Quantitative Comparison of Flood Fill and Modified Flood Fill Algorithms

George Law

Abstract—**Flood fill algorithm has won first places in the international micromouse competitions. To save computation time, the modified flood fill algorithm is often coded. Yet, the literature search reveals the scarcity of the quantitative details of the two algorithms. This article attempts to discuss their differences and uses Maze-solver simulator to collect and tabulate the maze-run statistics for various popular mazes. It will discuss these statistics and various aspects of the flood fill algorithm modifications.**

***Index Terms*—Flood fill algorithm, maze simulation, micromouse competition, modified flood fill algorithm.**

I. Introduction

Each year since 1972, the micromouse competitions have been held in cities and university campuses all around the world. The participants, students and engineers, design and program their micromice to autonomously find the center of a 16 by 16 cell maze within 10 minutes. After finding the center, the micromouse may map the entire maze to locate the shortest route to the center. Using the shortest route, the micromouse will attempt to reach the center in a fastest run.

The maze-solver simulator simulates the mouse run in various popular mazes such as the one used in Japan 2011 competition and other competitions. For each algorithm, the simulator tabulates the total cell traverses, number of distance updates, number of corner turns, etc. Luke Last, et al. [1] codes the Flood Fill algorithm in Java; the author augments and revises their Flood Fill algorithm and codes the Modified Flood Fill algorithm [2].

II. Maze Solving Algorithms

Many maze solving algorithms are readily available. With less demand for computation speed and with the assurance that the best run gives the smallest number of cells travelled, the modified flood fill algorithm is, by far, the most commonly used one in micromouse competitions.

A. Flood Fill Algorithm

The Flood Fill algorithm uses the concept of water always flowing from a higher elevation to a lower one [3][4]. It applies this concept by assigning each cell in the maze a value that represents how far the cell is from the center. The cells with higher values are considered to have higher elevations; the ones with lower values are considered to have lower elevations. The center cells are assigned zero values which are equivalent to the lowest elevations.

Manuscript received August 9, 2012; revised December 15, 2012. This work was supported in part by the Department of Electrical and Computer Engineering, California State University, Northridge and by the IEEE CSUN Chapter.

G. Law is with the California State University, Northridge, CA 91330, USA (e-mail: George.law@csun.edu).

For simplicity in illustration, instead of using a 16×16 cell maze, we shall use a 6x6 cell maze as shown in Fig. 1a. The concept remains the same except that it applies to a smaller region of cells.

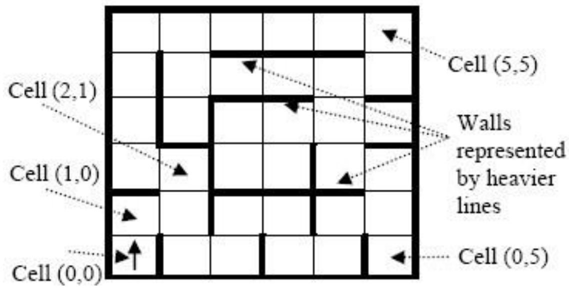

Fig. 1a. 6×6 cells sample maze

We shall use the matrix notation (x, y) for the cell location where x is the row value, x = 0 is the bottom row, and x = 5 is the top row; y is the column value, y = 0 is left-most column, and y = 5 is the right-most column. The micromouse starts at cell (0, 0) which is the cell at the lowest left-hand corner. Initially, the micromouse is placed at (0, 0) facing upward, as shown by the arrow. Before the micromouse starts its exploration, the maze is assumed to have no walls and each cell is assigned a value based on number of cell distance from the center. Fig. 1b shows the initial cells' distance values for a 6x6 cell maze.

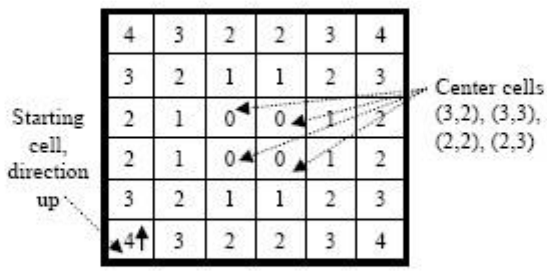

Fig. 1b. Initial distance values from center cells

The maze for any IEEE micromouse competition always has the east wall at the starting cell (0, 0) and the first move is always upward (north). At cell (0, 0), the cell has only one open neighbor (1, 0) which has a smaller distance from the center. Hence the micromouse will moves upward (north), from a higher elevation to a lower one, as shown in Fig. 1c. With the new wall detected north of cell (1, 0) and the old wall east of cell (0, 0), the Flood Fill algorithm floods the maze with the distances from the center cells as shown in Fig. 1c.

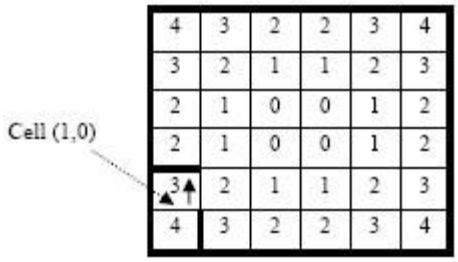

Fig. 1c. Distance values when micromouse reaches cell (1, 0)

At cell (1, 0), the micromouse encounters the north wall. This cell has two open neighbors (0, 0) and (1, 1). Since the open east neighbor (1, 1) has a distance smaller than cell (1, 0), the micromouse will turn and moves to (1, 1) eastwards, again from a higher elevation to a lower one, as shown in Fig. 1d. With the new wall detected east of cell (1, 1) and the old walls north of cell (1, 0) and east of cell (0, 0), the Flood Fill algorithm floods the maze with the distances from the center cells as shown in Fig. 1d. Refer to Fig. 1a for the other walls which are yet to be detected.

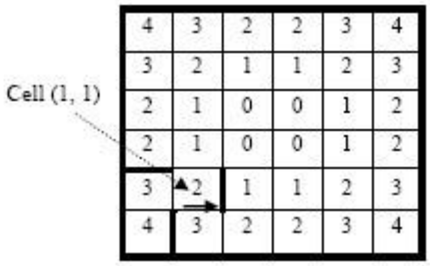

Fig. 1d. Distance values when micromouse reaches cell (1, 1)

At cell (1, 1), the micromouse encounters the east wall. This cell has three open neighbors (0, 1), (2, 1) and (1, 0). Since the open north neighbor (2, 1) has a distance smaller than cell (1, 1), the micromouse will turn and moves to (2, 1) northwards, again from a higher elevation to a lower one, as shown in Fig. 1e. With the new walls detected north of cell (2, 1) and east of cell (2, 1), and the old walls east of cell (1, 1), north of cell (1, 0), and east of cell (0, 0), the Flood Fill algorithm floods the maze with the distances from the center cells as shown in Fig. 1e.

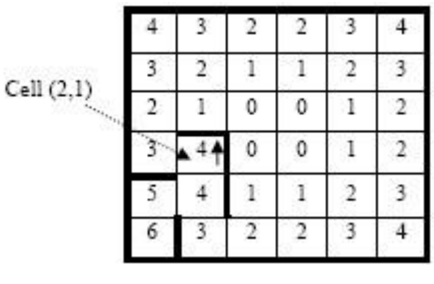

Fig. 1e. Distance values when micromouse reaches cell (2, 1)

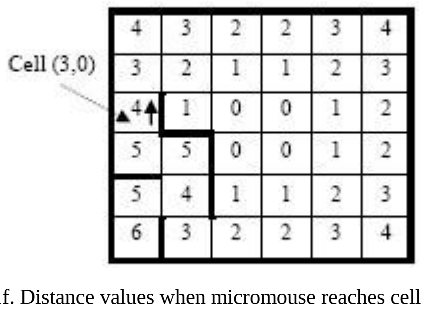

Fig. 1f. Distance values when micromouse reaches cell (3, 0)

The maze flooding is done each time the micromouse reaches a new cell. Again, when the micromouse reaches cell (3, 0), the distance values, as the result of maze flooding, is shown in Fig. 1f.

The same process continues until the micromouse reaches the center cell. Overall, to reach the center on the first run for this sample maze, the mouse will traverse a total of 24 cells. Because the Flood Fill algorithm floods the maze when the mouse reaches a new cell, including the starting cell, the total number of floodings performed is 24, which translates into a total of 24*36 = 864 cell distances updated. Fig. 1g shows the first run path which passes through 24 cells to reach the center.

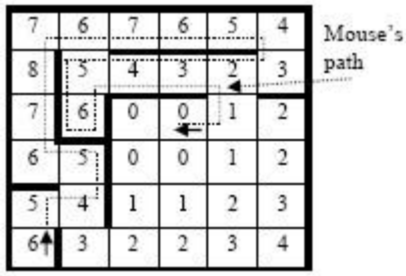

Fig. 1g. First run path dotted lines show micromouse's path to reach the center the 1st time

B. Modified Flood Fill Algorithm

The modified flood fill algorithm does **not** flood the maze each time a new cell is reached. Instead it updates only the relevant neighboring cells using the following revised recursive steps:

1) Push the current cell location (x, y) onto the stack.
2) Repeat this step while the stack is not empty.

- Pull the cell location (x, y) from the stack.
- If the minimum distance of the neighboring open cells, md, is not equal to the present cell's distance - 1, replace the present cell's distance with md + 1, and push all neighbor locations onto the stack. This revised distance update algorithm differs from the recursive steps outlined by S. Benkovic in http://www.micromouseinfo.com/ introduction /mfloodfill.html [5]. The Appendix discusses the differences.

Like the flood fill algorithm, the maze is first initialized (flooded) with the distances from the center with the assumption that there is no wall in the maze. Since there is no distance update until the micromouse reaches cell (2,1), the distance values remain unchanged as in Fig. 2a (same as Fig. 1d). Recall that neighboring cells' distances are updated only if the condition specified in the recursive step 2b is satisfied.

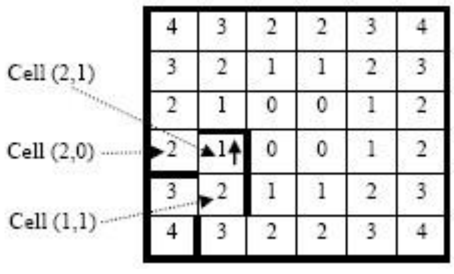

Fig. 2a. Distance values when micromouse reaches cell (2, 1) before distance update

Since the distance of the current cell (2, 1) – 1 = 0 is not equal to the minimum of the open neighbors (2, 0) and (1, 1), which is 2, the distance update is necessary. We shall follow the revised recursive distance update steps to see how the distance values are updated.

3) Push the current cell location (2, 1) onto the stack.

- Pull the cell location (2, 1) from the stack.
- Since the distance at (2, 1) – 1 = 0 is not equal to md = 2, the minimum of its open neighbors (2, 0) and (1, 1), update the distance at (2, 1) to md + 1 = 2 + 1 = 3. Push all neighbor locations (3, 1), (2, 0) and (1, 1), except the center location (2, 2), onto the stack. Fig. 2b shows the updated distances and Fig. 2c shows the current stack contents.

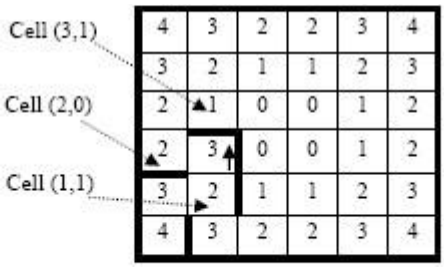

Fig. 2b. Distance values when micromouse reaches cell (2, 1) after distance update at cell (2, 1)

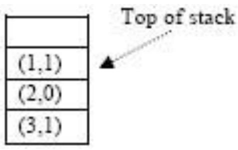

Fig. 2c. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (2, 1)

- Recursively, since the stack is not empty, pull the cell location (1, 1) from the stack.
- Since the distance at (1, 1) – 1 = 1 is not equal to md = 3, the minimum of its open neighbors (2, 1), (0, 1) and (1, 0), update the distance at (1,1) to md + 1 = 3 + 1 = 4. Push all neighbor locations (2, 1), (0, 1), (1, 0), and (1, 2) onto the stack. Fig. 2d shows the updated distances and Fig. 2e shows the current stack contents.

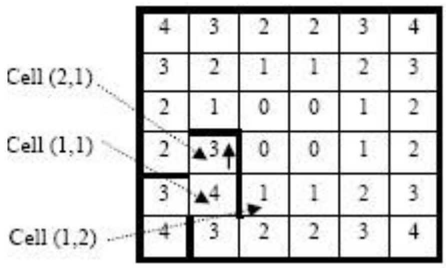

Fig. 2d. Distance values when micromouse reaches cell (2, 1) after distance update at cell (1, 1)

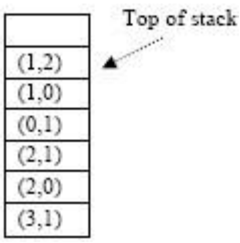

Fig. 2e. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (1, 1)

- Recursively, since the stack is not empty, pull the cell location (1, 2) from the stack.
- Since the distance at (1, 2) – 1 = 0 is equal to md = 0, the minimum of its open neighbors (2, 2), (0, 2) and (1, 3), no distance update is necessary.
- Recursively, since the stack is not empty, pull the cell location (1, 0) from the stack.
- Since the distance at (1, 0) – 1 = 2 is not equal to md = 4, the minimum of its open neighbors (0, 0) and (1, 1), update the distance at (1, 0) to md + 1 = 4 + 1 = 5. Push all neighbor locations (2, 0), (0, 0), and (1, 1) onto the stack. Fig. 2f shows the updated distances and Fig. 2g shows the current stack contents.

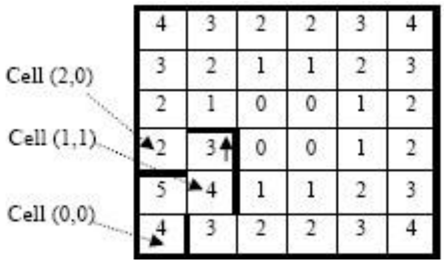

Fig. 2f. Distance values when micromouse reaches cell (2, 1) after distance update at cell (1, 0)

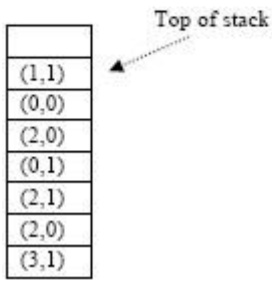

Fig. 2g. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (1, 0)

- Recursively, since the stack is not empty, pull the cell location (1, 1) from the stack.
- Since the distance at (1, 1) – 1 = 3 is equal to md = 3, the minimum of its open neighbors (1, 0), (2, 1) and (0,1), no distance update is necessary.
- Recursively, since the stack is not empty, pull the cell location (0, 0) from the stack.
- Since the distance at (0, 0) – 1 = 3 is not equal to md = 5, the minimum of its only open neighbor (1, 0), update the distance at (0, 0) to md + 1 = 6. Push all neighbor locations (1, 0) and (0, 1) onto the stack. Fig. 2h shows the updated distances and Fig. 2i shows the current stack contents.

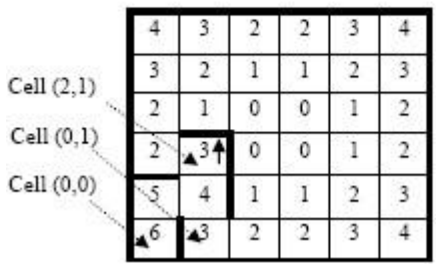

Fig. 2h. Distance values when micromouse reaches cell (2, 1) after distance update at cell (0, 0)

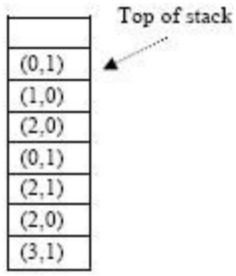

Fig. 2i. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (0, 0)

We shall not show the similar steps for the remaining cell locations in the stack. One can follow the previous steps, to obtain the distance values. When the stack is empty, the distance map of the maze will look like Fig. 2j.

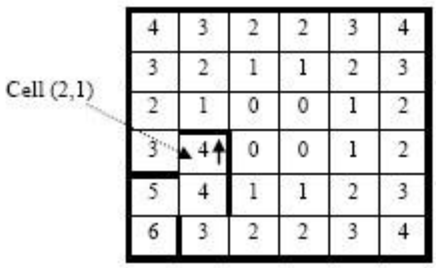

Fig. 2j. Distance values when micromouse reaches cell (2,1) after all neighbor cells' distances have been updated

The same process continues until the micromouse reaches the center cell. Fig. 2k shows the distance map when the micromouse reaches the center cell. Overall, to reach the center on the first run for this sample maze, the total number of distance updates is 36, which is obtained from the maze-solver's tabulated value.

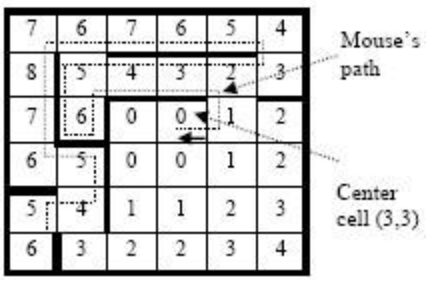

Fig. 2k. Distance values when micromouse reaches the center cell on the 1st run

TABLE I: TOTAL NUMBER OF DISTANCE UPDATES TO FIND THE BEST RUNS

Maze Names	Total number of cell traversed to find best run		Total number of distance updates to find best run	
	Flood Fill	Modified Flood Fill	Flood Fill	Modified Flood Fill
IEEE Region 6 2012	530	530	135,936	3197
Sample 16×16 cell Maze	179	179	46,080	793
APEC 2002	322	322	82,688	3307
Seoul 2002	585	585	150,016	3887
Minos 2003	246	246	63,232	2643

III. DISTANCE UPDATE COMPARISON

We use the Maze-Solver simulator to run the micromouse until the best run is found and to determine the total distance updates. For Flood Fill and Modified Flood Fill algorithms, Table I shows the total number of cells traversed and the total distance updates to find the best run for various mazes used in competitions. As could be expected from flooding the maze when the micromouse reaches a new cell, the total distance updates for Flood Fill algorithm is many fold larger than the Modified Flood Fill algorithm.

IV. SIMULATED RUN ON A SAMPLE 16×16 CELL MAZE

A. First Fast Run to Get to the Center

When the mouse finds its way to the center for the first time, a short path to the center that traces sequentially from the highest distance at the starting cell (0,0) to the lower ones and ending at the center has been mapped. This mapped path is shown in Fig. 3a as the gray path. If this first run mapped path is used in the speed run, the micromouse will start at the starting cell (0,0) and follow the gray path to get to the center.

B. Second Fast Run to Get to the Center

After reaching the center and using the walls and dead ends which are already marked on first run, the micromouse retraces from the center, following the lower distances and updating distances when necessary as it explores, back to the starting cell and starts the second run. Based on the newly updated cells and walls, the mouse will follow the Modified Floodfill algorithm to reach the center. When the mouse finds its way to the center for the second time, a second short path to the center that traces sequentially from the highest distance at the starting cell (0,0) to the lower ones and ending at the center has been mapped. This second mapped path is shown in Fig. 3b as the gray path. If this second run mapped path is used in the speed run, the micromouse will follow the gray path to get to the center.

C. Third Fast Run to Get to the Center

After reaching the center and using the walls and dead ends which are already marked on first and second runs, the micromouse retraces from the center, following the lower distances and updating distances when necessary as it explores, back to the starting cell and starts the third run.

Similar to the first and second runs, the third run will find a third short path to the center that traces sequentially from the highest distance at the starting cell (0,0) to the lower ones and ending at the center. This third short path is shown in Fig. 3c as the gray path.

The simulation shows that the subsequent runs follow the third run's path, which is an indication that it has found the best run. This is confirmed by the smallest number of cells traversed on this run. If priority is given to the smaller number of turns, the second run will be the best run. When the micromouse uses this best run for its speed run, it just follows the already mapped distances from the starting cell to the center cell, like water flowing from the higher elevation to the lowest elevation.

The total number of cells traversed and the total number of turns taken for all three fast runs are tabulated in Table II.

TABLE II: NUMBER OF CELLS TRAVELED AND TURNS TAKEN

ALL JAPAN 2011 SIMULATED RUN		
RUN	NUMBER OF CELLS TRAVELLED	NUMBER OF TURNS TAKEN
1ST	33	17
2ND	33	15
3RD	31	19

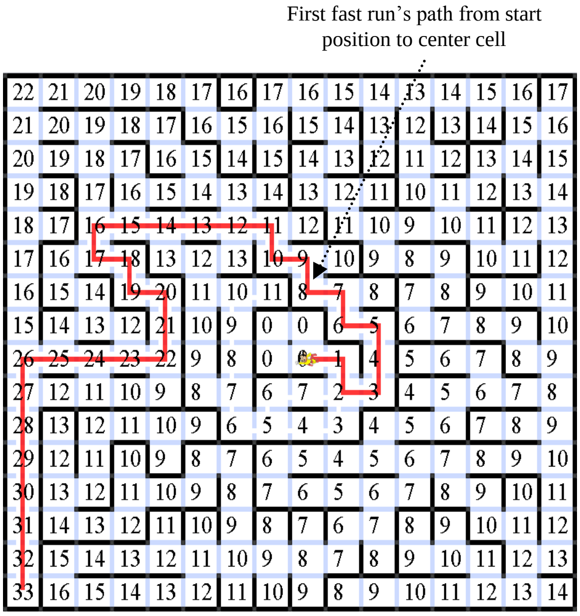

Fig. 3a. Micromouse's first fast run

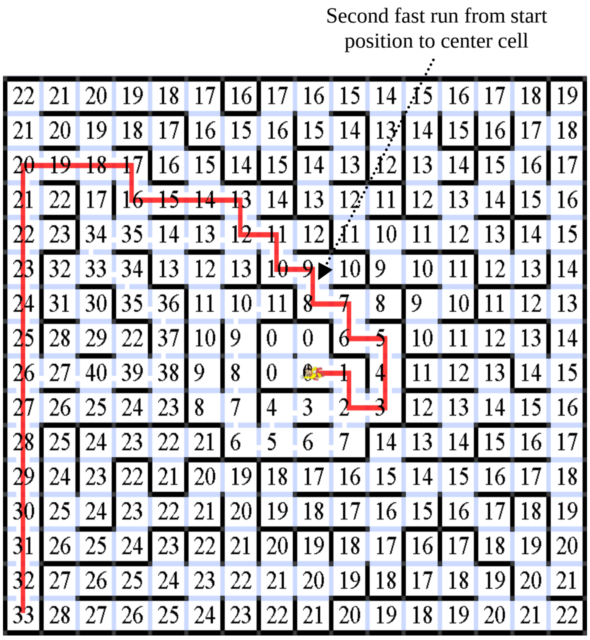

Fig. 3b. Micromouse's second fast run

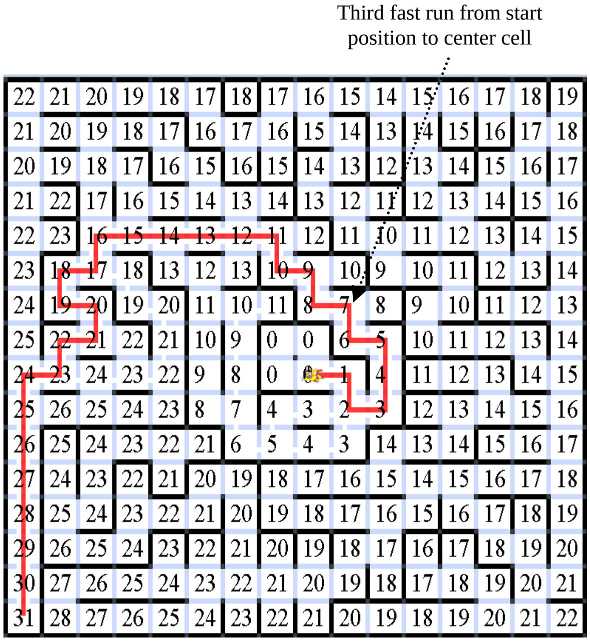

Fig. 3c. Micromouse's third fast run

V. CONCLUSION

Compared to the Flood Fill algorithm, the Modified Flood Fill algorithm offers a significant reduction in cell distance updates (Table I). With a lesser number of distances to update, the micromouse which uses the modified flood fill can traverse from cell to cell in a higher speed. Furthermore, if the dead end cells are marked on the first arrival, these marked cells will not be explored in subsequent runs, resulting in a reduced total number of cells explored. With less demand for computation speed and with the assurance that the best run gives the smallest number of cells traversed, the Modified Flood Fill algorithm may be the choice for micromouse competitions.

APPENDIX

Distance update algorithms comparison

As it has been pointed out earlier, the revised cell distance update algorithm differs from the recursive steps outlined by S. Benkovic in pushing all neighbor cell positions instead of pushing only the open neighbors, when the current cell distance is updated; S. Benkovic's algorithm pushes only open neighbor cell positions. The differences will be illustrated by the following micromouse run on a sample 16×16 cell maze. Fig. 4a, Fig. 4b, and Fig. 4c show only the relevant portion of the maze. Fig 4a shows the distance values when the micromouse reaches cell (4, 1) where the distance update is necessary. Comparing this distance update process will reveal the algorithms' differences.

When the micromouse reaches cell (4, 1), this cell's distance needs update. The revised distance update algorithm will push all neighbors cells (5, 1), (3, 1), (4, 0), and (4, 2) onto the stack whereas S. Benkovic's distance update algorithm will push only open neighbor cells (4, 0) and (4, 2). As a result, cell (3, 1) distance value will not be updated in his algorithm. Fig. 4b shows the revised algorithm's distance values when the micromouse reaches cell (6, 0). Fig. 4c shows S. Benkovic's distance values when the micromouse reaches cell (6, 0). Cell (3, 1) distance values are boldfaced to emphasize the source of discrepancy. This discrepancy may cause the micromouse to take extra steps in reaching the center cell.

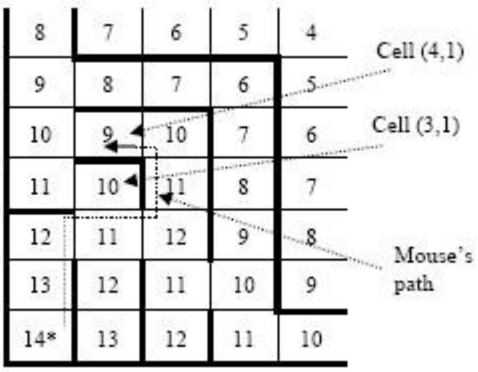

Fig. 4a. Distance values of a portion of a 16×16 cell maze when the micromouse reaches cell (4, 1) before its distance update. * denotes "start position"

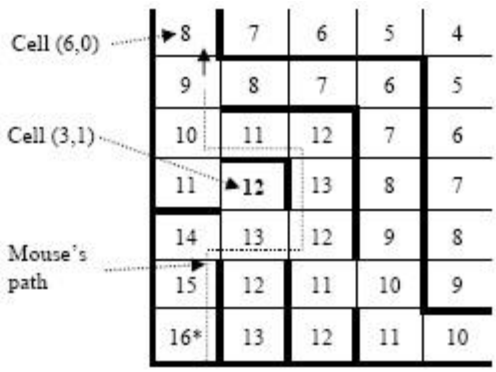

Fig. 4b. Revised distance update algorithm: Distance values of a portion of a 16x16 cell maze when the micromouse reaches cell (6, 0).

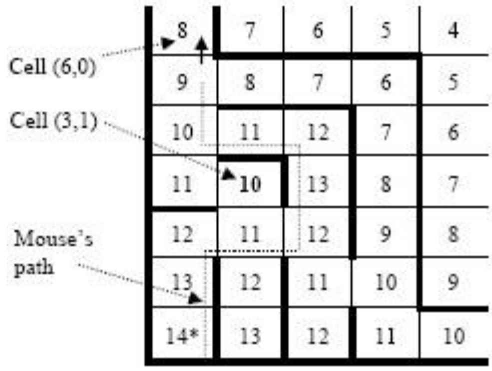

Fig. 4c. S. Benkovic's distance update algorithm: Distance values of a portion of a 16x16 cell maze when the micromouse reaches cell (6, 0)

ACKNOWLEDGMENT

The author thanks Dr. Ali Amini, Chair of Electrical & Computer Engineering, California State University, Northridge, for going out of his way to support the project, and also thanks the ICCSIT reviewers for their constructive suggestions.

REFERENCES

[1] L. Last, N. Veun, V. Frey and J. Smith. (2010). Maze-Solver Simulator. [Online] Available: http://code.google.com/p/maze-solver/downloads/list.

[2] G. Law. (2012). Augmented and revised maze-solver. [Online] Available: https://github.com/glaw-csun/svn/downloads.

[3] M. Sharma and K. Robeonics, "Algorithms for Micro-mouse," in *Proc. International Conf. on Future Computer and Communication*, Kuala Lumpur, pp. 581-585, 2009.

[4] S. Mishra and P. Bande, "Maze solving algorithms for micro mouse," in *Proc. IEEE International Conference on Signal Image Technology and Internet Based Systems*, Bali, 2008, pp. 86-93.

[5] S. Benkovic. The Modified Flood Algorithm. [Online] Available: http://www.micromouseinfo.com/ introduction/mfloodfill.html.

George Law received his B.S.E.E. degree from Georgia Institute of Technology, Atlanta, U.S.A, in 1981, M.S.E.E. degree from Florida Institute of Technology, Melbourne, U.S.A, in 1982, and Ph.D. in electrical engineering from University of Alabama, Tuscaloosa, U.S.A, in 1987. He is an Associate Professor at California State University, Northridge. His current research interests include system-on-chip, real-time operating system, RFID system, and programmable logic.