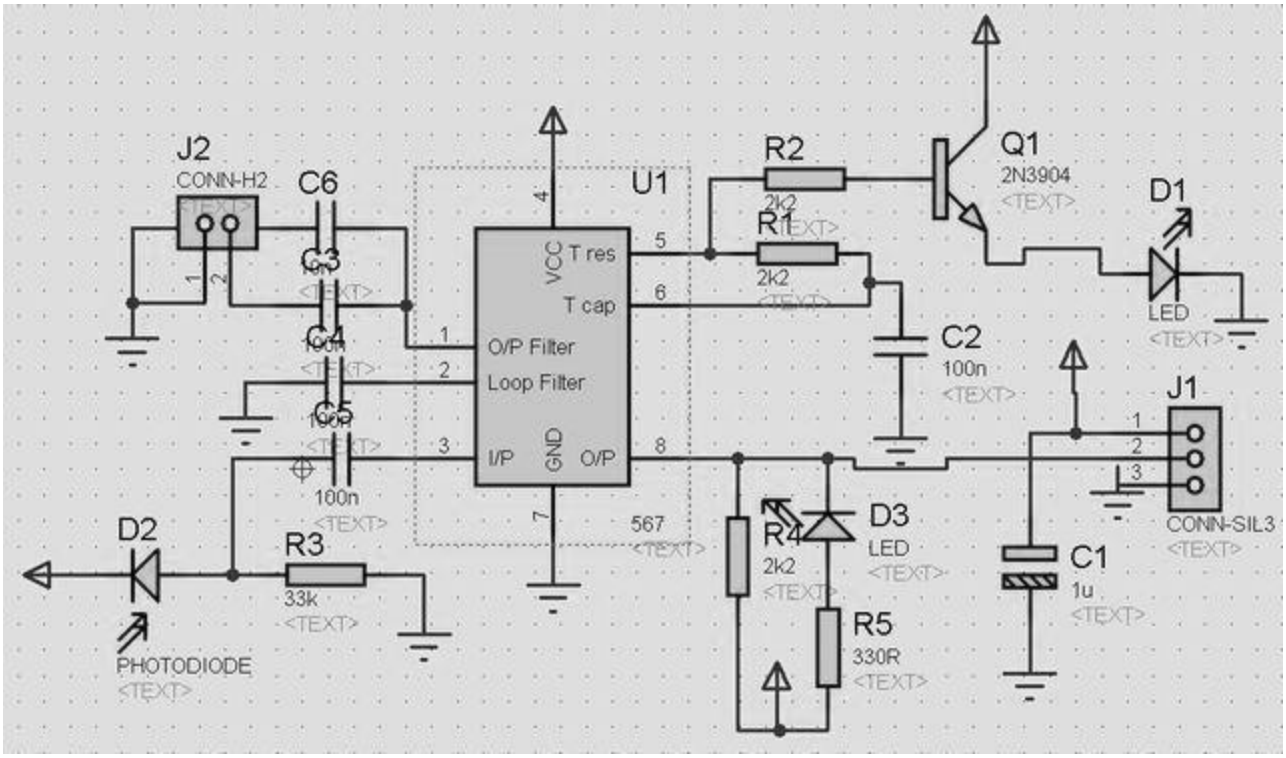
These types of sensors are usually mounted on "wings" so that they can look down onto the tops of the walls. If there is a wall directly below the sensor, it returns a logic value of *true*. If there is no wall below the sensor, the logic value is *false*. This type of mouse would look similar to this:



A Mouse With Proximity Sensors

It has at least one sensor in the front and at least three sensors to each side. The front sensor prevents the mouse from crashing into an oncoming wall. The side sensors map the walls and allow the mouse to correct its heading. Using the drawing above as an example, if the left wing were read we would get a value of 010 indicating that the mouse is centered within the cell. If a value of 100 is read, it would indicate that the mouse is too far to the right and needs to correct to the left. If a value of 001 is read, it would indicate that the mouse is too far to the left and needs to correct to the right.

The following are the schematic and details for an IR proximity sensor that is currently used in the project:



Components:

Resistors

3 R1,R2,R4 2k2; 1 R3 33k; 1 R5 330R

Capacitors

1 C1 1u; 4 C2-C5 100n; 1 C6 10n

Integrated Circuits

1 U1 567

Transistors

1 Q1 2N3904

Diodes

2 D1,D3 LED; 1 D2 PHOTODIODE

The sensor uses a 567 IC which is a tone decoder. The IC has an open collector o/p which goes low when the input frequency matches the freq set by external components. The resistor and capacitor on pins 5 and 6 form the oscillator with time period 1.1RC. The pin 5 will be oscillating. This is connected to a simple transistor and is used to switch the IR led. This way the IR led oscillates at the centre frequency. The 2.2k and 100nf Capacitor are used to get the frequency near 4kHz. The capacitor on pin 2 determines the bandwidth for receiving the signal. The capacitor on pin 1 forms an output filter and its value determines whether the sensor works as a digital sensor or an analog one. It determines the sensitivity of the 567. A value .01nf gave an analog output and cap of 100nf gave digital output experimentally.

Its features are

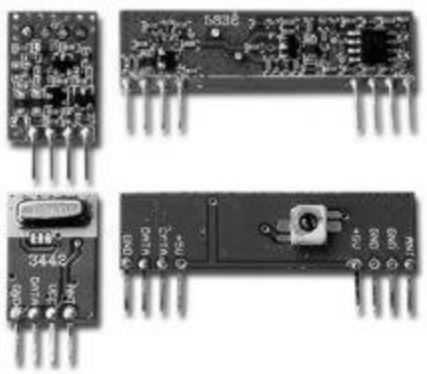
1. Ambient light rejection. range of over 12cm and improvements are possible
2. No microcontroller overhead like for a TSOP
3. Sensor can be made to give a digital o/p or an analog o/p so that it can be used to measure the rough distance from the obstacle.

8.2.3 Motor Driver IC

The motor requires more current than the signaling current from microcontroller. Hence and additional driver IC is required to drive the Stepper motors. ULN2003 Darlington Array IC is used for this purpose.

8.3 Communication Protocol

Communication between robots is essential for implementing distributed robotics. The project aims to use RF Transmitter-Receiver pair for wireless transmission. RF modules are used for this purpose. The serial output from USART module of microcontroller is modulated, send and demodulated using these modules.



RF Transmitter-Receiver pair

Specifications and features:

- Range in open space(Standard Conditions) : 100 Meters
- RX Receiver Frequency : 433 MHz
- RX Typical Sensitivity : 105 Dbm
- RX Supply Current : 3.5 mA
- RX IF Frequency : 1MHz
- Low Power Consumption
- Easy For Application

- RX Operating Voltage : 5V
- TX Frequency Range : 433.92 MHz
- TX Supply Voltage : 3V ~ 6V
- TX Out Put Power : 4 ~ 12 Dbm

8.4 Robot Intelligence

The objective of the project is to find the shortest path for solving the maze in least time. This requires some kind of artificial intelligence. This is provided by the microcontroller. For implementing the flood-fill algorithm, the following requirements should be satisfied:

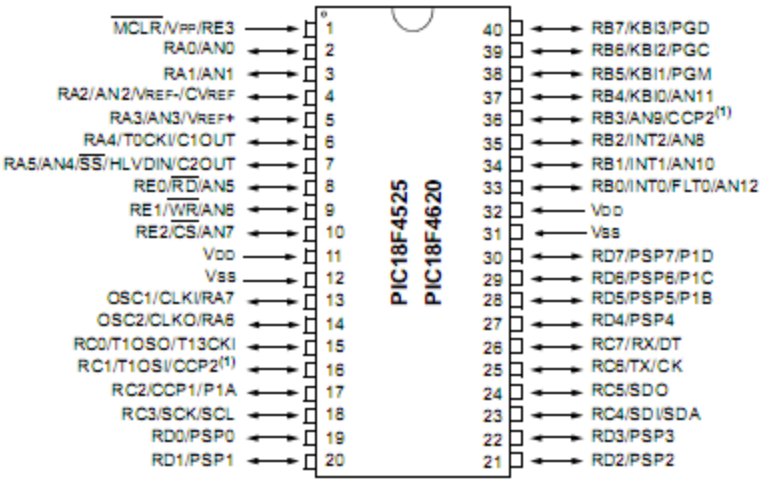
- Requires 512 bytes of RAM for solving a 16 * 16 Maze.
- One 256 byte array for storing Cost values.
- One 256 byte array for storing wall map.

The PIC series of microcontroller is chosen due to their ease of programming using the MPLAB IDE. The choice of PIC is done taking into consideration the following:

- Flood fill algorithm requires 1k of memory.
- 16k to 32k of memory required to be on the safe side.

The PIC 18F4525 is our current choice for this purpose and the compiler used is MPLAB C18 compiler. The important features of PIC 18F4525 are:

- 48 KB Program memory
- 3968 bytes SRAM
- 1 KB EPROM
- Enhanced USART



Pin Diagram of PIC 18F4525

VOLUME II

9. PROJECT COMPLETION AND SIMULATION

The project is currently at the stage before hardware implementation of communication stage. The code for Distributed flood fill has been completed. One robot has been constructed and the same can be duplicated for other bots. There were some practical difficulties encountered while doing the hardware implementation which are as follows.

- The 20x20cm constraint for each square requires the bot to be very compact.
- Mounting of sensors, pcb and battery becomes hard due to the size constraint
- The robot does not travel in a straight line even though stepper motors are used
- A correction mechanism involving 3 more sensors is suggested

Correction Mechanism

The correction sensors are placed at a higher elevation than wall sensors. They sense the wall only when the bot moves too close to the wall. When this happens, the corresponding correction sensor goes high. The correction is applied according to the following table:

Correction Sensor	Correction Applied
Left	Turn Right
Right	Turn Left
Front	Stop current movement

Simulation

The completed code was transported to MATLAB and simulation of it was successfully done using Image Processing Toolbox. MATLAB code closely resembles C coding. Hence transporting the MATLAB code back to C18 code is also easy. The Flood fill is shown by displaying the values as a gradient image with white for locations having minimum distance to centre and black for maximum distance.

The simulation also involves the following figures:

- Original maze
- Animation of path followed by the bot and the wall mapping
- Shortest path along the explored area

10. RESULTS AND ANALYSIS

The following pages show the results with various mazes. The results have been taken from the MATLAB simulation. In all the figures,

Top left – Final Flood fill

Bottom left – Original maze

Top right - Explored maze after scouting

Bottom right - Shortest path along the explored area

The result is shown as a matrix,

Result = [q h v r], where

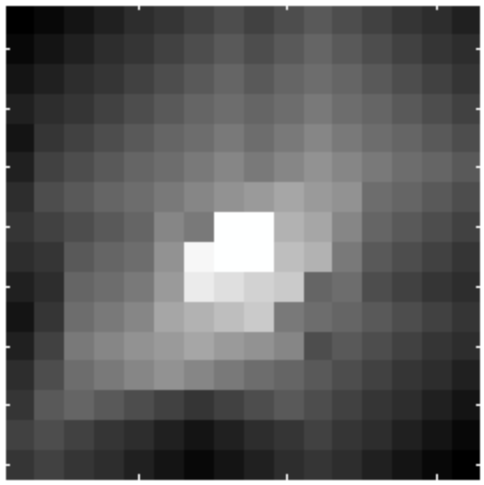
q = no of bots

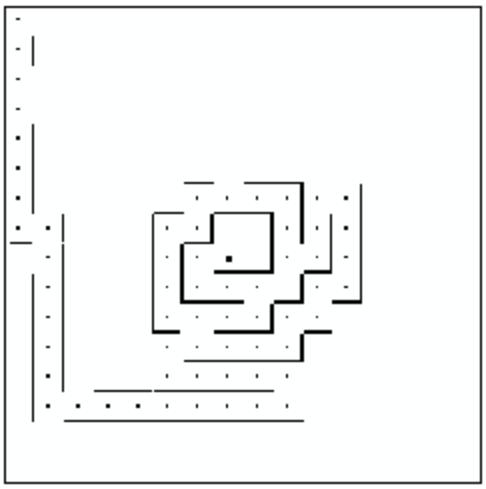
h = steps to reach center while mapping

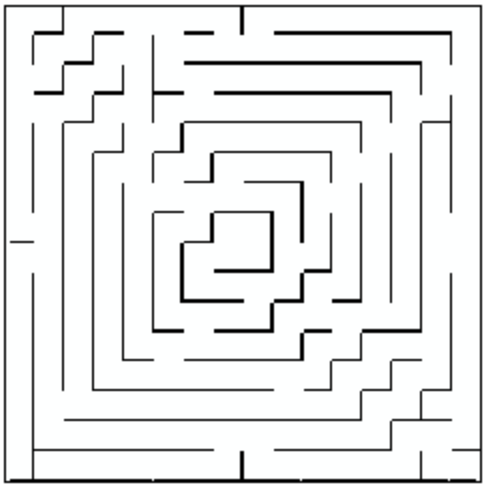
v = mapped shortest path length

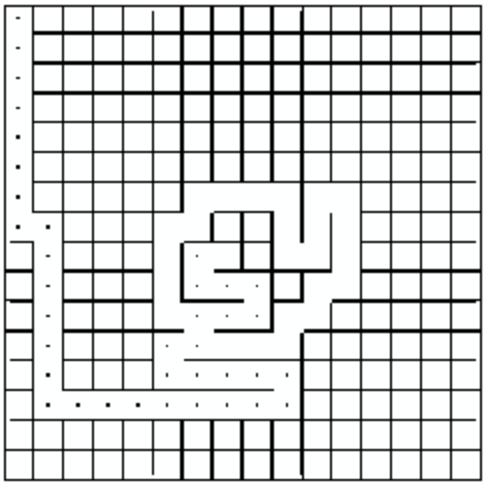
r = mapped squares

1. Maze 1; No. of Bots = 1



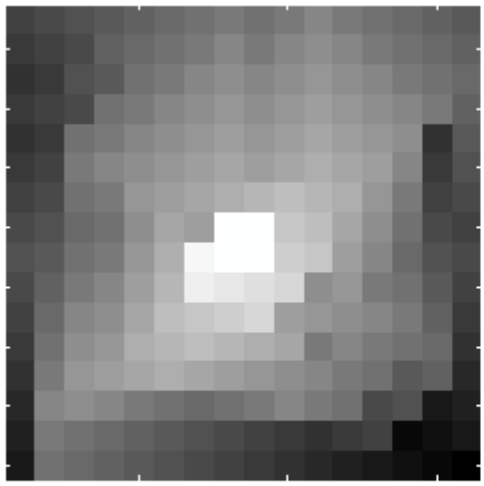


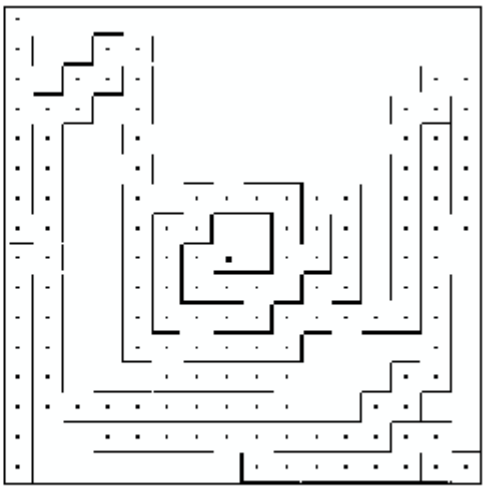


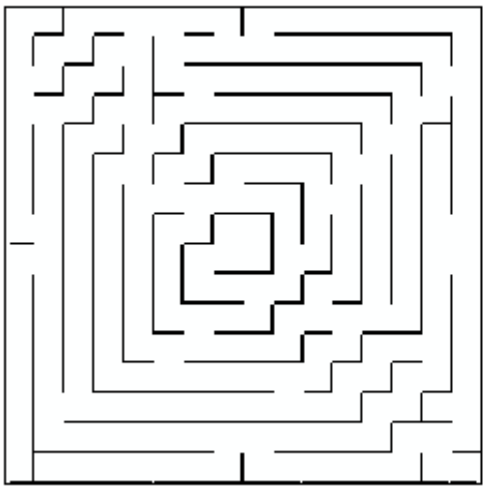


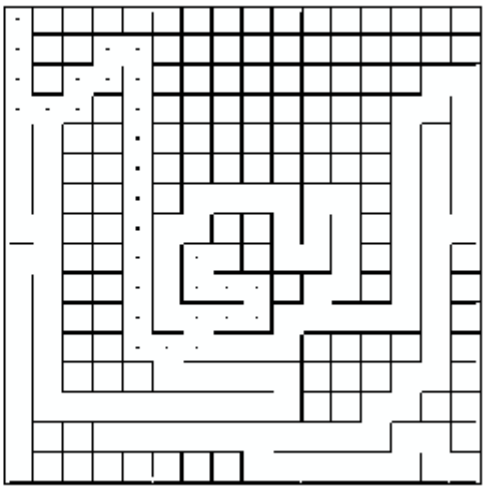
Result = [1 63 37 62]

2. Maze 1; No of bots = 3



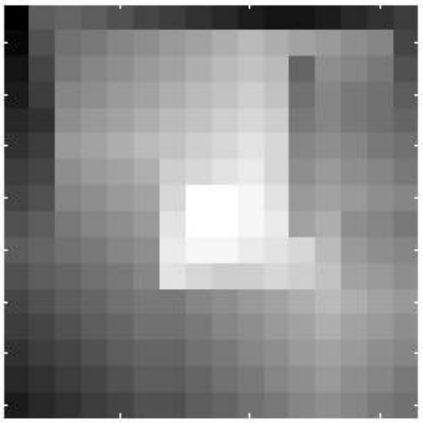


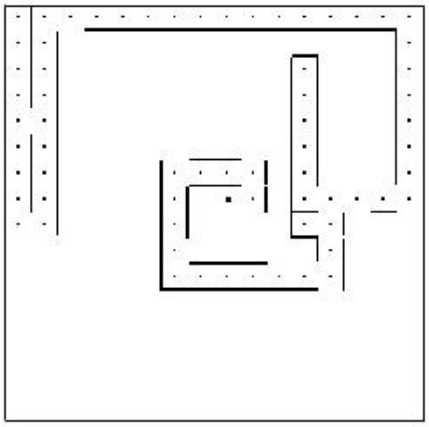


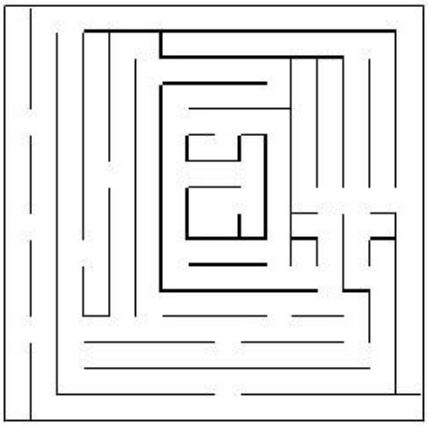


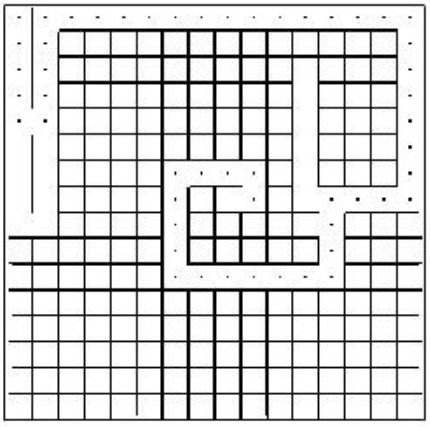
Result = [3 73 29 139]

3. Maze 2; No of bots = 1



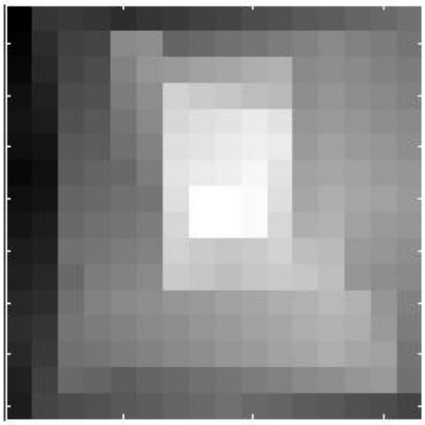


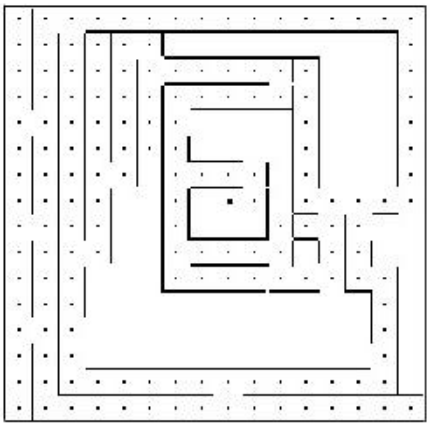


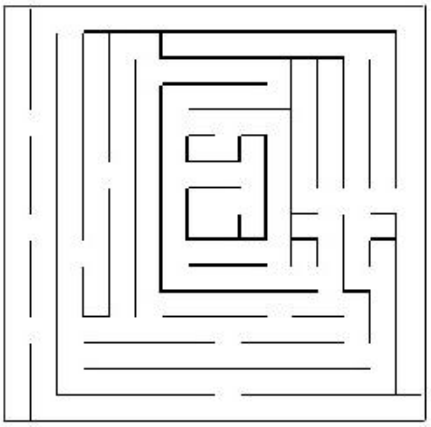


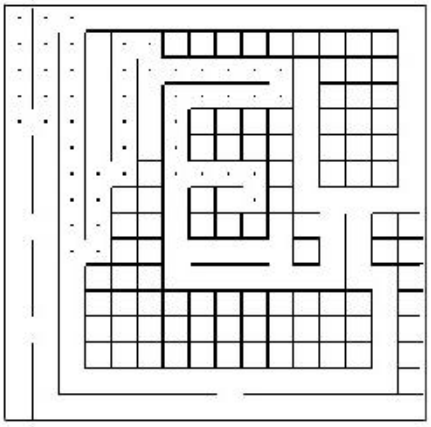
Result = [1 75 51 67]

4. Maze 2; No of bots = 4



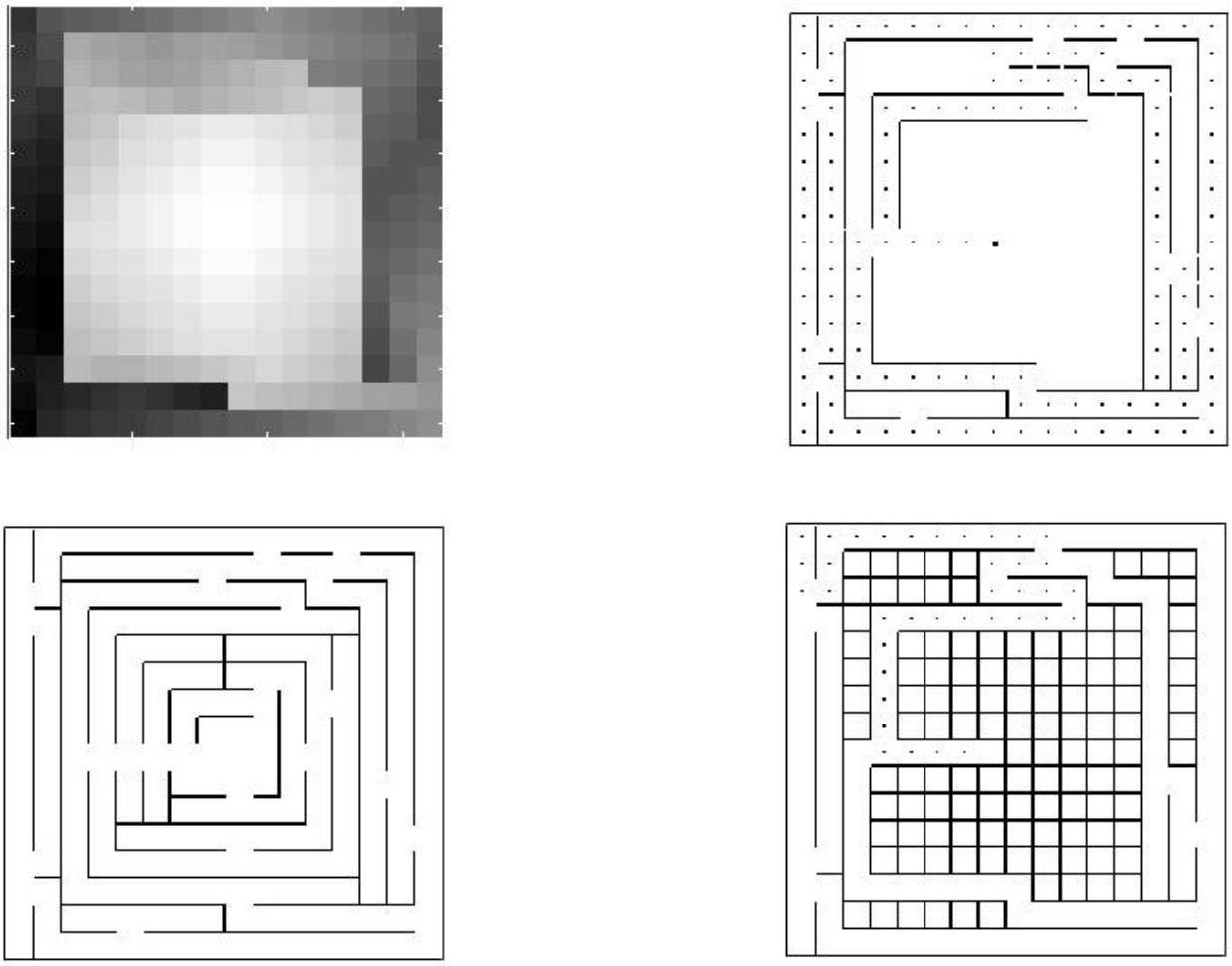






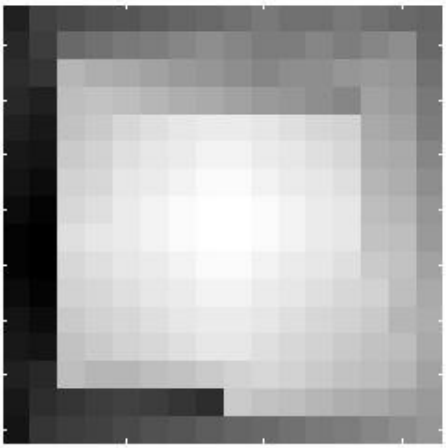
Result = [4 83 49 165]

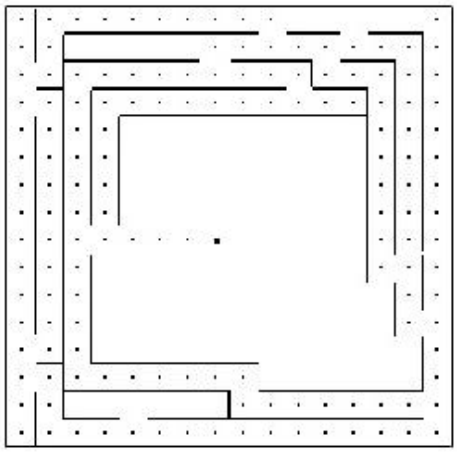
5. Maze 3; No of bots = 2

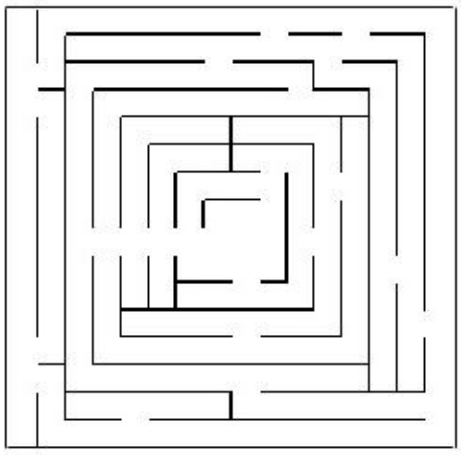


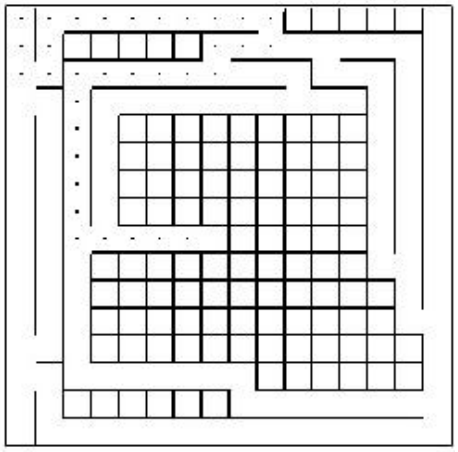
Result = [2 118 37 138]

6. Maze 3; No of bots = 4





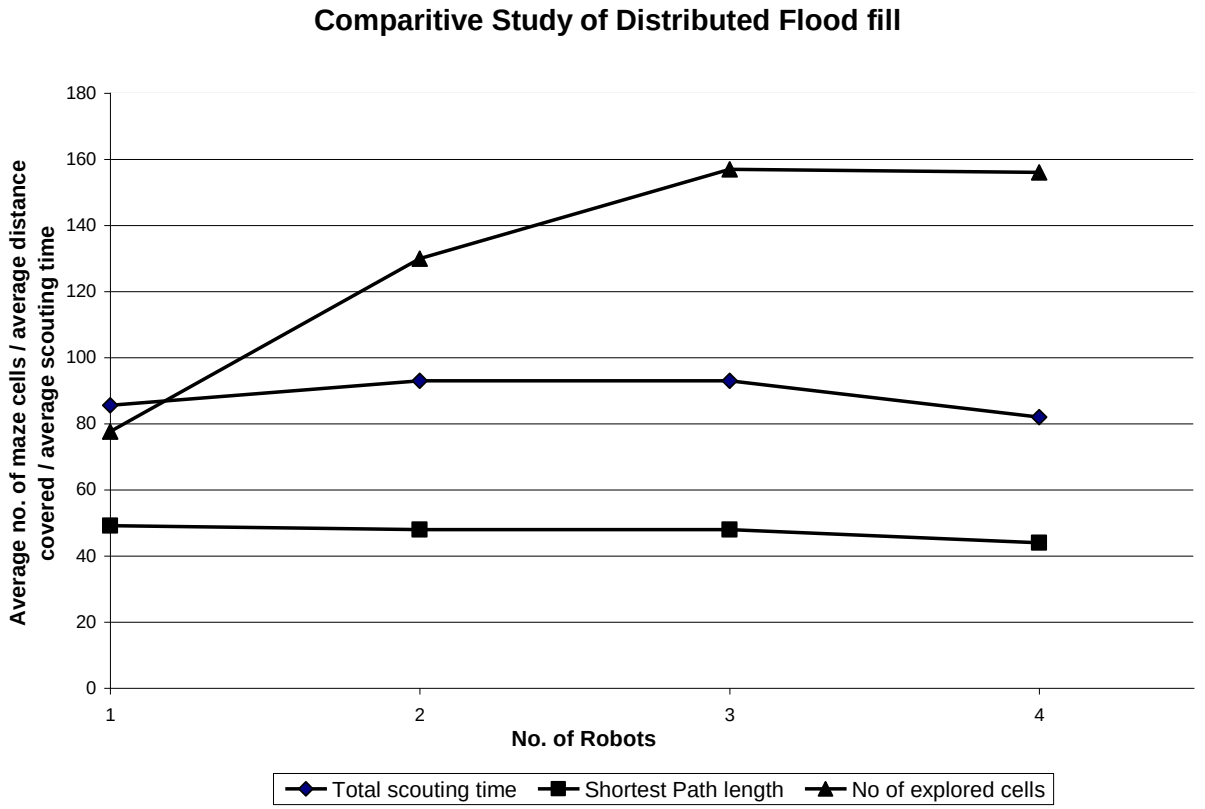




Result = [4 66 33 149]

A study was conducted on the advantage of using Distributed Flood fill over conventional Flood fill. The following chart is based on the results collected from 5 mazes chosen at random. The results are tabulated below:

No of bots	Average Scouting time	Average shortest path length	Average no. of cells explored
1	86	49.2	78
2	93	48	130
3	93	48	157
4	82	44	156



The result with one robot is same as the result of conventional Flood fill algorithm. We see that as the number of bots increase, the number of cells explored increases considerably without much increase in scouting time. Also the length of shortest path found over the explored area decreases as number of bots increase.

11. CONCLUSION

An initial study of the micromouse problem was done and analysis of the Flood Fill Algorithm, the most popular of the micromouse algorithm, carried out. Various other algorithms were also considered. Finally, decided to use a customized version of the Dijkstra's least cost algorithm incorporating some of the features of the Flood Fill Algorithm in the seventh semester. The Distributed Flood fill algorithm was developed and implemented in final semester. Motors and microcontroller were decided upon after considering all the available options. Proximity sensors were chosen based on the desired response and range. Detailed study of the MPLAB C18 compiler, which had to be used to program the microcontroller, was carried out. An initial prototype was constructed to test the mechanical strength and possible configurations and positioning of the motors and sensors. Simple programs for straight movement, turning and for reading from the sensors were written and tested on this prototype before moving onto developing the code of our least cost algorithm used to search for the centre of the maze given the starting block and orientation of the micromouse.

This prototype was fielded for a micromouse competition hosted by the College of Engineering, Thrissur in which we tied for the first place. Several useful observations were made from this exercise. The micromouse worked reliably and along expected lines as per the coded algorithm. The sensor range had to be toned down to prevent a wall which is far away from being detected and causing erroneous mapping of the maze. The micromouse developed an initial twitch before starting movement as the battery charge level came down. It was difficult to take a 180 degree turn within a cell unless it was very close to the centre of the cell due to the size constraints.

We have trimmed down the dimensions of the micromouse to solve the turning problem. This also makes it much easier to navigate the maze and makes the mouse more error tolerant regarding turning and positioning within the cell. The sensor gain was adjusted to control the range achieved within desired limits. So we successfully completed the work we had planned to do in seventh semester by building a single fully functional micromouse capable of traversing a maze and solving it using the algorithm aided by the sensor inputs to detect the presence of walls.

The final algorithm is a modified form of the conventional flood fill algorithm customized to optimize the advantage of using distributed robotics. In this we have multiple bots starting from different points of the maze and trying to solve the maze. It is seen that the explored maze area increases with the number of bots used. Increase in the explored area means better chances of finding an optimum path to the centre.

Some mazes have their shortest paths from the original intended starting position itself to the centre without having to visit the neighborhood of the other starting positions of our scouting bots, in which case the advantage of using distributed robotics is restricted. In all other cases it is observed that increase in the number of bots from 1 to 2 and 2 to 3 in the initial scouting section leads to a more optimal solution. Interestingly, increase in the number of scouting bots beyond a particular limit fails to yield further improvement.

One further improvement on conventional Flood fill algorithm is that the optimum shortest path is found out without much increase in scouting time. The total scouting time taken for distributed scouting is the time taken by the bot scouting maximum squares. If conventional Flood fill method was used, for scouting the same area, almost twice the time would be taken. Scouting more areas with less time opens better options for solving the maze. This factor becomes an advantage, especially while solving more complicated mazes. If Distributed Robotics is taken to real world scenarios this is would ultimately be the cornerstone of success.

12. APPENDIX 1: CODE SNIPPET

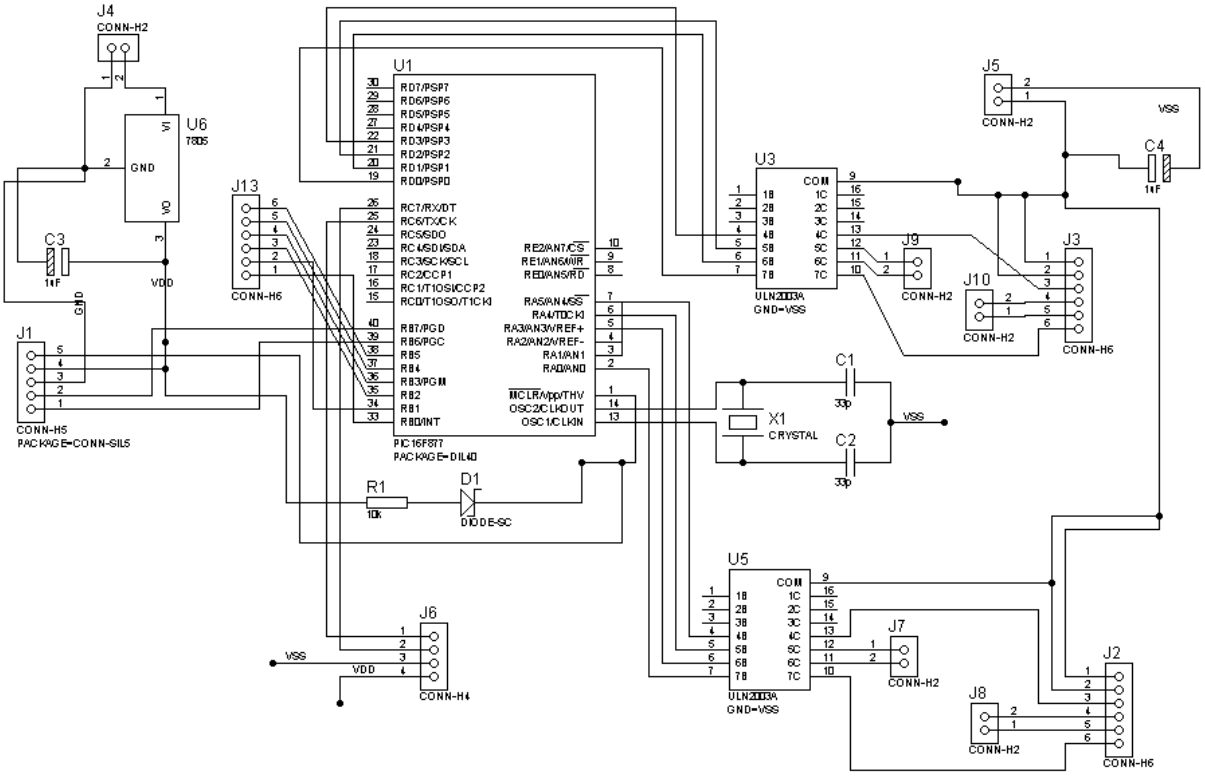
Flood fill code:

```
for(i=0;i<16;i++)          //value matrix initialisation
{
 for(j=0;j<16;j++) val[i][j]=0x00;
}

z=0x00;          //initialising value to 1
n=0x04;
a[0]=0x07; b[0]=0x07;
a[1]=0x07; b[1]=0x08;
a[2]=0x08; b[2]=0x07;
a[3]=0x08; b[3]=0x08;

do
{count=0;
for(k=0;k<n;k++)
          {
           i=a[k]; j=b[k];
           fill();
    }
 n=count;
 for(k=0;k<n;k++)
          {
           a[k]=c[k]; b[k]=d[k];
          }
 z++;
}while(count!=0);


void fill(void)
{
  i=a[k]; j=b[k];
  if(val[i][j]==0x00)          //check if already filled
    {
      val[i][j]=z;
    if(j!=15)
      {
       p=wall[i][j]/0x10;    //left wall
        if(p!=1)
          {
           if((!((i==7||i==8)&&((j+1)==7||(j+1)==8)))&&(val[i][j+1]==0))
                { c[count]=i; d[count]=j+1; count++;}
```

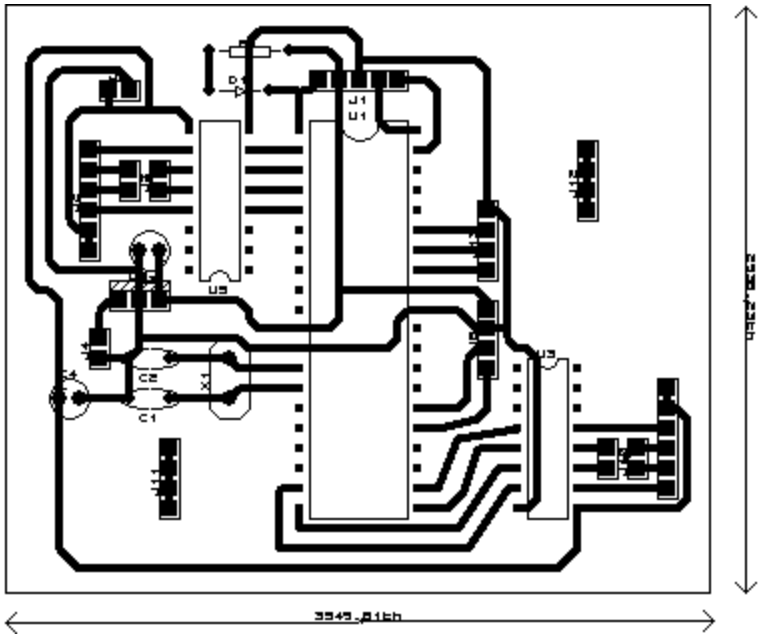
```
            }
        }
      if(i!=15)
        {
         p=wall[i][j]%0x10;    //straight wall
         if(p!=1)
           {
            if((!(((i+1)==7||(i+1)==8)&&(j==7||j==8)))&&(val[i+1][j]==0))
                    { c[count]=i+1; d[count]=j; count++;}
           }
        }
      if(j!=0)
        {
         p=wall[i][j-1]/0x10;  //right wall
          if(p!=1)
            {
             if((!((i==7||i==8)&&((j-1)==7||(j-1)==8)))&&(val[i][j-1]==0))
                    { c[count]=i; d[count]=j-1; count++;}
            }
        }
      if(i!=0)
        {
         p=wall[i-1][j]%0x10;  //back wall
           if(p!=1)
             {
              if((!(((i-1)==7||(i-1)==8)&&(j==7||j==8)))&&(val[i-1][j]==0))
                    { c[count]=i-1; d[count]=j; count++;}
             }
        }
      }
}
```

13. APPENDIX 2: CIRCUIT DIAGRAMS & PCB LAYOUTS

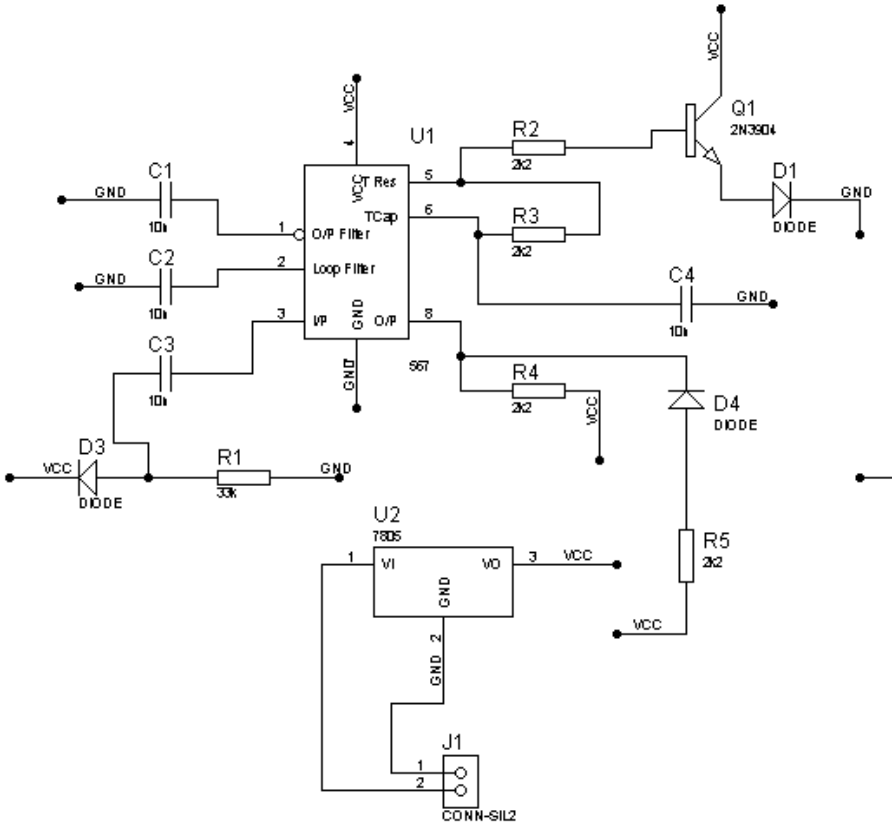
Main PCB Circuit Schematic



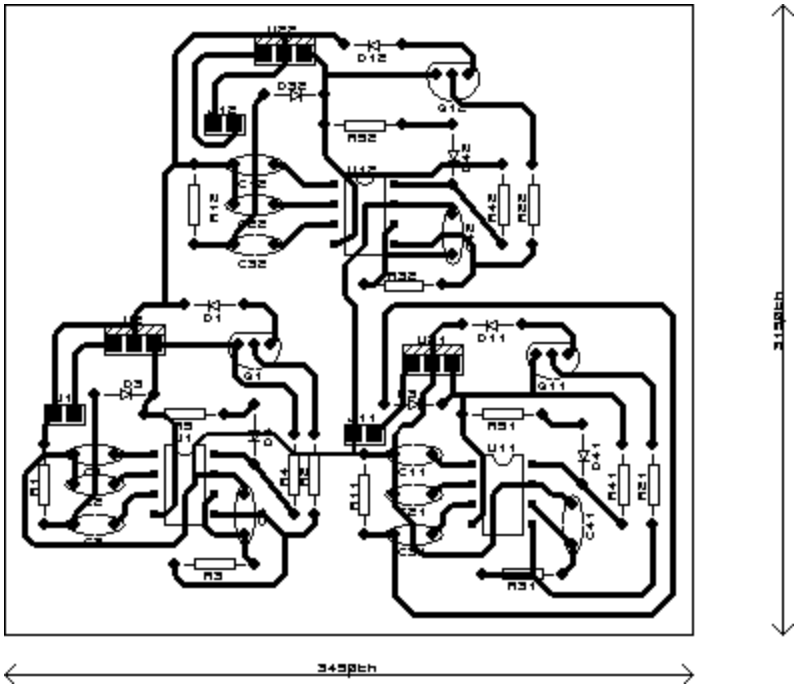
Main PCB Layout



Sensor PCB Circuit Schematic



Sensors PCB layout



13. REFERENCES

1. Bellman, Richard (1958), *On a Routing Problem*, in Quarterly of Applied Mathematics, 16(1), pp.87-90

2. Chen, Ning (1996), *An updated micromouse competition*, California State University in IEEE Frontiers in Education Conference.

3. Dimitri P. Bertsekas (March 1992). *"A Simple and Fast Label Correcting Algorithm for Shortest Paths". Networks, Vol. 23, pp. 703-709, 1993*. Retrieved on 2008-10-01.

4. Jin Y. Yen. (1970) *"An algorithm for Finding Shortest Routes from all Source Nodes to a Given Destination in General Network"*, Quart. Appl. Math., 27, 1970, 526–530.

5. Mishra, Swati and Bande, Panka, (2008), *Maze Solving Algorithms for Micro Mouse*, Signal Image Technology and Internet Based Systems.

6. Traunl, T, (Dec 1999), *Research relevance of mobile robot competitions*, Robotics & Automation Magazine, IEEE.

7. Websites:
   http://www.micromouseinfo.com/
   http://www.micromouseonline.com/
   http://en.wikipedia.org/
   www.robocet.com