Faculty of Electrical and Computer Engineering
Laboratory for Control Robotics and Machine Learning

Subject:

Optimal Path Planning for a Robot in an Unknown Environment

Authors:

Omer Huly & Roy Abudi

Supervisor:

Mr. Ori Menashe

Semester:

Winter 2024

Contents

List of Figures

List of Tables

Listings

1 Abstract

The Micromouse competition has been running since the late 1970s around the world. The goal is simple: get to the end of the maze as fast as possible. To solve the problem, a simulation was built to visualize several algorithms' results. The proposed solution combines several algorithms to ensure the robot can find the fastest route. First, a flood-fill algorithm is used for mapping the maze. The robot starts by trying to reach the goal and mapping the cells it encounters. After that, it spends time searching for areas it did not visit, prioritizing large unexplored areas while avoiding dead-ends before returning to where it started. After mapping the maze, Dijkstra's algorithm is used to find the fastest route. Although the flood-fill algorithm, which is very popular among Micromouse competitors, can quickly find the shortest path from the start to the goal, using it alone does not guarantee finding the fastest route. The proposed solution reinforces the flood-fill algorithm and helps solve the problem of finding the fastest route with greater accuracy.

Glossary

ASCII American Standard Code for Information Interchange. A character encoding standard for electronic communication. 9

auxiliary space The extra or temporary memory space that an algorithm uses. 18, 19

BFS Breadth-First Search. A graph traversing algorithm that visits all connected nodes in a level-by-level manner, exploring all the neighbor nodes at the current depth before moving on to the next level. 6

cell A unit of a maze. Each cell can have up to 4 walls surrounding it. 3, 4, 6–29

CSV Comma-Separated Values. A text file which allows data to be saved in a table structured format. 8

DFS Depth-First Search. A graph traversing algorithm that dives as deep as possible along a single branch before backtracking and exploring another branch. 6

graph A collection of vertices or nodes connected in pairs by edges. Often subdivided into directed graphs or undirected graphs according to whether the edges have a direction or not. 3, 16–20, 23, 28

IEEE The Institute of Electrical and Electronics Engineers. An American professional association for electronics engineering, electrical engineering, and other related disciplines. 6, 26

maze A rectangular area made out of cells with a starting position and a goal. 3, 4, 6–16, 18–29, 31–34

Micromouse A competition where small robots try to solve a maze without prior knowledge in the shortest time. 2, 4, 6, 7, 14, 15, 19, 20, 26–29, 31–34

nibble 4 bits, half a byte. 7, 29

2 Introduction

Today, robots are used more and more. Often, robots should move or walk and find their path autonomously. Therefore, the ability of robots to find their way around obstacles is needed. This project is based on the Micromouse competition as a basis for developing optimal path-planning algorithms in an unknown environment. Each competition complies with the IEEE Micromouse standard rules. The Micromouse competition features small autonomous robots called "micromouse". The task is to solve the maze. To complete this task, the robot needs to move from the start point to the final destination point in the center of the maze. The robot has no prior knowledge of the maze. The competition requires overcoming the engineering challenge of designing and programming a robot capable of solving the maze as fast as possible. Our project focuses on the algorithmic aspect of the problem and does not include a physical robot. A simulation has been built to emulate the physical maze environment and test our algorithms.

The competition is not new and there have been numerous attempts at solving the problem and refining the algorithms. The naïve wall-following algorithms were successful in the early days of the competition. Later, the goal of the maze was moved away from the edges. This change could lead to a robot using a naïve wall-following algorithm getting stuck in an infinite loop circling the maze [6]. Later, more sophisticated algorithms were tested such as BFS and DFS for traversing the maze but the former can be very slow because of backtracking and the latter is not guaranteed to find the fastest path. The solving algorithm continued to develop over the years and today the most common algorithm used in Micromouse is the flood-fill algorithm [6]. We will delve deeper into this algorithm and more later in this work.

In this paper, first, there will be an explanation of how the problem has been modeled. After that, there will be an overview of the maze simulation and its features. Next, there will be a comprehensive review of the various algorithms that have been tested. In section 5, naïve algorithms such as random path and wall follower are discussed. In section 6, more sophisticated algorithms such as the flood fill algorithm and Dijkstra's algorithm are discussed. Later in section 7, an explanation of the main product, our new proposed algorithm is presented. Finally, the results and conclusions are given in section 8.

3 Modeling the Problem

3.1 Maze Structure and Rules

The maze in the Micromouse competition consists of a 16×16 grid of unit squares. Each one of these squares is called a cell. External walls enclose the entire maze. The starting cell of the maze is located at one of the four corners and is always bounded by walls on three of its four sides. The goal cells are a 2×2 square at the center of the maze bounded by walls except for one opening. The robot's objective is to achieve the shortest runtime. A run begins every time the robot leaves the starting cell. When the robot reaches the goal, a runtime is recorded. If the robot re-enters the starting cell, the run ends, and a new runtime will begin when the robot leaves the starting cell. More about the competition rules can be found in section 9.3.5.

Each cell's wall is represented by its absolute position around the edges of the cell. Therefore, the wall on the upper edge is referred to as the north wall. The wall on the left edge is referred to as the east wall. The wall on the lower edge is referred to as the south wall. Lastly, the wall on the right edge is referred to as the west wall. Figure 1 presents a cell and its walls for clarity.

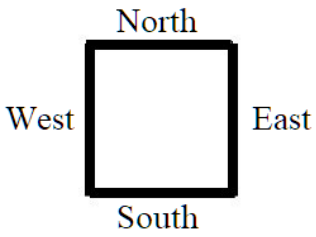


Figure 1: The name of each wall surrounding a cell

A maze is considered valid if it is:

1. Non-empty
2. Rectangular
3. Enclosed - all outer walls of the cells on the perimeter of the maze are present.
4. Consistent - adjacent cells agree on their shared wall.

3.2 Maze Representation

3.2.1 Prologue

There are many possible ways to represent a maze. This section gives an overview of the different maze representations we found that are used in previous works about the Micromouse challenge [1, 4, 5, 7]. The various maze representations are named after the most common file suffix for maze files using them. Later, in section 3.2.6, the chosen representation, which is used for the simulation and mapping of the maze by the robot's algorithm, will be explained.

3.2.2 "maz" Format Representation

Each cell is represented by its four walls using a single bit per wall where '1' represents an existing wall and '0' represents a missing wall. The memory layout of a cell is described in the following table.

bit	7	6	5	4	3	2	1	0
value	0	0	0	0	W	S	E	N

Table 1: "maz" format cell memory layout

The lower nibble is where the wall's data resides. The letters N, E, S, and W represent the North, East, South, and West walls respectively. The upper nibble is ignored. Thus, the whole maze is represented by a two-dimensional array of cells where the index of each cell is the coordinate of that cell in the maze. The top left cell in the maze has a coordinate of $(0,0)$. An example of this format and the resulting maze:

File (hex):

```
09 03 0b 0e 0c 06
```

Result:

```
+---+---+---+
|       |   |
+   +   +   +
|   |       |
+---+---+---+
```

3.2.3 ”CSV” Format Representation

A text format based on the CSV file format. Each row contains integers separated by commas. The coordinate of the cell in the maze is determined by its position in the text. The first integer in the first row is the top left corner of the maze. The next integer in the first row is the cell to the right of the first one. The first integer in the second row is the cell below the first integer in the first row. And the pattern continues. The integers are in the range $[1, 16]$. Each integer represents a different wall combination around the cell as can be seen in the following figure.

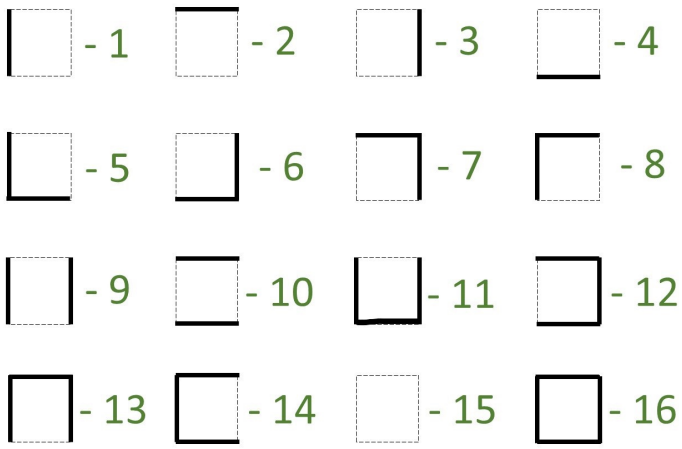


Figure 2: States for cell’s walls in ”CSV” format

The maze starts empty, then for every line:

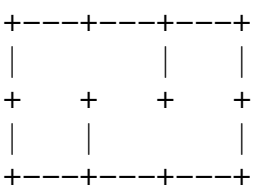
1. The next integer is read and parsed.
2. A cell is created in the maze at the corresponding coordinate.
3. The new cell is added with walls surrounding it based on the values of the integer.

After all lines have been parsed, the maze is checked for validity. An example of this format and the resulting maze:

File:

```
8, 7, 13
11, 5, 6
```

Result:



3.2.4 ”num” Format Representation

A simple text format where each line is made up of 2 integer values, X Y, followed by 4 boolean values, N E S W, representing:

- X, Y - the cell's X and Y coordinates. X is the horizontal axis (columns), Y is the vertical axis (rows), and $(0,0)$ is the bottom left corner.
- N, E, S, W - North, East, South, West - represent where the cell has walls.

The maze starts empty, then for every line:

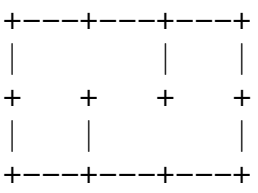
1. The numbers are read and parsed.
2. A cell is created in the maze at the (X,Y) coordinate.
3. The new cell is added with walls surrounding it based on the values of N, E, S, and W.

After all lines have been parsed, the maze is checked for validity. An example of this format and the resulting maze:

File:

```
0 0 0 1 1 1
0 1 1 0 0 1
1 0 0 0 1 1
1 1 1 1 0 0
2 0 0 1 1 0
2 1 1 1 0 1
```

Result:



3.2.5 ”maze” Format Representation

A simple ASCII file containing the maze itself. cells are 5 characters wide and 3 characters long and have a shared wall with each adjacent cell. The maze starts empty. The height and width of the maze are calculated. Then, for each cell a wall is set if there wasn't a space in its location. After that, the maze is checked for validity. An example of this format:

File:

```
+---+---+---+
|           |   |
+   +   +   +
|   |           |
+---+---+---+
```

3.2.6 Chosen Maze Representation

The ”maz” representation defined in section 3.2.2 is used for the robot's in-memory maze due to its simplicity and compatibility with basic bit-wise operations. However, for ease of use, the ”maze” representation defined in section 3.2.5 is used as the default for maze input/output due to its visual nature. However, our project supports input in all formats discussed above, which are then converted into the ”maz” format representation. We chose to support all formats to be able to use maze input from all the sources we found [1, 4, 5, 7].

3.3 Robot Representation

The robot is represented by a circle with a colored notch that illustrates the direction in which the robot is facing. The robot occupies a single cell in the maze. The robot can perform six types of actions:

1. Ready to start
2. Reset the algorithm and get back to the starting cell
3. Move forward one cell
4. Move backwards one cell
5. Rotate by 90° to the left
6. Rotate by 90° to the right

4 Simulation

4.1 Overview

The maze simulation is designed to simulate and examine the algorithms while providing visual information about the execution process. In addition, the simulation provides an interface between the robot and the maze. The simulation includes algorithms to choose from to solve the maze. Some of those algorithms are:

1. Random - explained in section 5.1
2. Left/Right wall-follower - explained in section 5.2
3. Flood-fill - explained in section 6.2

4. Flood-fill + Dijkstra - explained in section 7.2

5. Thorough flood-fill - explained in section 7.3

In addition, an ”idle” robot is provided for a robot that does nothing. Of course, this robot can’t solve a maze.

4.2 Robot-Maze Interaction

The simulation works in ”steps”. At each step, the simulation provides the robot with information about the walls of the cell it is currently in. Given this information, the robot decides what action to take and informs the simulation about it. Next, the simulation ensures that the robot performed a legal action. For example, if the robot is attempting to move forward one cell while there is a wall right in front of it, the simulation will give an error and halt the execution of the algorithm. If all checks are passed, the simulation updates the robot’s position or orientation based on the action. And the cycle repeats itself. When the algorithm is finished, meaning the robot does not take any action, the simulation stops the execution.

4.3 Graphical User Interface

When the simulation starts, the simulation’s screen as seen in figure 3 is displayed. It includes the two main components in the simulation’s screen:

- A full maze view that shows the robot, goal, and the fastest route calculated using Dijkstra’s algorithm. Using Dijkstra’s algorithm to find the fastest route in a maze is expanded upon in section 6.1.

- The robot’s view - the maze as seen by the robot. At the start of the simulation, the only known information is the starting cell’s walls and the goal cells’ location.

Figure 3: Simulation screen

The simulation's controls in figure 4 include various components:

- Drop-down menu to choose which algorithm to use to try and solve the maze. Initially set to *idle*.
- A start/stop button to make the robot start executing the algorithm or stop it mid-run and then start again from the same spot. Initially set to *stop* as the simulation starts executing the algorithm as soon as it is chosen.
- A step button to make the robot perform a single move operation.
- A simulation delay slider to control how much time is added to each step.
- Information about the maze such as the coordinates of the starting and goal cells and the direction the robot is facing at the start of the simulation. This information is loaded with the maze and cannot be modified directly.
- Information about the robot's path is displayed by three counters. The *step* counter shows the number of actions the robot has done in the current run and the total amount. The *weight* counter shows the weight of the path from the starting cell of the current run and the total weight. The *cell* counter shows the number of cells the robot has gone through in the current run and the total amount. Note that this counter does not count unique cells.

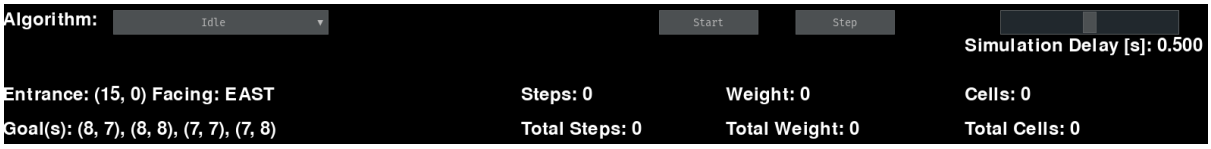


Figure 4: Simulation controls close-up

Moreover, the simulation displays two more statistics at the bottom of the screen: the number of explored cells and their percentage.

Figure 5: Simulation bottom close-up

4.3.1 Heat Map

As the simulation executes the robot's algorithm a heat map is generated on the full maze view. Each cell is colored from cool to warm to indicate how many actions the robot has performed in each cell. All cells start without a color. After every action, the color of the cell at the robot's position gets warmer. The color of the cell can be one of the following: blue, cyan, green, yellow, orange, red, and brown. Blue means the robot has performed one action in that cell while brown means the robot has performed at least seven actions in that cell. A 90° turn is composed of 2 actions: rotating the robot and moving forward. As such, a 90° turn counts as 2 steps, and a 180° counts as 3 steps. An example of a heat map can be seen in figure 6 where a right wall follower robot is attempting to solve the maze.

Figure 6: Heat map example

The heat map scale index is provided in figure 7.

Figure 7: Heat map scale index

4.3.2 Additional Cell Information

The simulation provides an easy way for an algorithm to save additional information about the maze. In addition, this information will be displayed by the simulation on the maze. The additional information that can be added to each cell is:

1. Weight - A numeric value the algorithm can assign to the cell, meant to be used by algorithms that use weights.

2. Color - The color of the cell. Can be used to add visualization to the algorithm.

3. Visited - A counter that can be used to keep track of how many times the robot has visited a cell. This value is automatically incremented by the simulation after every robot action.

4.3.3 Special Flood-Fill Information

The flood-fill algorithm, which is explained in section 6.2, stores the distances of each cell from the goal. When simulating an algorithm that uses the flood-fill algorithm to explore the maze, the currently stored distances of each cell from the goal are displayed in the robot's view. In addition, the robot's path is traced as it moves across the maze. An example of the additional

information provided for flood-fill-based algorithms can be seen in figure 8 where a flood-fill robot is attempting to solve the maze.

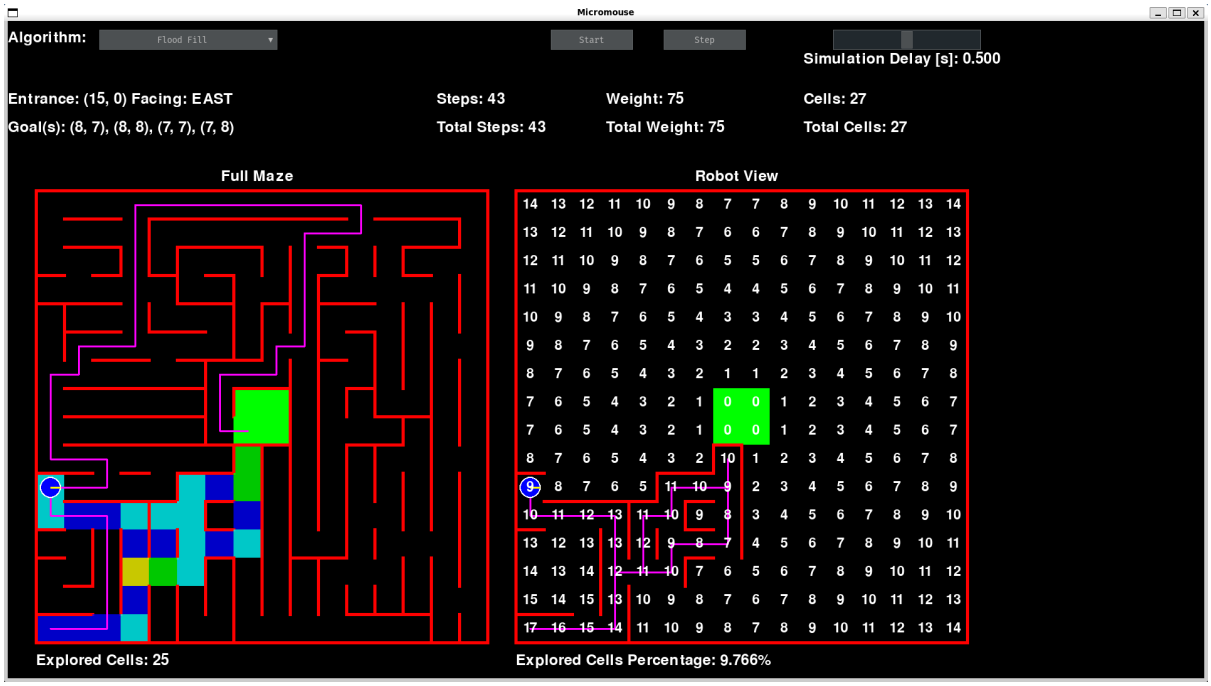


Figure 8: Extra information for flood-fill-based algorithms

4.3.4 Special Dijkstra's Algorithm Information

The Dijkstra's algorithm, which is explained in section 6.1, calculates the minimum distances of each cell from the starting cell. When simulating an algorithm that uses Dijkstra's algorithm to solve the maze, the calculated distances of each cell from the starting cell are displayed in the robot's view. In addition, the robot's path is traced as it moves across the maze. An example of the additional information provided for Dijkstra-based algorithms can be seen in figure 12 where each cell contains the distance from the starting cell calculated by Dijkstra's algorithm with the defined time weights for each action. In addition, a cell that was not explored or is not reachable, is colored accordingly.

4.3.5 Other Algorithm's Information

Specific algorithms can color the maze's cell to display information about the execution of the algorithm. For example, our new proposed algorithm, Thorough Flood-Fill explained in section 7.3, uses different colors for cells based on its current state:

- Cells that are unexplored, can be seen in figure 13.
- The cell that is the current exploration goal, can be seen in figure 13.
- Cells that are considered as a dead-end, can be seen in figures 14, 15, 16, and 17.

4.4 Generic Algorithm Test Environment

The code is structured to ease the process of adding and testing new algorithms. The intent is to provide a platform for testing Micromouse algorithms easily. Moreover, the GUI is replace-

able if you want to render the simulation differently or present different information about the algorithm. The technical details of adding new algorithms or changing the GUI are explained in detail in the code repository. A link to the code repository can be found in section 9.1.

5 Naïve Algorithms

The algorithms in this section have been implemented for the initial testing of the simulation environment. The main usage of those algorithms is to check the integration between the different components, mainly, the maze representation and the simulation environment. They are not intended to solve the Micromouse challenge. In addition, for simpler mazes, which those algorithms can solve in a reasonable time, they can be used as a starting point for improvement.

5.1 Random Path Algorithm

In this approach, the robot sees which walls surround it for each step and randomly decides where to go. This simple logic can be demonstrated by the following algorithm:

1. Randomly choose a cell which is one step away and has no wall blocking it and move to it.
2. If current cell is not the goal, go back to step 1, else finish.

This approach does not provide a robust solution to the problem. Most of the time, the robot will not find the goal in a reasonable time and therefore, will not be able to successfully solve the Micromouse challenge.

5.2 Wall-Following Algorithm

The wall-following algorithm is divided into two:

1. Left wall-following algorithm
2. Right wall-following algorithm

Both algorithms are almost identical with the only difference being which wall to follow. For instance, the logic for a right wall follower robot can be demonstrated by the following algorithm:

1. If current cell has no wall to the right of the robot then turn right.
2. If current cell has no wall in front of the robot then move forward, else turn left.
3. If current cell is the goal, stop, else return to step 1.

The algorithm for a left wall follower robot is achieved by replacing all instances of the word 'right' with the word 'left' and all instances of the word 'left' with the word 'right' in the algorithm presented above. As discussed in section 2, the wall-following algorithm is not suitable for solving the Micromouse challenge. A robot utilizing this algorithm will most likely never reach the goal because the maze is deliberately constructed in a way that prevents using wall-following techniques.

6 Sophisticated Algorithms

6.1 Dijkstra's Algorithm

6.1.1 Description

An algorithm for finding the shortest paths between nodes in a directed weighted graph. The algorithm can be described in pseudo-code as follows:

```
Dijkstra(graph, source):
    distance ← empty array
    previous ← empty array
    Q ← empty queue
    for each vertex v in graph:
        distance[v] ← ∞
        previous[v] ← NULL
        add v to Q
    distance[source] ← 0
    while Q is not empty:
        u ← vertex in Q with minimum distance
        remove u from Q
        for each edge e in graph connecting u and v:
            new_distance ← distance[u] + weight of e
            if new_distance < distance[v]:
                distance[v] ← new_distance
                previous[v] ← u
    return distance, previous
```

Listing 1: Dijkstra's algorithm pseudo-code

The *distance* array is the current distance from the source to each vertex. The *previous* array contains pointers to the previous-hop nodes on the shortest path from the source to the given vertex. Given a directed weighted graph and a source node, we can calculate the distance and the shortest route to every other node in the graph from the source node.

Therefore, representing the maze as a directed weighted graph with the start cell as the source node and applying Dijkstra's algorithm, will allow finding the shortest or fastest route in the maze.

6.1.2 Maze as a Directed Weighted Graph

To use Dijkstra's Algorithm, first, the maze must be represented as a directed weighted graph. Each cell in the maze is represented by up to four vertices in the graph. Those vertices correspond to the direction the robot is facing while located at that cell - North, East, South, and West. Each vertex appears if the corresponding wall does not exist. Because the robot rotation is limited to 90° left or right, there is an edge connecting the north vertex to the east and west vertices. Because the robot can move forward, there is an edge connecting the north and south vertices. The same applies to the east, south, and west vertices. Therefore, it can be viewed as a graph connecting the missing walls of the cells. The representation of a single cell as a directed graph is demonstrated in the following figure.

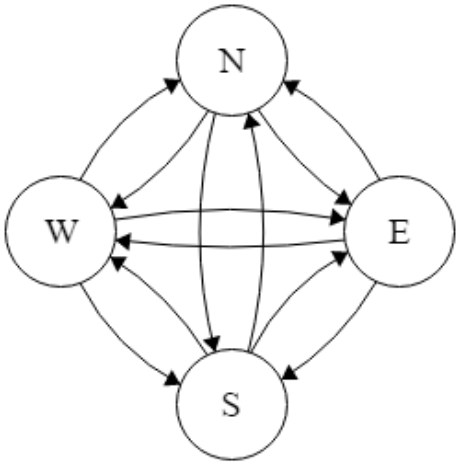


Figure 9: A cell as a directed graph

The weight of each edge can be changed to achieve different definitions of the fastest route. For example, setting the weight of all edges to 1 will result in finding the shortest path with the least amount of moving forward and turning.

Since we care about the fastest route rather than the shortest one, our implementation uses time units for weights. An arbitrary unit of 1 was decided for moving the distance of a single cell forward. Acceleration and deceleration were both also given a weight of 1 and a 90° turn was given a base weight of 2. A turn is composed of a decelerate-turn-accelerate sequence and therefore gets a total weight of $1 + 2 + 1 = 4$ in the graph to accommodate the necessary velocity changes.

To allow movement between the different cells, two zero-weight edges connect each shared wall vertices. An example of 4 adjacent cells with no walls as a directed weighted graph is shown in figure 10. The coordinates of each cell are written alongside the direction of the wall.

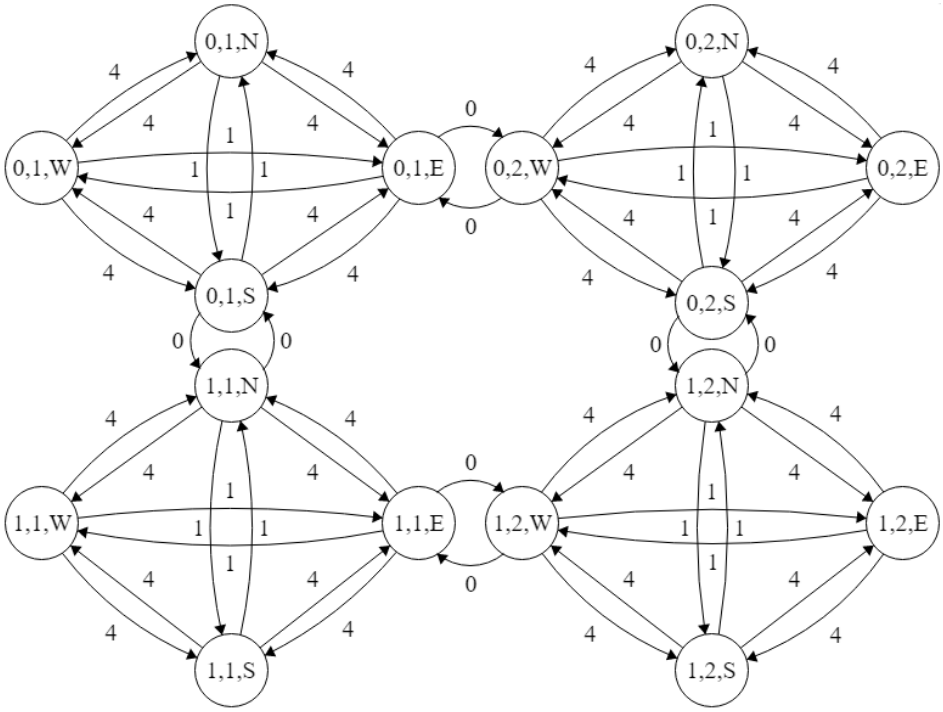


Figure 10: Four cells with no walls as a directed weighted graph

Next, the representation of the following sample maze as a directed weighted graph is shown in figure 11.

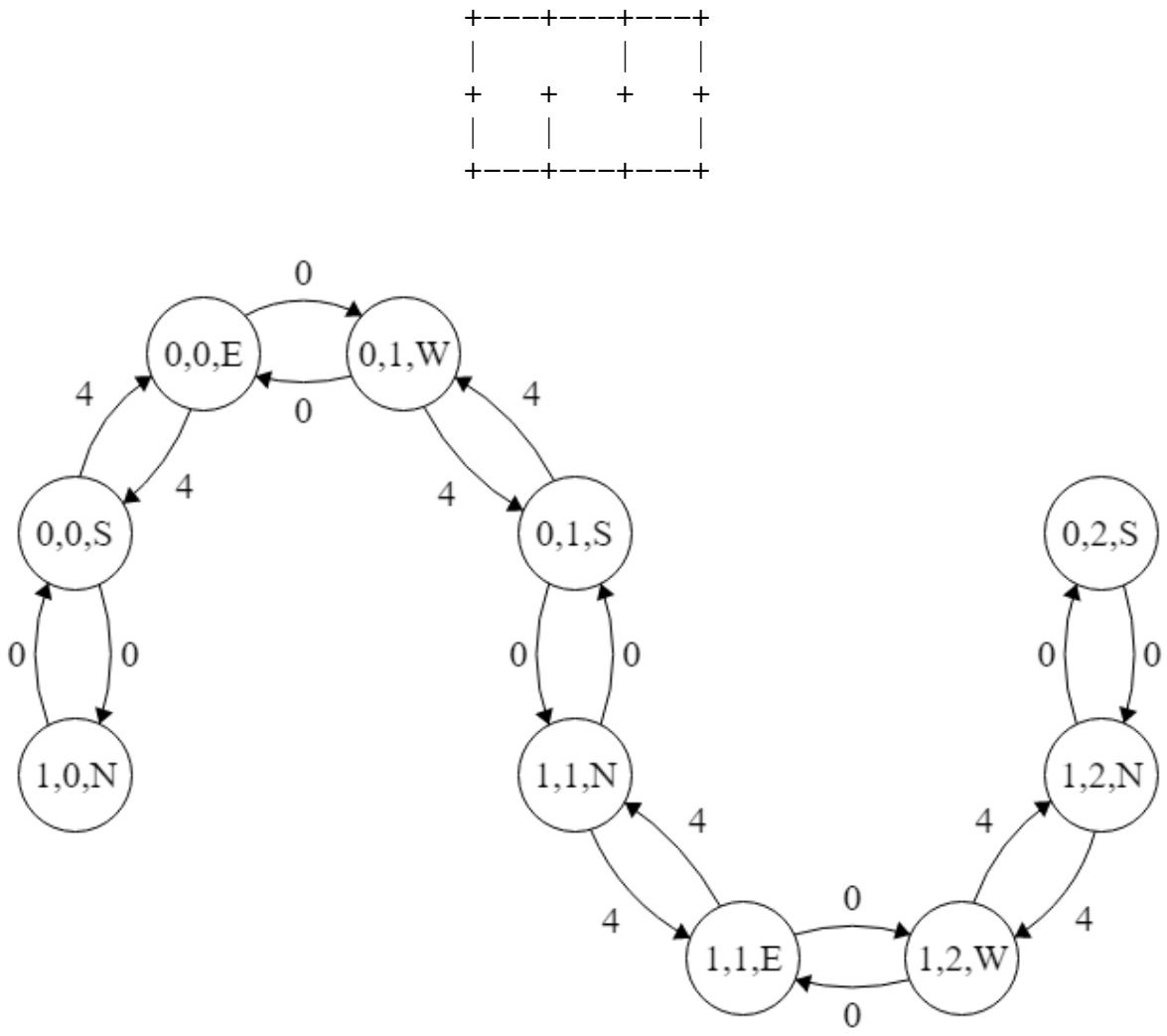


Figure 11: Sample maze as a directed weighted graph

The coordinates of each cell are written alongside the direction of the wall. Note that each cell in figure 11 only has nodes where the cell has no walls. This can save a lot of memory and computation time for algorithms that use the resulting graph.

6.1.3 Complexity

The time complexity for Dijkstra's algorithm is given by $O(|V| + |E| \log |V|)$ where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph. The auxiliary space complexity of the algorithm is $O(|V|)$ where $|V|$ is the number of vertices in the graph. The number of vertices in the graph, $|V|$, is proportional to the number of cells in the maze. Likewise, the number of edges in the graph, $|E|$, is proportional to the number of vertices in the graph. Therefore, the time complexity for this algorithm can be written as $O(n \log n)$ where n is the number of cells in the maze. In addition, the auxiliary space complexity of the algorithm can be written as $O(n)$ where n is the number of cells in the maze.

6.1.4 Drawbacks

As discussed in section 2, the robot has no prior knowledge about the maze. This limitation and the fact that Dijkstra's algorithm is not a graph traversing algorithm, mean the maze must be explored by other means before using Dijkstra's algorithm.

6.2 Flood-Fill Algorithm

6.2.1 Description

Unlike Dijkstra's algorithm, discussed in section 6.1, this algorithm assumes no prior knowledge about the maze. This property and its simplicity make it an extremely popular algorithm used in the Micromouse competition. The robot takes the most optimistic path to the goal. In each step the robot follows the next logic:

1. Update the maze to include the current cell's walls.
2. Update the distances from every cell in the maze to the goal based on the currently known walls in the maze.
3. Move to the cell with the lowest distances from the goal among the cells adjacent to the current cell. If more than one applies choose one at random.

The process resembles flooding the maze with water and updating values based on the flow. This resemblance gives the algorithm its name. Once the robot reaches the goal it can optimize the path it took by looking back and choosing the path following the trail of decreasing distances from the goal.

6.2.2 Complexity

The time complexity for this algorithm is given by $O(n)$ where n is the number of cells in the maze. The auxiliary space complexity of the algorithm is $O(n)$ where n is the number of cells in the maze. As can be seen, the complexity of this algorithm is better than the complexity of Dijkstra's algorithm, described in section 6.1.3, giving it a shorter execution time.

6.2.3 Finding the Shortest Path

In a Micromouse competition the robot needs to get back to the starting cell on its own to start the next run or a time penalty is issued, as noted in section 9.3.5. The flood-fill algorithm takes advantage of the return trip to further optimize the path. Once the robot reaches the goal, the return trip is a backward flood-fill, meaning the goal is the new starting cell and the starting cell is the new goal. Adding a bias to the flood-fill algorithm which tries to avoid cells already visited by the robot on the way to the goal, makes the return trip different. The bias is added by giving more weight to cells the robot has already visited.

For example, the robot can store one bit for each cell in the maze. This bit will be denoted as the *visited bit*. The *visited bit* will store whether the robot has already visited the cell. On the way back from the goal, the weight of each cell will be the distance from the starting cell plus the value of the *visited bit*. This will increase the weight of cells already visited by 1 and will make the robot less likely to travel through those cells on the way back.

Once the robot reaches the start it can optimize the path it took by looking back and choosing the path following the trail of decreasing distances. Between these two attempts, one of

them is likely to be the shortest path in the sense of the least amount of cells required to reach the goal.

6.2.4 Drawbacks

The flood-fill algorithm is simple and fast and does not assume any prior knowledge about the maze. This makes it a very fitting algorithm for solving the Micromouse challenge. However, the result of this algorithm is the shortest path from the starting cell to the goal, which is not guaranteed to be the fastest route the robot can take. The winner of the Micromouse competition is the robot that reached the goal in the least amount of time, as noted in section 9.3.6. The shortest path to the goal could have a lot of turns which will slow down the robot, while another longer path utilizes more straight lines the robot can race through much faster. Therefore, finding the shortest path is not always the best solution to the problem. A way to add weights that represent the time it takes for each action to complete is needed. Unfortunately, the flood-fill algorithm alone is not suitable for this kind of task.

7 The New Proposed Algorithm - Thorough Flood-Fill

7.1 Prologue

Up to this point, a way to solve a known maze while finding the fastest path was discussed in section 6.1. Moreover, a way to solve an unknown maze while also finding the shortest path was discussed in section 6.2. The new proposed algorithm utilizes the flood-fill algorithm to explore the maze and Dijkstra's algorithm to find the fastest path by the given time weights.

7.2 Combining the Flood-Fill Algorithm with Dijkstra's Algorithm

7.2.1 Description

At the start, the robot has no knowledge about the walls of the maze except for the walls of the starting cell. The flood-fill algorithm is used to get from the starting cell to the goal. Once the robot has reached the goal, it uses the flood-fill algorithm to navigate back from the goal to the starting cell in a different path, as described in section 6.2.3. Now, the robot has acquired some knowledge about the maze and found at least two different routes leading from the starting cell to the goal. For the solution, instead of following the shortest path between the two attempts, the robot finds the fastest one according to Dijkstra's algorithm is chosen, even if it is longer.

Even though not all the maze has been explored, the algorithm presented in section 6.1.2 is still viable. To use that algorithm, unexplored cells are treated as having four walls. Therefore, unexplored cells are excluded from the graph and ignored by Dijkstra's algorithm. As seen in figure 12, unexplored cells are assigned a weight of inf to indicate that the robot can't reach them.

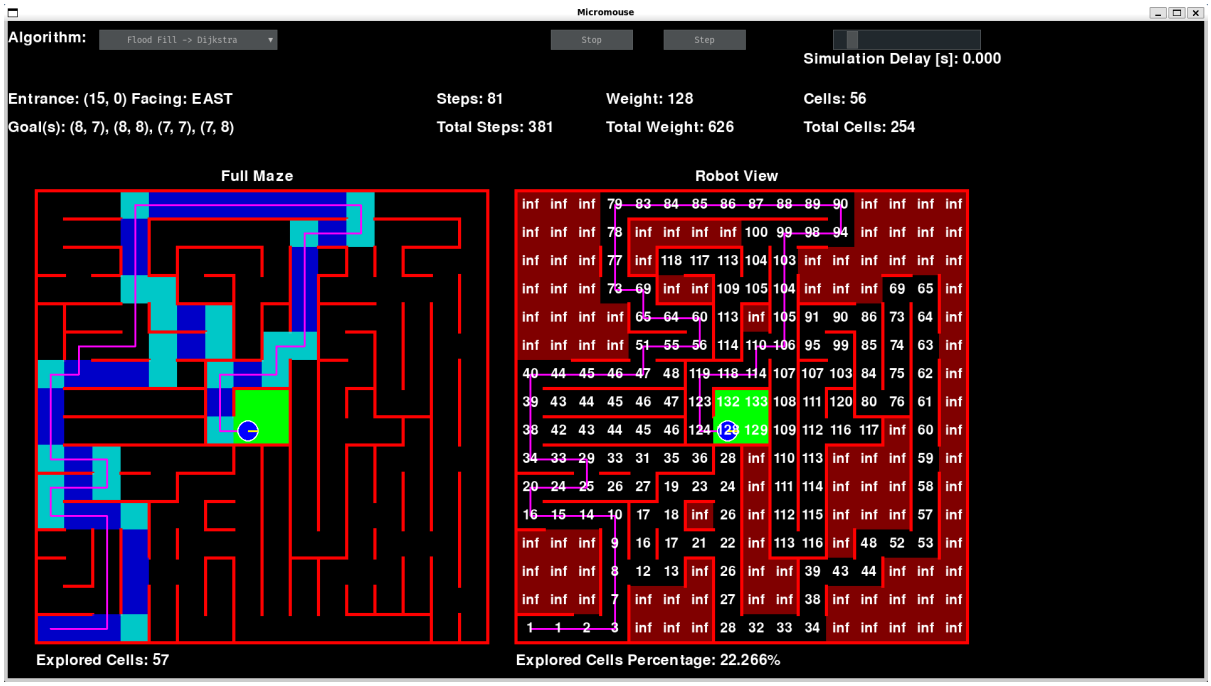


Figure 12: Simulating flood-fill combined with Dijkstra's algorithm

By combining the two algorithms the robot can explore an unknown maze and find the fastest route. This approach allows leveraging each algorithm's advantage while alleviating its disadvantages.

7.2.2 Drawbacks

As discussed in section 6.2, the flood-fill algorithm does not explore all the maze. Therefore, although it gives a good solution to the maze, which can be the optimal solution according to the defined time weights of each action, it is not guaranteed to be optimal.

7.3 Thorough Flood-Fill Algorithm

7.3.1 Description

The *Thorough Flood-Fill* algorithm is our new proposed solution. The name is based on the intensive exploration of the maze using the flood-fill algorithm. We want to use the algorithm discussed in section 7.2 while guaranteeing it can find the optimal solution. To achieve this, we should explore more regions of the maze. The downside of exploring more areas of the maze is that it takes time off of the allocated time of access to the maze, as noted in section 9.3.5. Therefore, the algorithm will prioritize finding the goal and then use the remaining time to explore the maze as much as possible before returning to the starting cell and finding the fastest route.

7.3.2 Mapping Unexplored Regions

After finding a path from the starting cell to the goal, the robot explores the maze as much as possible. To achieve that, once the robot reaches the goal, it performs the following exploration logic:

1. Sort all unexplored cells in the maze into groups of connected unexplored cells. The robot has no knowledge about the walls surrounding those cell yet. Therefore, unexplored cells are in the same group if there could be a path between them that does not cross an already explored cell.

2. Perform a flood-fill algorithm from the current cell to the cell that is furthest away from the current cell and is part of the largest group of unexplored cells. This cell is the next goal.

3. When reaching the next goal, mark it as the starting cell and return to step 1.

The simulation supports visual aids for the exploration part of the algorithm. As the robot reaches the goal, it turns to explore other areas of the maze. An example of mapping the maze can be found in figure 13. Unexplored cells are marked with a different color on the robot's view of the maze (right). The robot has marked the $(0,0)$ cell as its current destination and is heading there.

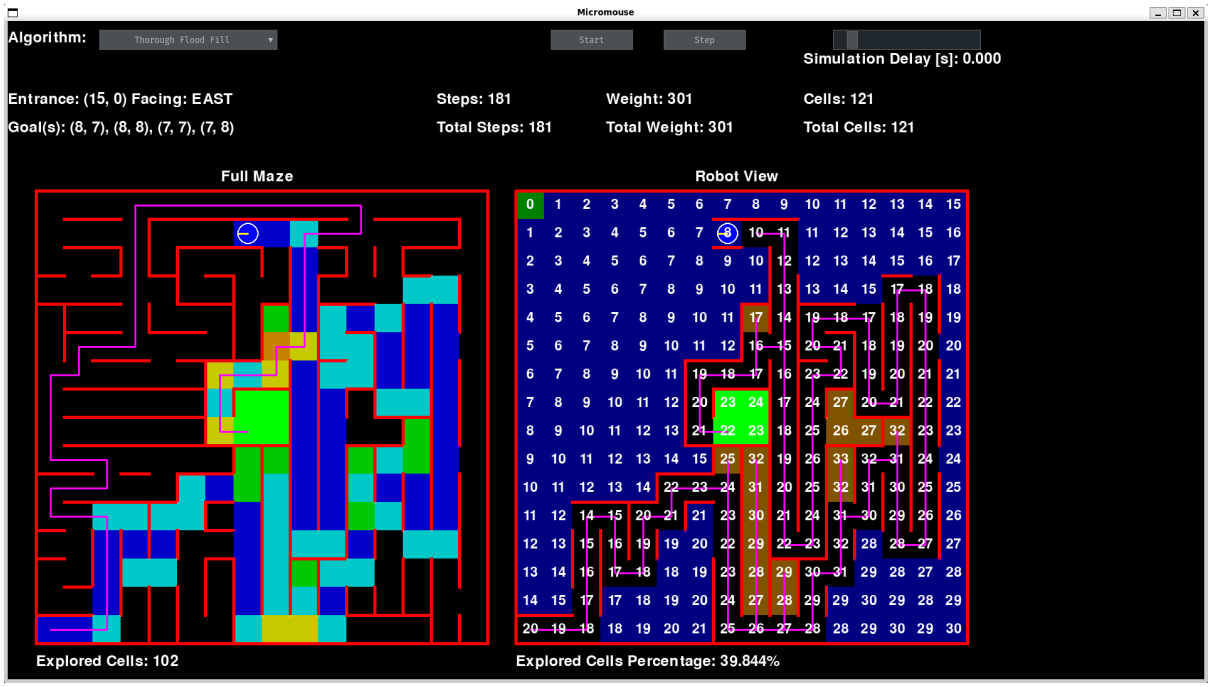


Figure 13: Robot exploring the maze using the thorough flood-fill algorithm

The exploration logic continues until the robot has explored enough cells in the maze. While exploring the maze, the robot needs to avoid the starting cell because this will start a new run, as noted in section 9.3.5. Stepping in the starting cell can end the exploration run prematurely. An infinite weight is assigned to the starting cell so that no route will go through it to avoid this problem. When the explored cells percentage in the maze is higher than the threshold, the robot stops exploring the maze. Next, the robot sets the starting cell as its next goal and uses Dijkstra's algorithm to find the fastest route from its current cell to the starting cell. The fast run will start once the robot reaches the starting cell. The robot uses Dijkstra's algorithm to find the fastest route from the starting cell to the goal.

7.3.3 Dead-End Detection

Rationale Given a directed weighted graph for a maze, Dijkstra's algorithm can find the fastest path. As explained in section 6.1.4, Dijkstra's algorithm needs a large amount of information about the maze to find the optimal path. Therefore, we want the robot to explore the minimum number of cells that still allows Dijkstra's algorithm to find the optimal path.

Adding Imaginary Walls To reduce the exploration time, once the robot has enough information about a group of cells, it can deduce that the group is considered a dead-end and not explore that group further. While the robot explores the maze, it tries to add walls around itself. If the wall's addition does not break the maze's connectivity (the goal cells, the starting cell, and the robot's current cell are all still connected) and creates a cell group that is separate from the rest of the maze, this cell group is considered a dead-end. The group is then colored with a distinct color to stand out.

Figure 14 demonstrates cells marked as a dead-end. Notice the three cell group to the robot's right, which is not marked as a dead-end. The region of interest is marked on the image with a circle. Those cells are not leading anywhere. By adding an imaginary eastern wall in the current cell, this group is disconnected from the maze and does not contain the starting cell, the goal, or the robot.

Figure 14: Thorough flood-fill before coloring dead-end to the right

Figure 15 shows the state after the robot tries to put an imaginary wall to its right and checks if any group of cells is disconnected from the graph. Notice that the three cells mentioned above are now colored differently. Those cells are marked as a dead-end, and the robot won't enter them in the future while exploring the maze or solving it.

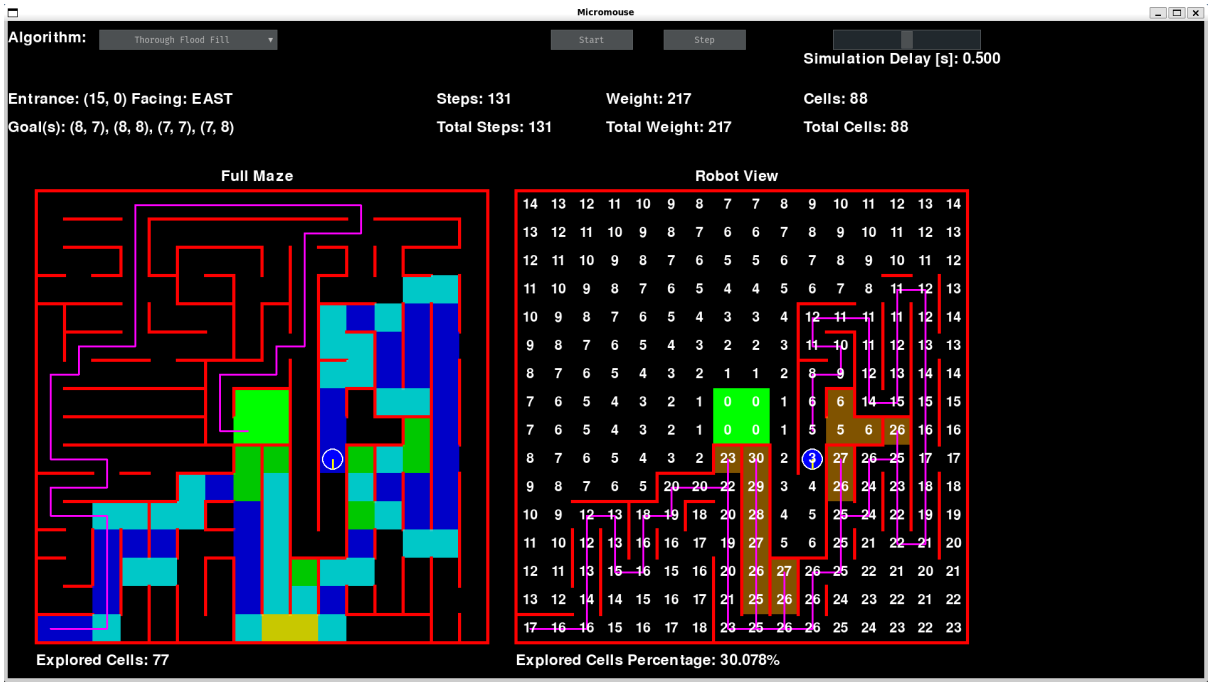


Figure 15: Thorough flood-fill after coloring dead-end to the right

This technique allows the robot to save time and still get all the information needed to find the fastest route to solve the maze.

Flood-Fill as a Dead-End Detector The method mentioned above can detect dead-ends. A dead-end is an area of the maze with only one way in or out. But, we also want to detect areas of the maze that are not dead-ends but are still not worth going through. We will refer to those areas as dead-ends even though they might have more than one way in or out.

The flood-fill dead-end detection algorithm works by looking at the flood-fill distances of each cell in the maze from the goal and tries to find groups of cells that satisfy the following criteria:

- Every cell in the group must have a distance higher than the cells from which the robot can access the group.
- The group does not contain the starting cell or the goal cells.

Such groups are guaranteed to not be part of the shortest path according to the flood-fill algorithm.

Every time the robot gains new information about the maze, the flood-fill distances are re-calculated, and if a group of cells satisfies the conditions above, it is considered a dead-end.

For example, in figure 16, the robot is making its way to a cell while exploring the maze. Notice the group of cells the robot is facing. The region of interest is marked on the image with a circle. Those cells are making a detour around the main path. Therefore, they are not worth checking. In this position, the robot knows about all the walls surrounding this group of cells.

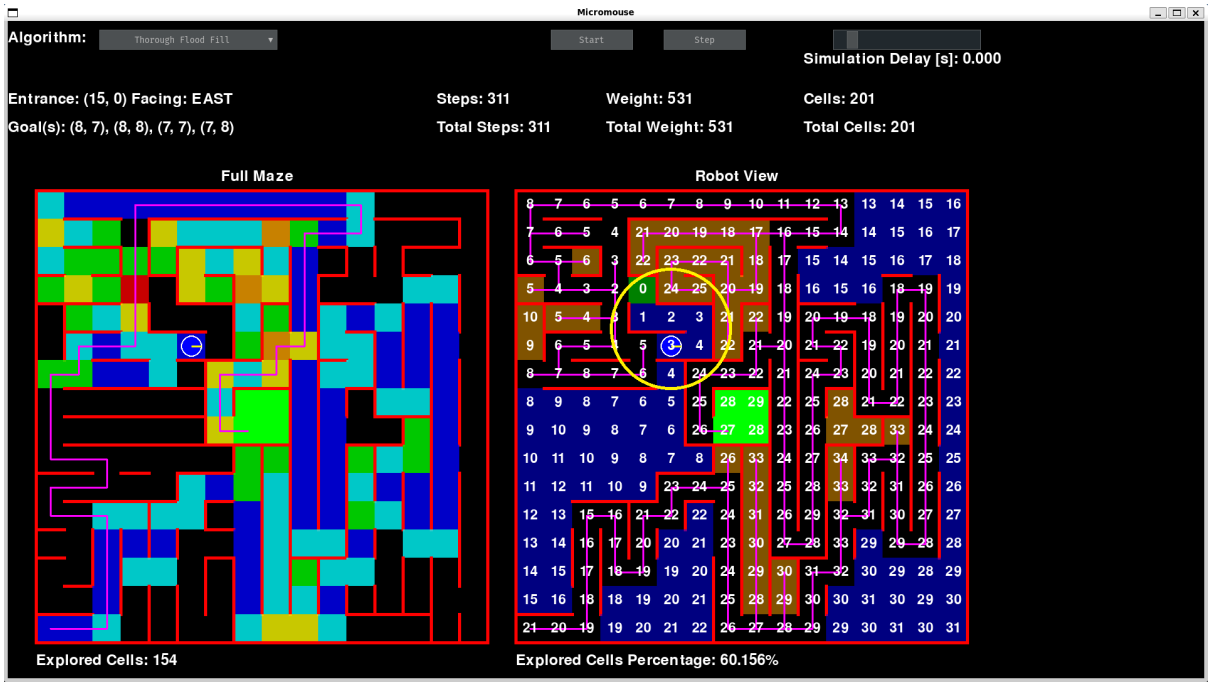


Figure 16: Thorough flood-fill before coloring dead-end using flood-fill

Figure 17 shows the state after the robot updated the flood-fill distances of the maze and found the group of cells that satisfy the dead-end conditions. Those cells are now a dead-end, and the robot won't get into them in the future while exploring the maze or solving it. Moreover, notice that the target cell the robot was trying to go to is now marked as explored, and the robot changed its exploration target.

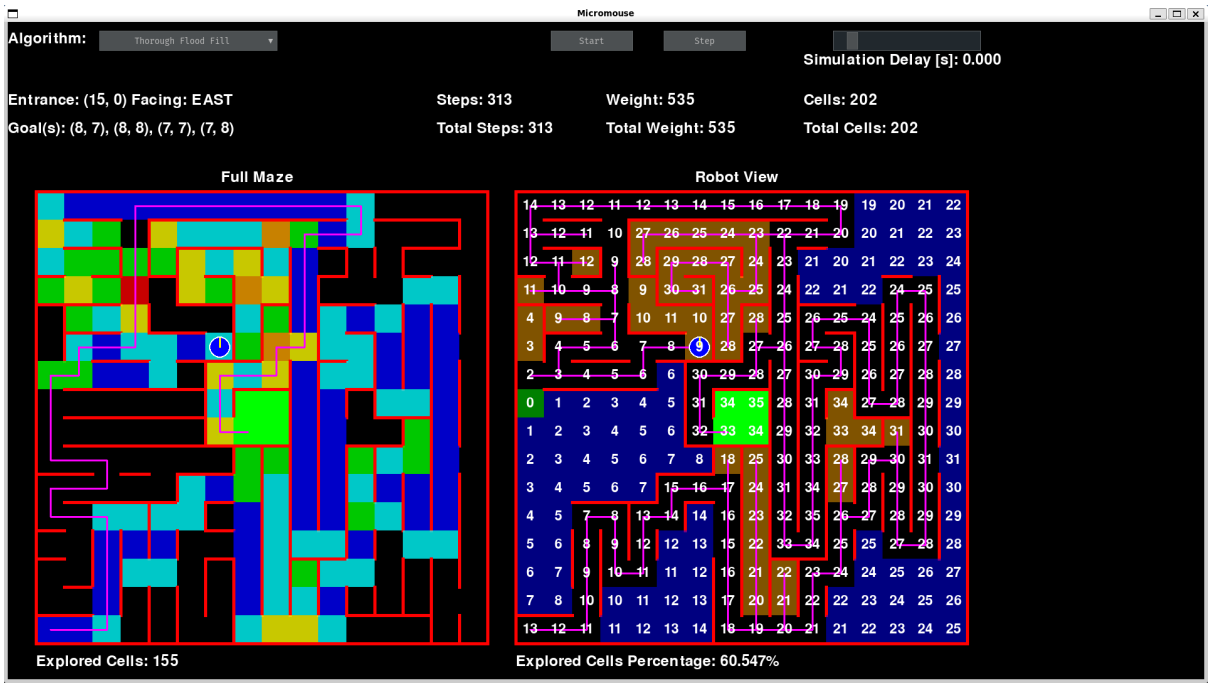


Figure 17: Thorough flood-fill after coloring dead-end using flood-fill

The problem with this approach is that the flood-fill algorithm uses distance from the goal as its weight function. As explained earlier, in section 6.2.4, the flood-fill algorithm finds the

shortest path and not the fastest one, which we seek. This property means that in some cases, the flood-fill dead-end detection algorithm might find a section of the maze not worth a visit and mark it as a dead-end even though it is part of the fastest path (but not the shortest). Therefore, Dijkstra's algorithm is run every time this algorithm finds a dead-end to check if the fastest path goes through a group of cells marked as a dead-end. If the fastest path goes through such a group, this group is then removed from the dead-end list and is re-added to the unexplored cells groups.

The flood-fill dead-end detection algorithm was tested with the maze from the Japan 2017 Micromouse competition (The maze appears in the explanation video mentioned in section 9.4 at 10 minutes and 30 seconds into the video). A thorough flood-fill explorer with flood-fill dead-end detection "competed" against an identical robot without flood-fill dead-end detection. Both robots found the optimal route. However, the exploration phase was different. Table 2 shows the number of actions each robot made during the exploration phase, the weight of those actions calculated by the time weights defined in section 6.1.2, and the number of cells each robot went through.

Flood-Fill Dead-End Detection	Steps	Weight	Cells
Off	762	1312	484
On	612	1042	394
Improvement	24.51%	25.91%	22.84%

Table 2: Flood-fill dead-end detection comparison

The improvement is defined to be the percentage error between the "on" and "off" values using the following formula:

$$Improvement = \frac{\text{Off} - \text{On}}{\text{Off}} \cdot 100\%$$

Conclusion Using the two methods above to detect dead-ends can significantly improve the exploration time while still exploring enough of the maze to find the fastest route in any maze.

7.3.4 Drawbacks

This solution solves the problem presented in section 7.2 and finds the fastest route with higher probability. Although the algorithm takes longer to explore the maze, it tries to do so efficiently.

8 Results and Conclusions

8.1 Results Evaluation

The Chinese 2009 IEEE Micromouse competition semifinal maze was used to evaluate the results. The maze can be seen in figure 3 in the full maze view (left) with the optimal solution in terms of the time weights as defined in section 6.1.2. Each algorithm was evaluated by the number of actions the robot made during the exploration phase and the solution, and the weight of the solution path by the time weights defined in section 6.1.2. Table 3 presents the results of each algorithm solving the maze.

Because the algorithms were tested in a simulation, measuring the time of running the algorithm, which is the scoring criterion in the Micromouse competition, has less meaning. We assume all algorithms use the same ”physical” robot. Therefore, the time it takes an algorithm to solve the maze is defined only by the actions it issues the robot.

The results include a comparison between the flood-fill algorithm described in section 6.2, the flood-fill with Dijkstra algorithm described in section 7.2, and the thorough flood-fill algorithm described in section 7.3. The results do not include algorithms described in section 5 because they either can not solve the maze or do it in a reasonable time.

Algorithm	Flood-Fill	Flood-Fill with Dijkstra	Thorough Flood-Fill
Number of Actions for Exploring	300	300	619
Number of Actions for Solution	83	81	69
Solution Weight	130	128	108

Table 3: Comparison between algorithms

8.2 Result Explanation

Both the flood-fill algorithm and the ”flood-fill with Dijkstra” algorithm have the same exploration method. The ”flood-fill with Dijkstra” algorithm achieved a better solution than the flood-fill algorithm alone because it tried to find the fastest route rather than the shortest one. The ”flood-fill with Dijkstra” algorithm does not continue to explore the maze and applies Dijkstra's algorithm on the maze as it is after the flood-fill algorithm. The explored maze lacks the information needed to find the optimal solution with Dijkstra's algorithm. The thorough flood-fill algorithm makes sure the information needed to find the optimal solution is available and thus has successfully found the fastest route. The trade-off of the thorough flood-fill algorithm is the longer exploration time portrayed by more actions for exploring the maze.

8.3 Conclusions

In different mazes, both the flood-fill and flood-fill with Dijkstra algorithms may yield different results where one is better than the other or both give the same results. The fluctuation is because our implementation of the flood-fill algorithm allows the robot to continue and explore the maze on its second run when trying to reach the goal after the initial run exploring the maze. Meaning the robot can step in unexplored cells. The flood-fill with Dijkstra's algorithm does not allow the robot to visit new cells on its second run and relies solely on the cells explored during the first run. But, if exploration time is less of a concern, then using the thorough flood-fill algorithm can yield the fastest route by the given time weights in any maze.

8.4 Other Possible Solutions

8.4.1 Partition-Center Algorithm

Another possible solution is the *partition-center algorithm*, which can be implemented and tested in the simulation. This algorithm divides the maze into partitions. The robot follows a different algorithm to find the fastest path possible [2] in each partition.

8.4.2 Other algorithms for finding the fastest path

Bellman–Ford Algorithm The Bellman-Ford algorithm can find the fastest path in a directed weighted graph. We considered using this algorithm but figured its complexity was very high compared to Dijkstra's algorithm. The complexity of this algorithm is given by $O(|V||E|)$ where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph. The primary difference to Dijkstra's algorithm is that the Bellman-Ford algorithm can detect negative-weight circles in the graph. However, negative weights don't make sense for time-based weights.

Johnson's Algorithm Johnson's algorithm is another algorithm to find the fastest path in a directed weighted graph. We considered using this algorithm but figured its complexity was very high. The complexity of this algorithm is given by $O(|V|^2 \log|V| + |V||E|)$ where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph. Moreover, this algorithm finds the fastest path between all pairs of vertices in the graph, meaning between every two cells in the maze. However, we only care about paths from the starting cell, meaning this algorithm does redundant work.

When the robot has no information about the maze, using any of the mentioned algorithms might help navigate the maze. But, every time the robot gains new information about the maze, it must repeat the algorithm to account for the change in the maze. The high complexity of those algorithms makes them less viable candidates for exploring and solving the maze. But, further research can validate this point.

8.5 Future Research

8.5.1 Diagonals

Our simulation only supports rotation in a 90° resolution. This limitation does not allow the robot to move freely in the maze as possible in a real Micromouse maze. Implementing support for rotations in other degrees, such as 45° rotations, will allow the robot to move diagonally across the maze. Supporting diagonal movement will require changing the implementation of the algorithms described in this paper to include such movements. Allowing the robot to move in diagonals or more freely can further improve the solving time and allow for a faster path in some cases.

8.5.2 Improve Thorough Flood-Fill Exploring

The thorough flood-fill algorithm exploring time is greatly affected by the route chosen for exploring. If the destination cells for exploring are chosen in a way the robot is traveling across cells already explored multiple times, the exploration time can grow drastically. Our solution takes the cell that is furthest away from the current cell and is part of the largest group of unexplored cells as the exploration goal, as described in section 7.3. This method can lead to the robot going back and forth across the maze and spending precious time doing so. It is possible that the best exploration method depends on the maze structure and might change from one maze to another.

8.5.3 Optimizing the Maze Representation

Our maze representation uses the ”maz” format covered in section 3.2.2. As seen in table 1, only the lower nibble is used. A more efficient maze representation can hold the information about the presence of walls of two cells in a single byte to save memory. Another possible way to optimize memory usage is saving data about the current cell in the high nibble, such as color, whether the cell was visited, and so on. The downside might be more computations and higher complexity to extract the data from the relevant bits.

8.5.4 Channels

Channels are sections of the maze where the robot can only move forward. The flood-fill algorithm can be optimized to perform fewer calculations when the robot is inside a channel. Although this improvement will not improve the number of actions the robot takes, it can improve the efficiency of the flood-fill algorithm, which, in turn, can improve the time it takes for the robot to solve the maze [3].

8.5.5 Testing in a Real Environment

Although the simulation is an easy testing tool for Micromouse solving algorithms, nothing compares to a real robot in a real maze. Our new proposed algorithm, Thorough Flood-Fill, described in section 7.3, has not been tested on a real machine. Testing the algorithm in a physical environment requires facing the mechanical challenge as well as the algorithmic one.

9 Appendix

9.1 Project's Code

The source code, installation instructions, and more are found on GitHub at the following link: `https://github.com/Roynecro97/micromouse-py-sim`.

9.2 Running The Simulation

9.2.1 Code Installation

1. Clone the repository:

   ```
   git clone https://github.com/Roynecro97/micromouse-py-sim &&
   cd micromouse-py-sim
   ```

2. Create a *virtualenv* (optional):

   ```
   python3.12 -m virtualenv .venv
   source .venv/bin/activate
   ```

3. Install all dev requirements:

   ```
   pip install -r dev-requirements.txt
   ```

9.2.2 Running

With the simulation installed, the *micromouse* command is automatically added to the path:

```
$ micromouse -h
usage: micromouse [-h] {sim,tool} ...

Micromouse simulator.

options:
-h, --help  show this help message and exit

subcommands:
  {sim,tool}
    sim       Run the simulator.
    tool      Run a tool.
```

Use the *micromouse sim* command to run the simulator. For example, from the git clone's directory, run:

```
$ micromouse sim -p semifinal
```

9.2.3 More Information

More information about using the simulation can be found in the link provided in section 9.1.

9.3 Micromouse Competition Rules

9.3.1 Introduction

Competition Description In this competition, the participant, or team of participants, must design and build an autonomous robotic "mouse" capable of traversing a maze of standard dimensions from a specified corner to its center in the shortest time possible.

9.3.2 Participant Eligibility

IEEE Membership All participants shall be registered IEEE student members attending the university they identified when registering for the SAC.

Team Composition Teams shall consist of one to five participants. A team of four or five participants shall not include more than two graduate students. A team of two or three participants shall not include more than one graduate student. Teams consisting of individual graduate students are not allowed. In the case of a team with more than one participant, it should be possible to demonstrate that each participant made a significant contribution.

Number of Teams There is no limit to the number of teams that one university may register for this competition.

9.3.3 Micromouseobot Rules

Fabrication The Micromouse robot submitted by a team must be designed and built from scratch.

Self Containment The Micromouse robot shall be self-contained (no remote controls). The robot shall not use an energy source employing a combustion process.

Dislodged Parts The Micromouse robot shall not separate from any part of itself while navigating the maze. To complete the maze, the robot, in its entirety, must enter the center of the maze.

Method of Movement The Micromouse robot shall not jump over, fly over, climb, scratch, cut, burn, mark, damage, or destroy the maze's walls.

Micromouse Size The Micromouse robot shall not be larger either in length or in width than 25 centimeters. The dimensions of a Micromouse robot that changes its geometry shall not be greater than 25cm x 25cm (length and width, respectively). There are no restrictions on the height of the robot.

Inspection All Micromouse robots are subject to inspection before starting their competition, to ensure they are within the specifications outlined by these rules, and that they do not pose potential safety hazards.

Rules Violation Any violation of these rules will constitute immediate disqualification from the contest and ineligibility for any associated prizes.

9.3.4 Maze Specifications

Maze Dimensions The maze is composed of 18cm x 18cm unit squares arranged to form a 16 x 16 unit grid. The walls of the maze units are 5 cm high and 1.2 cm thick (assume a 5% tolerance for mazes). An outside wall encloses the entire maze.

Maze Coloration 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 and finished with non-gloss paint.

Maze Fabrication Inconsistencies Do not assume the walls are consistently white, that the tops of the walls are consistently red, or that the floor is consistently black. Fading may occur and parts from different mazes may be used. Do not assume the floor provides a given amount of friction. It is simply painted plywood and may be quite slick. The maze floor may be constructed using multiple sheets of plywood. Therefore there may be a seam between the two sheets on which any low-hanging parts of the robot may snag.

Start/End Zones The starting square of the maze is located at one of the four corners. The starting square is bounded on three sides by walls. The start line is located between the first and second squares. As the mouse exits the corner square (signified by crossing the start line), the run timer starts. The destination is a gateway to the four-unit square at the center of the maze. The destination square has only one gateway.

Lattice Points Small square posts, each 1.2 cm x 1.2 cm, at the four corners of each unit square are called lattice points. The maze is assembled so that there is at least one wall at each lattice point.

Multiple Paths Multiple paths from the starting square to the destination square are allowed and should be expected. The destination square will be positioned so that a wall-hugging mouse will NOT be able to find it.

9.3.5 Competition Rules

Time Each team is allocated 10 minutes of access to the maze, starting when the competition administrator acknowledges the team and grants access to the maze. Any time used to adjust the team's Micromouse robot between runs is included in these 10 minutes. A run-time is

recorded for each run (from the start cell to the center zone) in which the robot successfully reaches the destination square. The minimum run time within the 10-minute trial shall be the mouse's official time.

Stopping/Removing the Micromouse Robot Each run shall be made from the starting square. Multiple runs, or run attempts, may be made within the allotted 10-minute maze time. The team may abort a run at any time, and return the Micromouse robot to the starting square. If the robot has reached the destination square and has acquired a "run time," the robot may take the maze back to the corner starting square on its own. Alternatively, it may be removed at any time without affecting the runtime of that run. If the robot is placed back in the maze at the starting square, a one-time penalty of 30 seconds will be added to the robot's next run time.

Reprogramming After Reveal After the competition maze is revealed to the teams at the start, the operator shall not reprogram his or her Micromouse robot but may elect to change the positions of switches on the robot.

Room Conditions The illumination, temperature, and humidity of the room shall be those of an ambient environment. (40 to 120 degrees F, 0% to 95% humidity, non-condensing).

Ambient Lighting Do not make any assumptions about the amount of sunlight, incident light, or fluorescent light that may be present at the competition site.

Run Time The run timer will start when the front edge of the Micromouse robot crosses the start line and stops when the front edge of the mouse crosses the finish line. The start line is at the boundary between the starting unit square and the next unit square clockwise. The finish line is at the entrance to the destination square.

Starting a Run A new run begins every time the Micromouse robot leaves the starting square. If the robot does not enter the destination square, no runtime is recorded. For example, if the robot re-enters the starting square (before entering the destination square) on a run, that run is aborted, and a new runtime will begin when the robot leaves the starting square.

Continued Navigation If the Micromouse robot continues to navigate the maze after reaching the destination square, the time taken will not count toward any run. The robot may and should make several runs without being touched by the operator. Once the robot has found the center, it is common practice to explore the maze via an alternate path on the return trip to the starting square.

Modifying the Robot The team may not feed information about the maze to the Micromouse robot. Therefore, changing ROMs or downloading programs is NOT allowed once the maze is revealed. However, participants are allowed to do the following:

- Change switch settings (e.g. to select algorithms).
- Replace batteries between runs.
- Adjust sensors.

- Change speed settings.
- Make repairs.

Changing the Robot's Weight After the team's 10-minute time allotment begins, they shall not alter their Micromouse robot in a manner that alters its weight (e.g. removal of a bulky sensor array or switching to lighter batteries to get better speed after mapping the maze is not allowed). The judges shall arbitrate all interactions with the robot.

9.3.6 Scoring

Regionality and Presentation There is only one official IEEE Micromouse contest each year in each area or region. All Micromouse robots, whether or not they have competed in previous contests, compete on an equal basis. All robots must be presented to the judges by the original team.

Scoring First place goes to the team with the shortest official time (without the team touching its Micromouse robot during runs). Second prize to the team with the next shortest time, and so on. Teams with the Micromouse robot that does not enter the center square will be ranked by the judges based on the following criteria:

- How close the robot gets to the destination square without being touched.
- Evidence that the mouse knows where it is relative to the destination square.

If, on occasion, the robot becomes immobilized in a corner or on a wall, the team may manually intervene to correct the problem (with care not to modify the mouse's intended direction of movement. The frequency of such corrections will be considered by the judges while scoring.

Requesting Breaks If requested, a break may be provided to the team after the completion of a run if another team is waiting to compete. The 10-minute timer will stop. When the team resumes their play, the 10-minute timer will continue. Judges and administrators shall determine whether breaks are allowed on a case-by-case basis.

Judges' Discretion The judges reserve the right to ask the team for an explanation of their Micromouse robot and its actions. The judges also reserve the right to stop a run, declare disqualification, or give instructions as appropriate (e.g., if the maze's structure is jeopardized by the mouse's continuing operation).

9.4 Explanation Video About The Competition

An informative video about the Micromouse competition made by *Veritasium* can be seen at `https://www.youtube.com/watch?v=ZMQbHMgK2rw`. This video can help visualize some of the algorithms discussed in this paper and gives additional info about the mechanical aspect of the challenge, which was not discussed in this work.

References

[1] AD-Codex. Micromouse_c_simulation_2022. https://github.com/AD-Codex/Micromouse_C_Simulation_2022.

[2] Jianping Cai, Jianzhong Wu, Meimei Huo, and Jian Huang. A micromouse maze sovling simulator. In *2010 2nd International Conference on Future Computer and Communication*, volume 3, pages V3–686–V3–689, 2010.

[3] Hongshe Dang, Jinguo Song, and Qin Guo. An efficient algorithm for robot maze-solving. In *2010 Second International Conference on Intelligent Human-Machine Systems and Cybernetics*, volume 2, pages 79–82, 2010.

[4] jimenezjose. Micromouse_simulator. https://github.com/jimenezjose/Micromouse_Simulator.

[5] mackorone. mms. https://github.com/mackorone/mms.

[6] Swati Mishra and Pankaj Bande. Maze solving algorithms for micro mouse. In *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*, pages 86–93, 2008.

[7] vineetvb. micromouse. https://github.com/vineetvb/micromouse.