Advanced Algorithms for Micro Mouse Maze Solving

Swati Mishra[1], Pankaj Bande[2]
[1]Inderprastha Engineering College, Ghaziabad, Uttar Pradesh, India
[2]LARE, Mumbai, Maharashtra, India

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 decide the appropriate logic for maze solving to give optimized results. This paper begins with the A* algorithm 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.*

Keywords: Mobile Robot Navigation, Algorithms, Micromouse, Flood fill, Djikstra, A*.

1. Introduction

Autonomous agents are mobile versatile machines capable of interacting with an environment and executing a variety of tasks in unpredictable conditions. Autonomy means capability of navigating the environment; navigation, in turn, necessarily relies on a topological and metric description of the environment [6].

One of the major components for the creation of autonomous robot is the ability of the robot to "plan its path" and in general the ability to "plan its motion". In a limited or carefully engineered environment it is possible to program the robot for all possible combinations of motions in order to accomplish specific task [2]. The problem of path planning is not confined to the field of robotics but its applications exist in various genres. For example, molecule folding, assembly/disassembly problems and computer animations are areas where comparable problems arise [7].

A robot is a mechanical device, which performs automated physical tasks, either according to direct human supervision, a pre-defined program, or a set of General guidelines using artificial intelligence techniques. There is a paradigm shift in engineering education from conventional classroom teaching to hands on projects, robotic projects are very useful for students who have to deal with an open ended problem and this way their creativity is stimulated. As project incorporates wide range of engineering fields, they can convert variety of theoretical study into practice. Furthermore, they learn teamwork.

To stimulate this learning process in upcoming engineering, a wide range of robotic competitions is held worldwide, and the most sought after among them is MICROMOUSE. It is one of the most efficient ways to inculcate the true "engineering values" in students. As mentioned above, the change in the mode of education is under revolution, as a result of which not much students are exposed to the field of robotics.

Traditional maze solving algorithms, while appropriate for purely software solutions, break down when faced with real world constraints. When designing software for a maze-solving robot, it is important to consider many factors not traditionally addressed in maze solving algorithms. Our information about the maze changes as we explore, so we must make certain assumptions about unseen portions of the maze at certain steps in most algorithms.

2. A* algorithm

A* is a type of search algorithm. Some problems can be solved by representing the world in the initial state, and then for each action we can perform on the world we generate states for what the world would be like if we did so. If you do this until the world is in the state that we specified as a solution, then the route from the start to this goal state is the solution to your problem.

A *node* is a state that the problem's world can be in. In this case, a node represents a cell of the maze. All the nodes are arranged in a *graph* where links between nodes represent valid steps in solving the problem. These links are known as *edges*. So the edges here are paths that lead from one cell to another. **A*** (pronounced "A star") is a best-first, graph search algorithm that finds the least-cost path from a given initial node to one goal node. The goal node here is the centre of the cell.

It searches first the routes that *appear* to be most likely to lead towards the center, and it also takes the distance already traveled into account. The algorithm traverses various paths from start to centre. For each node *x* traversed, it maintains 3 values; g(x) which is the actual shortest distance traveled from initial cell to current cell, h(x) which is the estimated (or "heuristic") distance from current cell to center, f(x)which is the sum of g(x) and h(x).

The *heuristic* is like an algorithm, but with a key difference. An algorithm is a set of steps which you can follow to solve a problem, which always works for valid input. A heuristic is not guaranteed to work but is useful in that it may solve a problem for which there is no algorithm. What we need is to use our heuristic at each cell to make an estimate of how far we are from the center.

2.1 The Algorithm

Starting with the initial cell, we maintain a priority queue of cells to be traversed. The lower the *f*(*x*) for a given cell *x*, the higher is its priority.

At each step of the algorithm, the cell with the lowest *f*(*x*) value is removed from the queue, the *f* and *h* values of its neighboring cells are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal cell has a lower *f* value than any cell in the queue (or until the queue is empty). Goal cell may be passed over multiple times if there remain other cells with lower *f* values, as they may lead to a shorter path to a goal. After the algorithm finishes its execution the *f* value of the goal is then the length of the shortest path. The algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal cell.

The nomenclature used in this algorithm is as follows.

Cell_traversed is the set of the cells already visited and evaluated

Cell_remaining is the set of the cells not evaluated so far.

Value_g represents the value of g(x) function

Value_h represents the value of h(x) function

Value_f represents the value of f(x) function

Cell_neighbor is the set of the adjacent cells to the cell under evaluation.

temp_value_g is the temporary value of g(x)

dist_btw(x, y) is a function that gives the distance between two adjacent cells. For uniform maze, this distance is constant.

better_temp_value_g is a variable that determines if the value of g(x) is better than the rest of the values of g(x).

came_from is the cell that the micromouse has come from after traversing.

reproduce_path(came_from, current_cell) is the function to reproduce the path traversed to reach the current cell from the cell desired.

The algorithm proceeds as follows;

```
STEP 1: Initialize
  cell_traversed = null
  Cell_remaining= initial cell
  Value_g[start_cell]=0
Value_h[start_cell]=heuristic_function(start_cell, center)
  Value_f[start_cell]=value_h[start_cell]
STEP 2: check if cell_remaining is empty
STEP 3: if yes then goto step  else goto step4
STEP 4: x = the cell in cell_remaining set having the lowest value_f[] value
STEP 5: check if x = centre
STEP 6: if yes then call function reproduce path
STEP 7: delete x from cell_remaining
STEP 8: add x to cell_traversed
STEP 9: check if wall is present in each of 4 directions.
STEP 10: if no then each cell is added as y to Cell_neighbor(x)
STEP 11: check if y is in cell_traversed
STEP 12: if no then goto step 13 else check for the next value of y in set cell_neighbor(x)
STEP 13: temp_value_g = value_g[x]+ dist_btw(x,y)
STEP 14: better_temp_value_g = 0 (false)
STEP 15: check if y is in cell_remaining if no then add y to cell_traversed
STEP 16:value_h[y] = heuristic_function(y, center)
STEP 17:better_temp_value_g=1
(true)
STEP 18: temp_value_g<value_g[y]
STEP 19:better_temp_value_g = 1 (true)
STEP 20: check if better_temp_value_g = 1
came_from[y] =x
value_g[y] = temp_value_g
value_f[y]=value_g[y]+value_h[y]
STEP 21: check for next value in cell_neighbor got step 12 until Cell_neighbor(x) is empty.
```

2.2 Drawbacks of the A* Algorithm

This algorithm may though be capable of solving many of the complex mazes or almost all of the mazes because it works on cell by cell basis and keeps a track of all the visited and non-visited cells so as to save time in revisiting the same cells over and over again. But the drawback of the algorithm lies in its dependence on the heuristic function to determine the distance of the current position from the destination cell i.e. the center. For simple mazes the heuristic function may be simpler but as the mazes get complex it becomes more and more complicated and the determination of the heuristic function becomes difficult. Therefore, the algorithm fails when it comes to complex heuristic functions. Also, determining heuristic functions is a time taking task and for different mazes it may be different.

Hence, we can modify this algorithm and then use it to reduce the drawbacks and get better results.

The time taken by the robot can be calculated. For the robot used here, the cell to cell movement time is 5 sec. and the turning time is 1 sec. so the total time taken by the robot to reach the center is (50*5) +(16*1)= 266 sec+ (extra time for traversing the other paths that lead to the center).

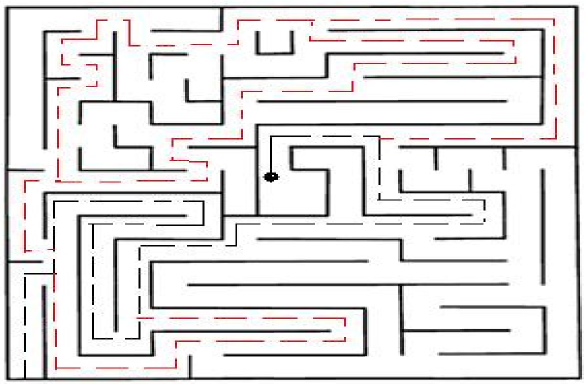

Figure 1: Maze solved by A*.(red dashes show the other paths found by the algorithm).

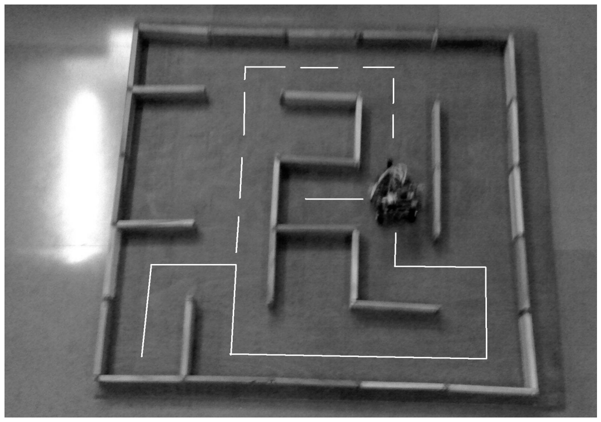

Image 1: Maze solved by A*(plane line shows the other path found by the robot)

3. Djikstra’s algorithm

So we observe that the design of an autonomous navigation system with multiple tasks to be accomplished in unknown environments represents a complex undertaking. Concepts from immune network theory are employed these days to convert an earlier reactive robot controller, based on learning classifier systems, into a connectionist device. Starting from no *a priori* knowledge, both the classifiers and their connections are evolved during the robot navigation. [1]

As a conclusion from the above fact we now move a step ahead and implement the Djikstra’s Shortest Path algorithm for solving our problem. The algorithm deals with finding the shortest path from a directed graph of a given set of nodes. This algorithm is basically a special case of A* algorithm where h(x)=0.

The approach adopts a strategy of multi-behavior coordination, in which a novel path-searching behavior is developed for determining the shortest path [11].

3.1. Maze Solving

The input of the algorithm consists of a weighted directed graph G and a source vertex s in G. We will denote V the set of all vertices in the graph G. Each edge of the graph is an ordered pair of vertices (u, v) representing a connection from vertex u to vertex v. The set of all edges is denoted E. Weights of edges are given by a weight function w: E → [0, ∞); therefore w(u,v) is the cost of moving directly from vertex u to vertex v. The cost of an edge can be thought of as (a generalization of) the distance between those two vertices. The cost of a path between two vertices is the sum of costs of the edges in that path. For a given pair of vertices s and t in V, the algorithm finds the path from s to t with lowest cost (i.e. the shortest path).So now this fundamental nature of the algorithm can be used to find the graph. The stepwise functioning of the algorithm has been described as below.

```
STEP 1: Start “ready set” with start node
Set start distance to 0, dist[$s$] =0;
others to infinite: dist[$i$]= (for $i$ $s$);
Set Ready = { }.
STEP 2: Select node with shortest distance from the starting point that is not in Ready set
Ready = Ready + {$n$}.
STEP 3: Compute distances to all of its neighbors
For each neighbor node $m$ of $n$
Check if dist[$n$] +edge ($n$, $m$) < dist[$m$]
If yes then dist[$m$] = dist[$n$] +edge ($n$, $m$);
STEP 4: Store path predecessors.
pre[$m$] = $n$;
STEP 5: Add current node to “ready set”.
STEP 6: Check if any node is left, if yes goto step 2
STEP 7: end.
```

Now the problem arises of how to generate the directed graph G of the nodes we have talked about so far. So for this we need to get the Micromouse traverse the whole maze and generate the nodes! So the traversal function is defined as follows.

```
Traverse ( ):
STEP 1: Move straight
STEP 2: Check if any wall is present in front. If yes, then goto step 3 else goto step 1.
STEP 3: check if the current location is present in V, if no then add the location in the set V, Calculate the distance between the previous and the present location. Store the value in the set E, else take 180 degree turn and traverse to the previous entry of V.
STEP 4: Check if wall is present on right. If present, take a 90 degree turn left if not then take a 90 degree turn right.
STEP 5: Goto step 1
Repeat steps 1 to 5 till the entire maze is traversed.
```

Calculation of distance between two consecutive nodes is done by adding a counter circuit at the base of the chasis near the wheels so as to count the number of cells between the destination and the source location. This number would denote the weight of the edge E.

Each location is represented by a cell of a 16x16 two dimensional array, cell being represented by [R, C], R represents row, and C represents column. So the vertex set V consists of a pair of variables [R, C] representing a single node.

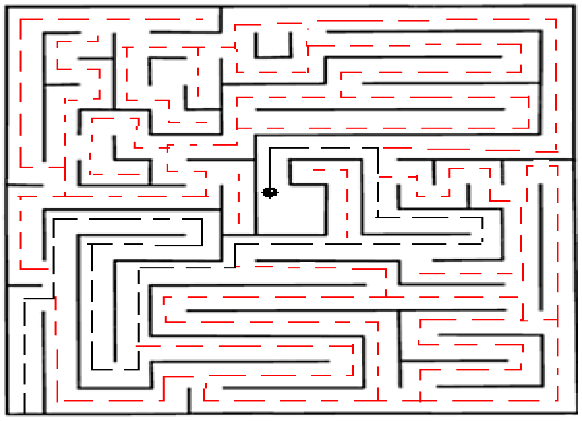

Figure 2. Maze as solved by Djikstra's algorithm.(red dashes show the maze travelled for map generation)

After generating the graph the final algorithm is applied on the graph and the shortest path reachable to the destination node which is the centre is obtained.

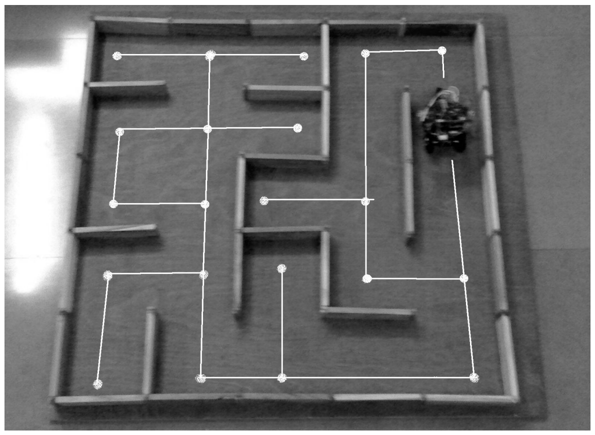

Image 2. Maze being solved by Djikstra's algorithm. White line shows the path traversed by the device and the white points show the nodes as perceived by the algorithm.

The time taken by the robot can be calculated. For the robot used here, the cell to cell movement time is 5 sec. and the turning time is 1 sec. so the total time taken by the robot is (50*5) +(16*1)= 266 sec+ (extra time for maze mapping) for reaching the center of this maze.

Now we see that it effectively calculates the shortest path but to find that path it has to travel the entire maze. This traversal is to generate the directed graph G. Though the time taken to reach the centre is less but it takes more time to traverse the maze. So if we calculate the total time taken, then it would be more in this case.

3.2 Drawbacks of the algorithm

There are, however, problems in using this algorithm, the major one being that the whole maze has to be traversed. For identifying the nodes, it is important to travel all the parts of the maze, irrespective of whether that portion of the maze contains the shortest path or not. Now, this is time consuming and also a lot of energy is wasted in the traversal. This problem would be solved if we can design a way where both the maze interpretation and path finding are done simultaneously.

The other problem is that for counting the number of cells to generate the edge set E, an additional hardware is involved which includes a counter circuit that counts the rotation of the wheels and hence the distance between two consecutive locations. This adds to the complexity of the design and increases the probability of error in input data from the external environment.

To avoid such complexities, we use yet another solution to solve our problem which has been described below.

4. Flood Fill algorithm

The speed of robot to find its path, affected by the applied algorithm, acts the main part in the present project. 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 [3].

The maze is represented as a 16x16 array in the memory. The centre is given the value (0, 0).all cells in its immediate vicinity are assigned 1, the cells next to it as 2, and so on. The array is divided into 4 symmetrical regions and then the assignment is done.

Upper left quarter, loop decrements the column, increments the row: R= R+j, C=C-i, i, j vary from 0 to 8.

Upper right quarter, loop increments the column, increments the row: R=R+j, C=C+i, i, j vary from 0 to 8.

Lower left quarter, loop decrements the column, decrements the row: R=R-j, C=C-i, i, j vary from 0 to 8.

Lower right quarter, loop increments the column, decrements the row: R=R-j, C=C+i, i, j vary from 0 to 8.

decr= 0, incr= 1
decc= 1, incc=0
	decr=0, incr= 1
	decc= 0, incc= 1

decr= 1, incr= 0
decc= 1, incc=0
	decr= 1, incr= 0
	decc= 0, incc=1

Figure 3. Values of the 4 variables in the 4 quadrants

So it is combined into a MATHEMATICAL EQUATION as follows:

Row increment and decrement: R-(i*decr) + (i*incr)
Column increment and decrement: C-(j*decc) + (j*incc)
Now when the assignment is done, the entire equation is as follows: (where the variables i, j vary in the loop from 0 to 8)

```
MAZE[R-(i*decr) + (i*incr)][C-(j*decc)+j*incc)]
= i+j;
```

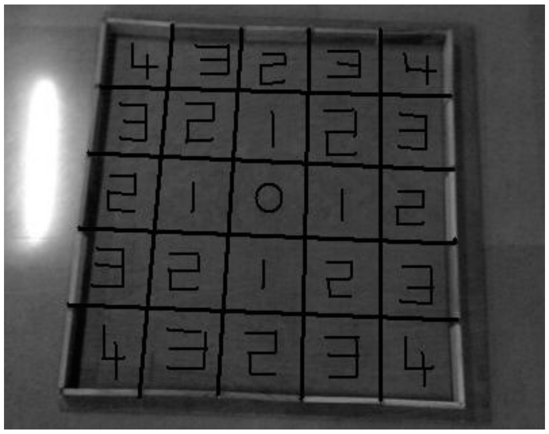
Image 3. An actual maze as depicted in the memory of the micro mouse.

Formation of array temp [4] for each cell:

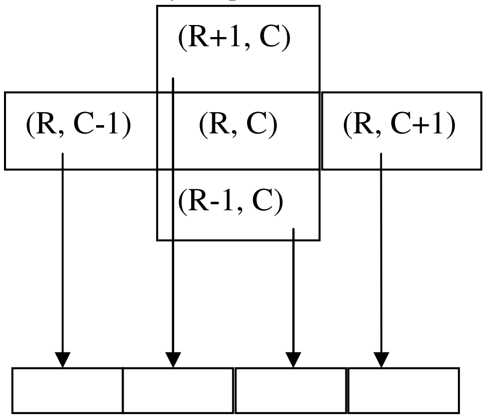

Figure 4. Storing elements in array; temp [4]

Each cell is interpreted as an array cell of a 2-d array and is represented with a value, R and C, which represents a row and column, respectively. The values, therefore, of the neighboring cells are as shown in the diagram. The values of the cell arrays are as according to the index values assigned to a 16x16 array in the computer memory. Initially the value of the first cell is assigned as, R=1, C=1.The values of the neighbors are stored in the above array. After the values are stored in the array, they are sorted using any kind of sorting. We have used here selection sort.

After this, the array is ready for further processing, which includes the deciding of which path to be taken and which values in the map to be changed.

The maze after being flooded is then traversed and the map of the maze is updated after every traversal. Every time a new cell is traversed, it creates the array described above and decides the lowest value nearby that can be traversed. The path followed is always from a higher value to a lower value.

```
Main():
START: form array temp[4] for Maze[R][C].
STEP 1: From the array, select the ith element, (i=1 initially)
STEP 2: If the value temp [i]<= Maze [R][C] , then go to step 4. If temp [i]>=Maze[R][C], call check(R,C).
Step 3: if the value temp [1]=256, turn 180 deg, i=i+1,goto step 1
 STEP 4: locate the cell of the value temp [i].
STEP 5: check if the wall is present in the way, if yes then, i++, go to step1
STEP 6: check if Maze[R1][C1]=temp[i+1], if yes, then use call Locate( R, C, temp(i+1)) algorithm defined below, to locate its cell.
STEP 7: store the result in (R2, C2)
STEP 8: call the function direction of move (R1, R2, C1, C2)
 STEP 9: move to Maze (R', C')
STEP 10: update value, R=R', C=C'
 STEP 11: check if Maze[R][C]=0, if yes, call return to start ( ), else go to START.
STEP 12: call follow ( ).
```

Decide which cell is preferable to move by using the following algorithm: (direction of move (R1, R2,C1,C2))

```
STEP 1: check which cell is obstructed by a wall
STEP 2: move in direction of no wall, if wall is present before all selected cells, then i=i+2, go to STEP
3: give the priority to forward straight movement.
STEP 4: if the wall is in front, move towards right or left, priority can be given to any direction.
STEP 5: if it is dead end, then move in backward direction turn 180deg.
STEP 6: return the decided value in (R', C').
```

Location of the value in array, which is first dealt with, is found by following routine:

```
STEP 1: Initialize: flag1=0, flag2=0, flag3=0, flag4=0
STEP 2: Locate(R, C, temp [i])
{check if temp [i]==Maze [R+1][C], if yes      flag1=1; R1=R+1, C1=C;
check if temp[i]==Maze[R-1][C], if yes
flag2=1;      R1=R-1, C1=C;
check if temp[i]==Maze[R][C+1], if yes
flag3=1;        R1=R, C1=C+1
check if temp[i]==Maze[R][C-1], if yes
flag4=1;        R1=R, C1=C-1            }
STEP 3:  Return (R1, C1)
```

The assignment of flags deals with the problem that arises when more than one cell has the same value.

```
Check (R, C, i)
{Step 1: Maze[R][C]=temp[i] + 1
 Step 2: update()
 Step 3: return to step 2 of main( )}
```

There is an updating based on the fact that the value of the cell near the center is always less than the value of the cell away from the center. So,

For 1<R<8, and 1<C<8, Maze[R+1][C]<Maze[R][C]
Maze[R][C+1]<Maze[R][C]

For 9<R<16, and 1<C<8, Maze[R+1][C]>Maze[R][C]
Maze[R][C+1]<Maze[R][C]

For 1<R<8, and 9<C<16,
Maze[R+1][C]<Maze[R][C]
Maze[R][C+1]>Maze[R][C]

For 9<R<16, and 9<C<16,
Maze[R+1][C]>Maze[R][C] Maze[R][C+1]>Maze[R][C]
So the equation which determines the update ( ) function is as follows:
If Maze [R-(i*decr) + (i*incr)][C-(j*decc)+j*incc)] >= Maze [R-(i*decr) + (i*incr)][C-((j+1)* decc)+((j+1)*incc)], then

Maze [R-(i*decr) + (i*incr)][C-(j*decc)+j*incc)]+1 = Maze [R-(i*decr) + (i*incr)][C-((j+1)* decc)+((j+1)*incc)], and

If Maze [R-(i*decr) + (i*incr)][C-(j*decc)+j*incc)] >= Maze [R-((i+1)*decr) + ((i+1)*incr)][C-(j* decc)+((j)*incc)], then

Maze [R-(i*decr) + (i*incr)][C-(j*decc)+j*incc)]+1 = Maze [R-((i+1)*decr) + ((i+1)*incr)][C-(j* decc)+(j*incc)]
Where i, j are variables varying from 0 to 8, in loop.

The given maze is solved using the flood fill logic. The time taken by the robot can be calculated. For the robot used here, the cell to cell movement time is 5 sec. and the turning time is 1 sec. then the total time taken by the robot is (50*5) +(16*1)= 266 sec. for reaching the center

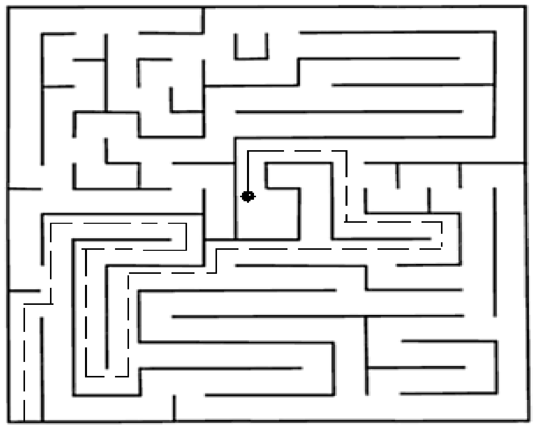
Figure 5. Maze solved through flood fill algorithm:

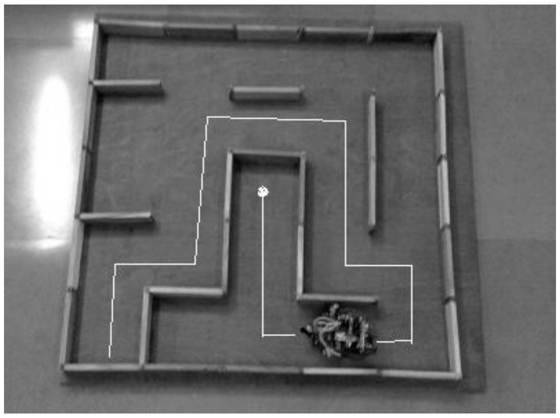
Image 4: Sample maze solved through flood fill algorithm:

White line shows the path followed by robot.

5. Result:

Motion planning is a key requirement demanded of autonomous robots. Given a task to fulfill, the robot has to plan its actions including collision-free movement of actuators or the whole robotic platform.[1]
A comparative study on the path length & time taken performance of our robot with regards to different algorithms is also done. Both simulation and real tests are performed.
The problem encountered in A* algorithm is solved by Djikstra's algorithm. And this algorithm is once again refined to Flood fill. The A* logic is restricted to a limited kind of the mazes only with simpler heuristic functions, while the Djikstra's algorithm can solve practically any kind of maze. But it requires a lot of time for maze interpretation and mapping which reduces its efficiency. So for situations in which the time is not the priority, this algorithm could serve the best. In case of A* algorithms, as well as the Flood fill algorithm, the maze interpretation or we can say map

generation is done along with the maze solving. Both the tasks performed together improve the efficiency of the micromouse robot. An elaborate analysis of the above algorithms gives us a basis of how to proceed in path planning, of intelligent devices capable of navigation. In contrast, to this in case of the Djikstra's algorithm, the maze interpretation is prior to maze solving.
For comparison, we have used a single virtual maze but for effective elaboration, the logics were verified on various mazes. Also we have used a standard IEEE maze for experimentation. For non uniform mazes, the same algorithms can be used with an external hardware attached that would keep a track of the distance travelled by the robot. The future work of this paper gives an emphasis on this problem.

6. Conclusion:

Hence we conclude that, if we don't have any time and hardware constraints we can effectively use the Djikstra's algorithm, but if both are the constraints then Flood fill would be superior to others. Further, if we do not wish to have any complex calculation to embed in the system, that means, if we have a memory constraint as well as heuristic function for the maze to be solved can be easily determined, then the A* algorithm would be best to use. The kind of logic implemented for robot path finding, depends upon the functional priority of the system.
The speed of robot to find its path, affected by the applied algorithm, acts the main part in the present projects that are concerned with robot navigation. While there is no limitation to improve the algorithms, there are some restrictions on developing robot's mechanic or electronic. Developing algorithm is usually cheaper than other parts.

7. References:

[1] Swati Mishra, Pankaj Bande;"Maze Solving Algorithms for Micromouse" IEEE conference, SITIS Bali, Indonesia, 2008.

[2] Renato Reder Cazangi, Associate, *Member, IEEE*, and Fernando J. Von Zuben, *Member, IEEE* **"**Immune Learning Classifier Networks: Evolving Nodes and Connections" , IEEE Congress on Evolutionary Computation ,Canada, 2006

[3] Dimitris C. Dracopoulos;"Robot Path Planning for Maze Navigation", 1998.

[4] Babak Hosseini Kazerouni, Mona Behnam Moradi and Pooya Hosseini Kazerouni;"Variable Priorities in Maze-Sloving Algorithms for Robot's Movement", 2003.

[5] Sung-Hee Lee, Junggon Kim, F.C.Park, Munsang Km, and James E.Bobrow;"Newton-Type Algorithms for Dynamics-Based Robot Movement Optimization", Digital Object Identifier, 2004.

[6] Horst-michael gross, Alexander Koenig; "Robust Omniview-basad Probilistic Self-loalization for Mobile Robots in Large Maze-like Environments", proceedings of the 17th International Conference on Pattern Recognition, ICPR-2004.

[7] Shinichiro Yano, Manabu Noda, Hisahiro Itani, Masayuki Natsume, Haruhiko Itoh and Hajime Hattori, Tadashi Odashima, Kazuo Kaya, Shinya Kataoka and Hideo Yuasa, Xiangjun Li, Mitsuhiro Ando, Wateru Nogimori and Takahiro Yamada; "A Study of Maze Searching With Multiple Robots System", 7th International Symposium on MicroMachine and Human Science, 1996.

[8] Frank Lingelbach; "Path Planning using Probabilistic Cell Decomposition", International Conference on Robotics & Automation, 2004.

[9] Javier Antich and Alberto Ortiz; "Extending the potential Fields Approach to Avoid Trapping Situations", CICYT-DPI-2001.

[109] Gorden Mc Comb, Myke Predko,"Robot Builder's Bonanza", Mc-Graw Hill, 2006.

[11] Thomas Braunl; "Embedded Robotics mobile robot design and applications with Embedded systems", Springer 2006.

[12]Meng Wang, James N.K. Liu, "Fuzzy Logic Based Robot Path Planning in unknown Environment, the Fourth International Conference on Machine Learning and Cybernetics, Guangzhou, 2005

[13] Roland Buchi, Gilles Caprari, Vladimir Vuscovic, Roland Siegwart; "A Remote Controlled Mobile Mini Robot", 7th International Symposium on MicroMachine and Human Science, 1996.