Analysis of Micromouse Maze Solving Algorithms

David M. Willardson
ECE 557: Learning from Data, Spring 2001

***Abstract*—This project involves a simulation of a mouse that is to find its way through a maze. There has been a competition around since 1980 called the "Micromouse Competition," that involves autonomous robotic mice that compete against time to find their way through a maze of predetermined size and dimensions. This project does not involve a robotic mouse, but a simulated mouse and maze, programmed in JAVA.**

I. Introduction

The Micromouse Competition involves a square maze of fixed size, with predetermined start and finish positions in the maze. The finishing point and starting points are "known" to the mouse before it starts. The starting point is always in a corner, and it is always surrounded by three walls. The maze is 16 blocks wide by 16 blocks high, and the finishing point is any one of the center four blocks.

Some mice rely on maze solving algorithms that provide a very small level of machine learning. Algorithms can involve "dead reckoning" at worst, which gives no guarantee that the mouse will find its way through the maze at all, much less in a competitive time. The typical algorithms involve a slight degree of machine learning. In these cases, any dead ends that are found can be blocked out, so that the finish point will eventually be found. In this case, the direction to turn at an intersection is often fixed, such as "always left" or "always right." This sometimes leads to mice that get stuck in infinite loops. Some have tried to improve on this by making the turn direction random, or to turn one way or the other depending on what quadrant of the maze the mouse is in. This sometimes helps the mouse solve the maze when it would have otherwise been unable, but it does not necessarily increase the average time to find the finish.

The better algorithms are "top-secret," meaning that they are not publicly available. It is suspected that they use some more advanced forms of machine learning, such as statistical decision theory. This simulation is an attempt to find the best algorithm for searching a maze using decision theory, depending on given parameters.

II. The Maze

The standard Micromouse maze is a 16x16 grid. Note that this is not a "perfect" maze, that is, the maze can have more than one route to the finish. In other words, loops are allowed, as are "rooms," which are closed-off areas.

Typical maze generators create only perfect mazes. It is difficult to write a maze generator that allows rooms and loops, while ensuring that the finished maze is actually solvable! The standard, mainstream maze generators are unsuitable for the Micromouse competition. The generation of mazes for simulation has been difficult as best, mostly since the mazes are anything but random; namely, they are deliberately chosen and engineered to make them as difficult as possible, thus to provide a challenge for designers.

This program makes an attempt to rectify this situation. A provision exists to allow a user-defined maze, so an actual Micromouse maze can be used if desired. A rudimentary generator was also created, but there is no guarantee that the generated mazes are solvable. It is recommended that any generated maze be examined before use, since an unsolvable maze will inevitably cause the program to be stuck in an infinite loop.

III. The Program

The program explores three different algorithms for searching a maze. They are as follows:

DEAD RECKONING:

This algorithm is used mainly as a control, or null hypothesis, for determining if the other two algorithms are better, based on statistical significance. The mouse simply goes straight until it encounters a fork or dead end. At a fork, it goes one of the available directions at random. At a dead end, it turns around.

DEAD END LEARNING:

This algorithm is similar to Dead Reckoning, except that any dead ends are remembered and a virtual wall is placed at the opening so that the mouse does not re-enter. This is the most common algorithm used for maze solving. It has the advantage of simplicity, but it is anything but optimal – maze designers often deliberately design mazes so as to disadvantage mice that use this algorithm. Furthermore, this algorithm cannot guarantee that the quickest route will be found if the maze has loops. Depending on how the algorithm is implemented, sometimes it will cause the mouse to go around loops several times before actually finishing.

How does this work? For example, suppose the mouse encounters a square with one entrance and three walls, as shown in Figure 1:

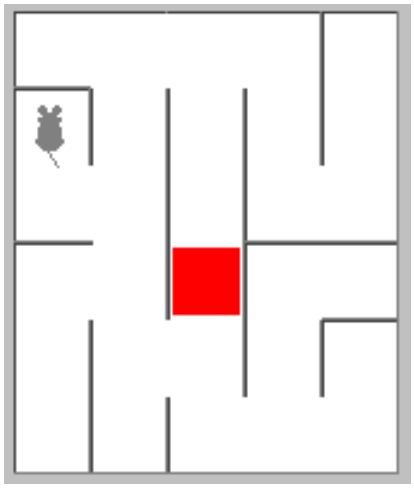

Figure 1: Mouse encounters a dead end.

The passage with three walls is one that the mouse has no reason to re-enter. So the mouse will turn around and move one space

forward, and the program will place a virtual wall there, creating a closed room that the mouse can never re-enter, as shown in Figure 2:

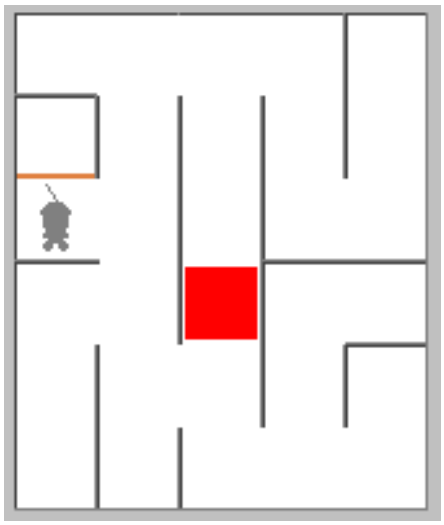

Figure 2: Mouse turns around a places a virtual wall.

Now we have created another three-walled square, and this can also be blocked off. The mouse can now turn to the left and move forward one space, and the program will place a virtual wall at that entrance as well, effectively blocking off the entire dead end, as shown in Figure 3:

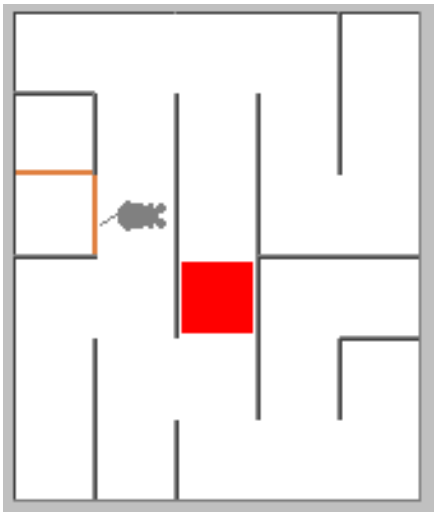

Figure 3: Continue placing walls until out of dead end.

This algorithm is the simplest one that also involves some degree of machine learning. This type of algorithm will solve any maze that is free of loops, but randomness has to be added so it can solve any maze, regardless of whether it has loops. The main advantage to this algorithm is that it is quite easy to modify the simple dead reckoning code to accommodate this extra feature. Even the same variables can be used. The only change is that the array that holds maze data is no longer static – we change it on the fly to account for dead ends.

FLOOD FILL:

This algorithm is much more complex than the previous two. It involves an initial assumption that there are no walls in the maze. A number is assigned to each cell. The number corresponds to the distance to the goal. This algorithm has the advantage of always finding the shortest path between start and finish. However, the shortest path is only in terms of distance; depending on the number of turns and the associated time to turn, the shortest path may not be the quickest. This concept is illustrated below in Figure 4:

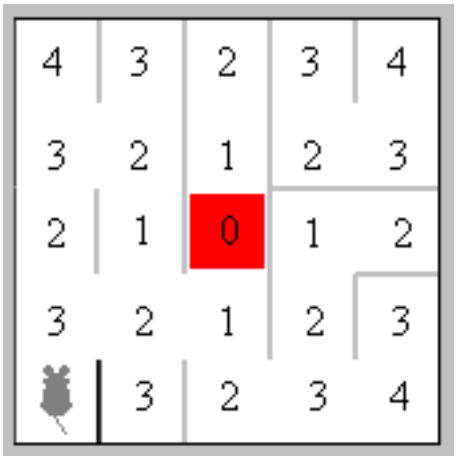

Figure 4: Maze seeded with optimistic distance values to the finish. Black walls are known to the mouse, but gray walls are not.

The mouse always starts in the corner, and there is always a wall to the left, as per the Micromouse Competition official rules. As in the previous example, the maze shown here is only 5x5 for simplicity's sake. A regular Micromouse maze is 16x16.

Note that the only walls that are known to the mouse at this point are the border walls and the wall to its right. The gray colored walls are currently unknown to the mouse. These will get updated as it searches the maze.

The number under the mouse is not shown, but in this case, it is number 4. Again, this number represents the most optimistic estimate of the number of moves it will take to get to the finish (which is the center square, which is always number 0.) In this case, as is common with mouse robots, the mouse can only see the walls of the space it is currently residing in.

The next step is for the mouse to move forward one cell. See Figure 5. This will usually be done in preference to turning, since turning takes additional time. Generally, turning should only be done if there a dead end is encountered.

The first check is to see if the mouse is at the finish. Since the number under the mouse is not zero, the mouse is not there, so it will check the numbers in adjacent squares to see if they are consistent. In this case, they are, so the next step is to move forward. See Figure 6.

The mouse has found a wall and it is remembered. It is colored black for illustration. The mouse is not at the destination square, so it checks the consistency of the numbers around it. The number under the mouse is 2, but it knows that there is a wall to the left now, so this is a violation of the algorithm – each number no longer represents the number of moves to the destination.

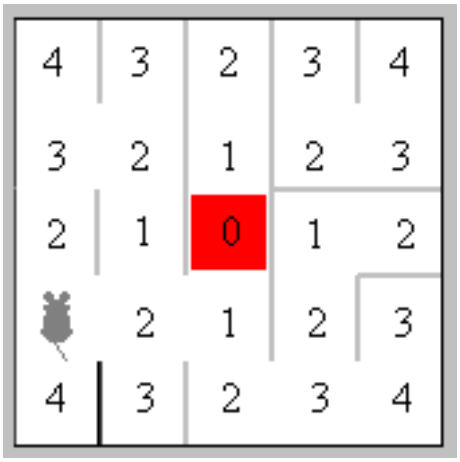

Figure 5: Mouse follows numbers in descending order.

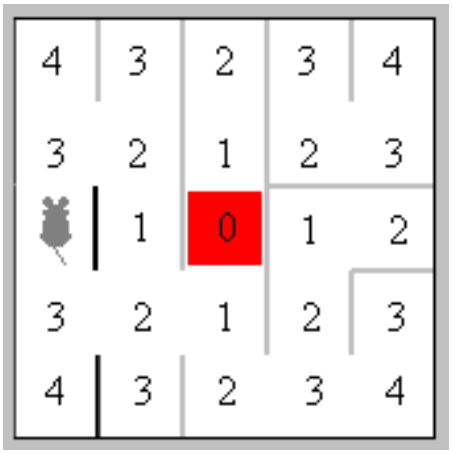

Figure 6: Mouse finds wall to right. Numbers are now inconsistent.

To remedy this, it checks the value of the square that it just arrived from and adds one to it, to make 4. This number is applied to the current square. Then it checks the square in front for consistency, and it is correct, at least as far as the mouse knows. If it was not correct, it would have to run through several iterations and update several numbers to maintain consistency throughout the maze.

The next step is to move forward. Again, the mouse does not know what is ahead of it, so it will move forward by default. Note the updated number in the square that the mouse just left, in Figure 7.

The next step would be to turn to the right and head toward the 2, then right again toward 1. Figure 8 illustrates that it may be necessary to change more than one number at once. There is more than one way to handle this. Either a stack can be set up to contain information on neighboring cells that need to be updated, or the brute-force method can be used: The mouse can simply scan its entire rendition of the maze in memory for consistency after every move, and if it finds a cell that is not consistent, it will change it and then repeat this cycle until all of the cells are consistent.

Several of the numbers need to be updated in this case. Figure 9 shows how the mouse's rendition of the maze should look after this action.

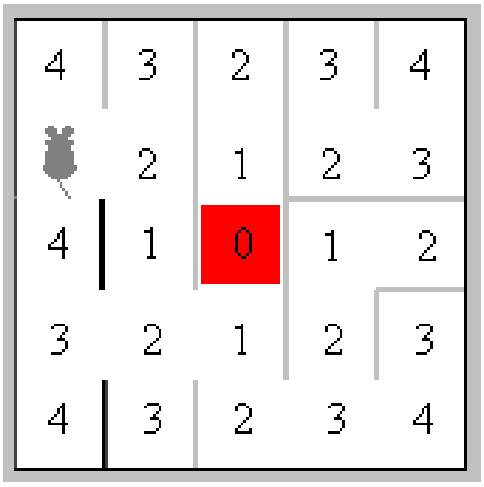

Figure 7: Numbers corrected, so mouse follows numbers in descending order.

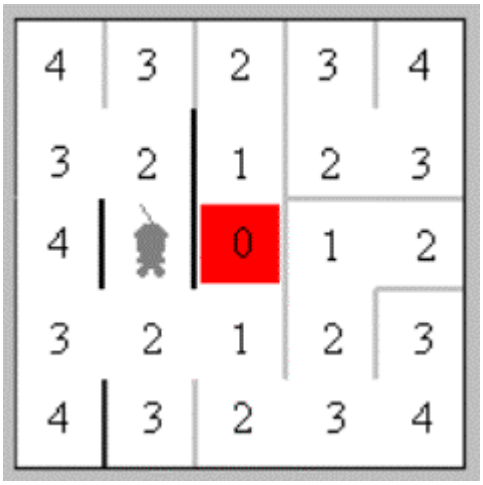

Figure 8: The number below the mouse is inconsistent.

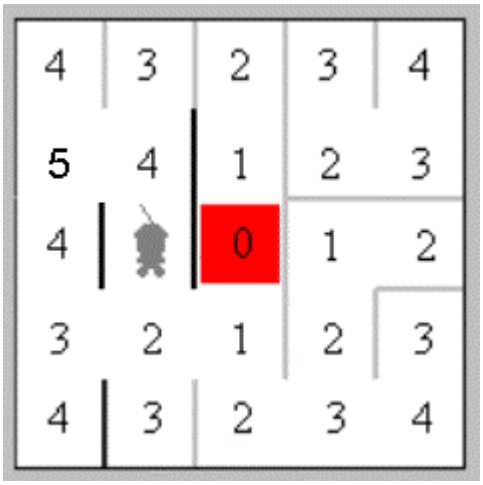

Figure 9: This change affects other numbers in the maze.

The flood fill algorithm has a disadvantage, in that it is nearly impossible to design into a state machine – a microcontroller is usually required. A simple microcontroller such as the Intel 8051 is

typically not powerful enough. Several kilobytes of RAM are usually required just to store the maze and wall data, not to mention the program itself. If the program is written in C, it will likely require even more overhead for the debugging information added to the end of the code.

The flood fill algorithm purposely prevents moving to numbers that are higher, even if that would have been a shorter path. This seems like a disadvantage, but there is no way of knowing the shortest path before the walls are mapped. Following the algorithm strictly will improve the average time to finish in any maze. This algorithm always works, and is not random – it is systematic and predictable. A maze could be deliberately designed to favor one of the previous algorithms, but this would not be expected from maze designers.

To further illustrate how this algorithm works, some pseudocode is below:

Stack method:

```
Create a 16x16 array to hold maze elements
Create a stack that can hold 256 entries
Put zero into the finish cells
Fill the rest of the cells with the most optimistic distance in cells
    to the goal (i.e. pretend there are no walls until found otherwise.)
Push the current cell onto the stack
While the stack is not empty:
    Pull a cell from the stack
    Is the cell's value one greater than the minimum value of its
       accessible neighbors?
    If yes, keep pulling cells and checking them.
    If no, change the cell's value to one greater than the minimum
       value of its accessible neighbors, and push all of the cell's
       accessible neighbors onto the stack.
End While
```

Brute-force method:

```
Create a 16x16 array to hold maze elements
Put zero into the finish cells
Fill the rest of the cells with the most optimistic distance in cells
    to the goal (i.e. pretend there are no walls until found otherwise.)
While the current cell's value is incremented:
    For every cell in the maze:
       Is the cell's value one greater than the minimum value of its
          accessible neighbors?
       If no, change the cell's value to one greater than the minimum
          value of its accessible neighbors.
    End For
End While
```

Either method will work, and both will do exactly the same thing. The best one to choose depends on processor speed and memory availability. For a slow processor with plenty of memory, the stack method is preferred, but for a faster processor and limited memory, the brute force method is preferred.

For a real Micromouse maze, which is 16x16 cells, in some cases it may be necessary to push nearly the entire maze onto the stack. This requires much more memory overhead than the brute force method. The reason is because three variables must be passed to the stack for each cell, that is, the cell's X-position, the cell's Y-position, and the cell's number.

The brute force method is simpler, and it is plausible that the main routine could be implemented in assembly language. This may make the mouse more efficient in terms of both memory use and processor overhead.

RUNNING THE PROGRAM:

The JAVA programming language is usually called from HTML, via the Internet. This allows the program to be accessed by anyone at any time, and from virtually any computing platform.

In the figure below, the maze shown is a model of the actual maze used in the worldwide Micromouse competition in 1986. The program has three parameters that can be passed to the JAVA applet on the fly. The three text boxes at the bottom indicate, from left to right, the time to move forward one cell, the time to turn 90º in place, and the actual time the simulation takes to perform either of the two previous tasks, all in milliseconds. The default values are all 100 milliseconds, which is on the fast side for a real mouse.

For recording data, I used 1000 milliseconds to move forward one cell and 500 milliseconds to turn. I estimated these values from a video of an actual competition that I viewed. I used 10 milliseconds for the actual time in the simulation. In the worst cases, mice could take in excess of an hour to finish a maze if viewed in real time, so the speedup factor here allows the simulation to complete in less than a minute.

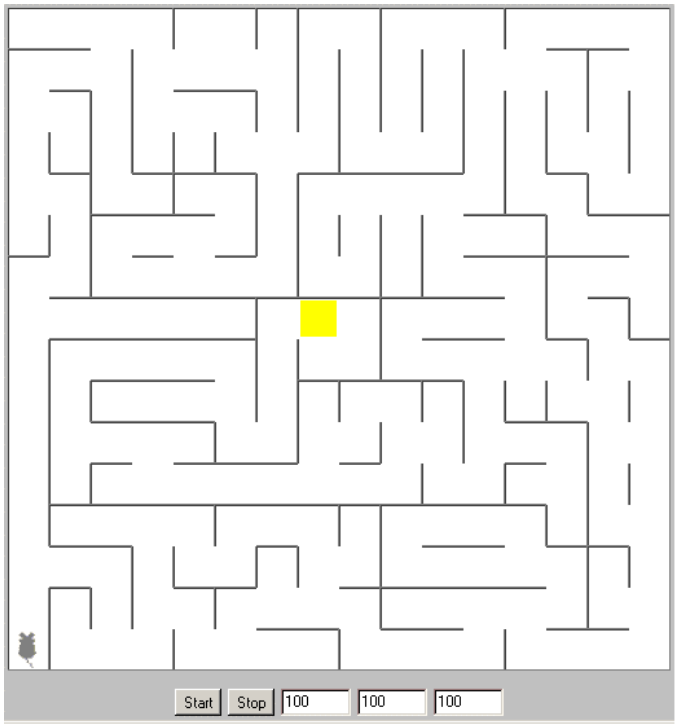

Figure 10: The Micromouse program used to simulate various mazes and algorithms. Maze is Chicago 1986.

IV. Results

The program was run with many different parameters, to test the viability of each algorithm.

Chicago 1986 maze:

Algorithm	Mean: (Seconds)	Standard Deviation: (Seconds)	Number of Samples:

Dead Reckoning	1167	601	30
Dead End Remembering	2043	1012	30
Flood Fill	153	0	1

The maze used in the 1986 Chicago competition is shown above.

Note that the number of samples for the flood fill algorithm was only one. This is because the algorithm produces the same results every time it is run.

Using statistical decision theory, I conclude that there is no statistical significance between Dead Reckoning and Dead End Remembering, however. This is due to the large variance in the performance of these algorithms.

However, the Flood Fill algorithm is better than both of the others with 95% confidence for this maze. This confidence can be roughly measured without normalizing the mean and standard deviations. All that is required is that the Flood Fill algorithm's time be more than about 1.64 standard deviations from the mean of other algorithms.

Although it is not statistically significant, the mean and variance for the Dead End Remembering algorithm is larger than the Dead Reckoning algorithm. The Dead Reckoning algorithm was meant as control. The reason for this discrepancy was determined from visual examination of the algorithms in action. It appeared that remembering shorter dead ends that are close to the finish give the mouse fewer opportunities to turn around. This means that the mouse had to travel through a greater portion of the maze before it encountered a junction that might make it turn around and head toward the finish.

Japan 1982 maze:

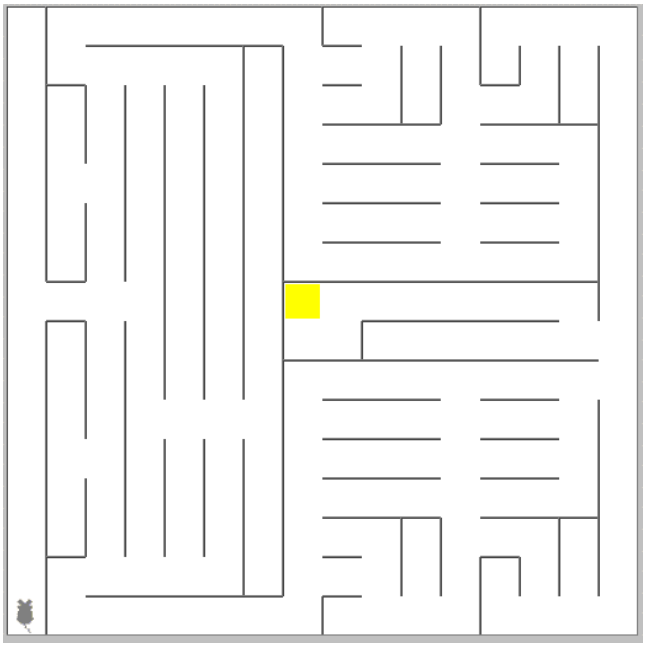

Figure 11: Japan 1982 maze.

Algorithm	Mean: (Seconds)	Standard Deviation: (Seconds)	Number of Samples:
Dead Reckoning	973	355	30
Dead End Remembering	755	289	30
Flood Fill	66.5	0	1

In this case, the opposite was true with respect to the first two algorithms. From visual examination of the maze, remembering dead ends does not appear to reduce the mouse's opportunities to turn around and find the finish. This observation, however, is not statistically significant.

However, I can conclude that the Flood Fill algorithm is better than the others with 99% confidence for this maze.

Random maze:

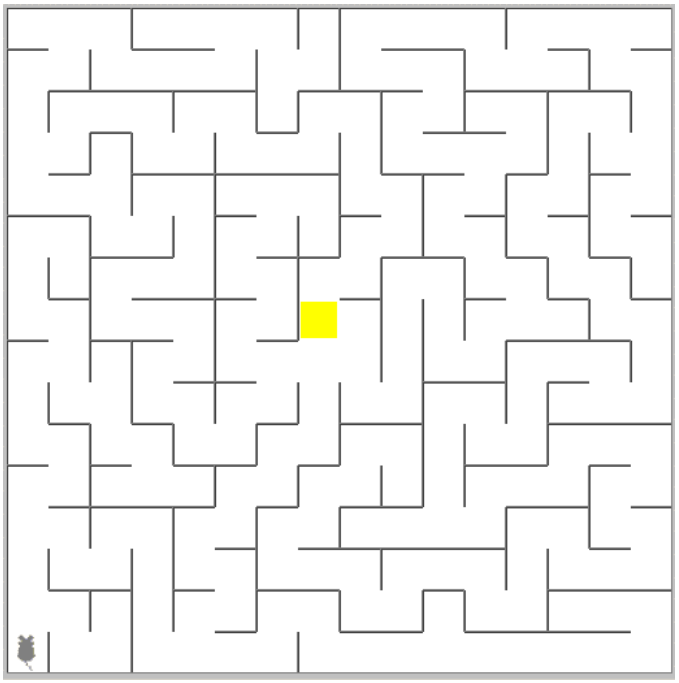

Figure 12: Randomly generated maze.

Algorithm	Mean: (Seconds)	Standard Deviation: (Seconds)	Number of Samples:
Dead Reckoning	131	82	30
Dead End Remembering	81	40	30
Flood Fill	126.5	0	1

In this case, the Flood Fill algorithm performed worse compared to the previous two mazes. Although the values are too close for any statistical significance, this finding shows that the Flood Fill algorithm is not better than the others in all cases. In this case, a simpler mouse that only remembers dead ends would probably solve a maze faster. This shows that mazes can be designed to favor one algorithm over another.

V. Suggestions for further study

It may help to further back up the Flood Fill algorithm's viability to test it with more actual Micromouse competition mazes. In this study, only professional level mazes were studied. Perhaps drastically different results would be produced for student level mazes, which may be simpler, and favor the Flood Fill algorithm less.

Instead of conducting multiple trials on the same randomly generated maze, many individual trials could be conducted on different random mazes. Also, a better random maze generator could be written.

VI. Conclusions

The Flood Fill algorithm provides a systematic and predictable way to solve a Micromouse maze. It appears to be better than simpler algorithms for mazes that are used in real competitions. Although it did not meet the mark for a randomly generated maze, it should be noted that Micromouse mazes are never randomly generated. Rather, they are deliberately designed. It would provide less of a challenge to the Micromouse designers if the maze were designed to favor simpler maze solving algorithms.

Although the Flood Fill algorithm was statistically better than the other two algorithms tested for the two actual mazes, this may not be the case with other mazes.

VII. Bibliography

Micromouse Introduction. By Peter Harrison.
http://www.cctc.demon.co.uk/micromouse/index.htm

UC Davis Micromouse Home Page.
http://www.ece.ucdavis.edu/umouse/