Maze Solving Using the Flood Fill Algorithm

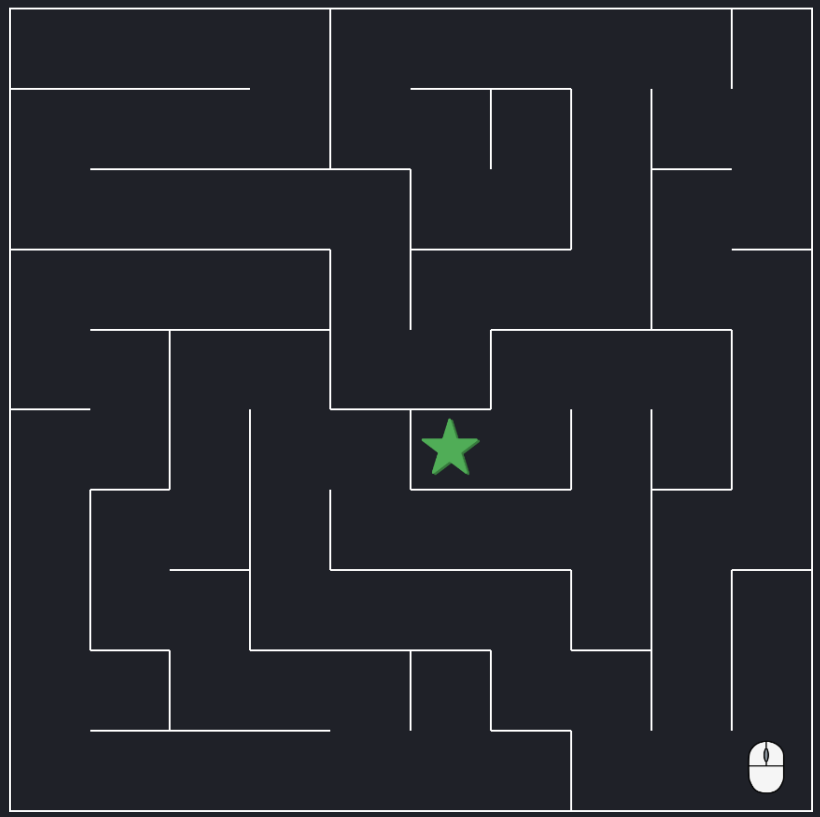
Rayan Bouhal and Niko Paraskevopoulos

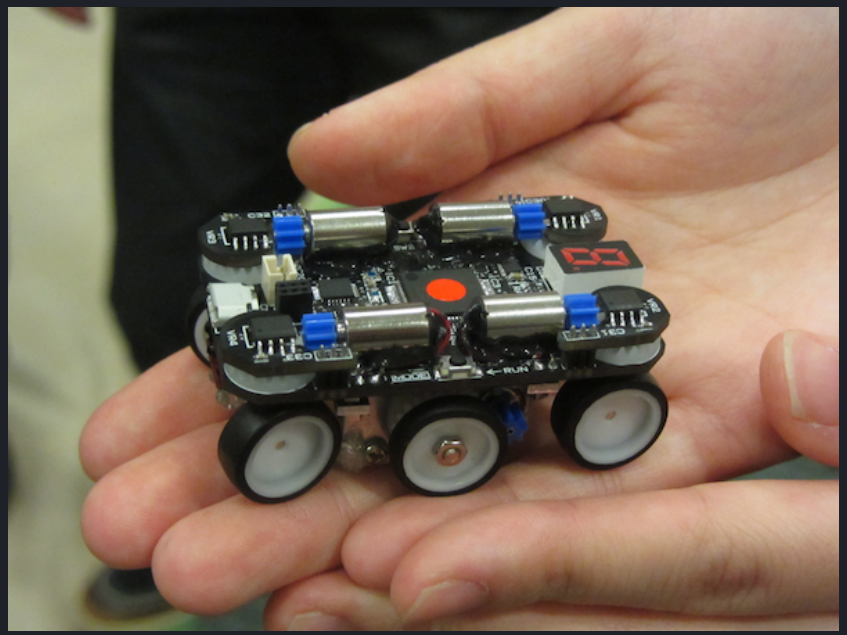
November 2023

Contents

Micromouse

- Micromouse is a robotics competition in which teams design a miniature autonomous vehicle that can intelligently solve a maze.
- The goal is to find the end of the maze in the shortest possible time.





[6]

Robotics

- A main component of Micromouse is scaling the hardware to a minature size while still allowing the mouse to move at top speeds and use infrared sensors to solve the maze on its own.
- This major feat requires it's own area of research which we will not cover.

Computer Science

- The second core part of the competition is coding an intelligent algorithm which can solve the maze.
- There have been several iterations of this algorithm since the competition started but currently the most widely used is the dynamic programming algorithm, Flood Fill. [3]

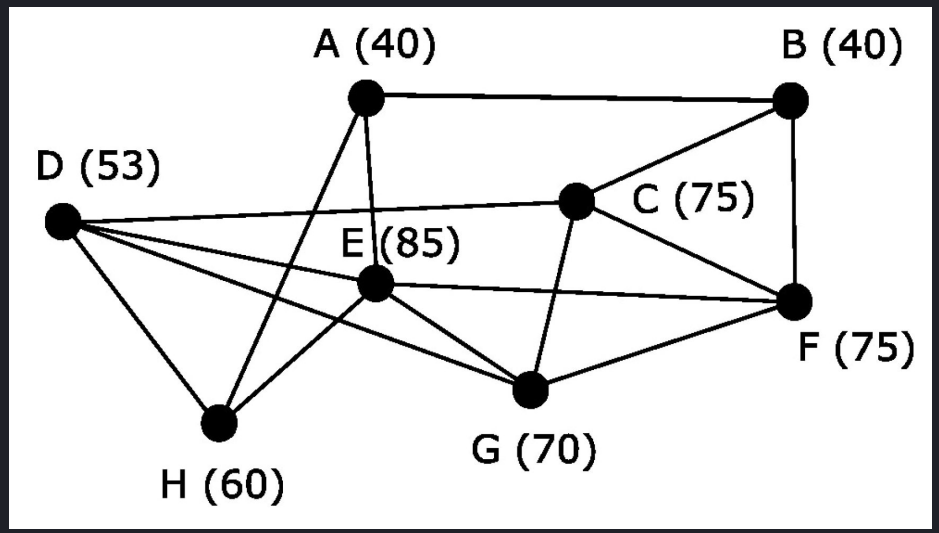
Combinatorics

The competition employs Combinatorial principles because the maze solving algorithm is designed using concepts from graph theory such as networks and shortest-paths. [5]

Definitions

Network:

- A graph where each edge is assigned a non-negative integer.
- Let's denote this value by k, where $k \in \mathbb{Z}$ and $k \geq 0$.



- The k values can clearly be seen in the graph depicted above, these values generally determine the weight, length, or cost of traversing an edge.

[5]

Definitions cont.

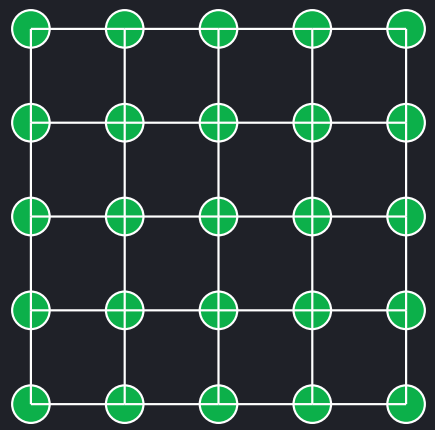
Shortest Path:

- Shortest Paths are a common problem for Network optimization.
- Generally, you are given a start and a destination and you need to find the shortest path between the two points.
- Depending on the size of your graph brute forcing this type of problem could take an unreasonable amount of time and effort.
- To optimize our efforts we use shortest path algorithms, which there are many of, but our focus today is the dynamic programming algorithm, Flood Fill.

[5]

Networks and Flood Fill

- The Flood Fill Netowrk weights or k-values are based on distance to the target destination.
- The network is most often visualized as a grid or matrix rather than a traditional graph with vertices and edges (as seen in the previous slide).



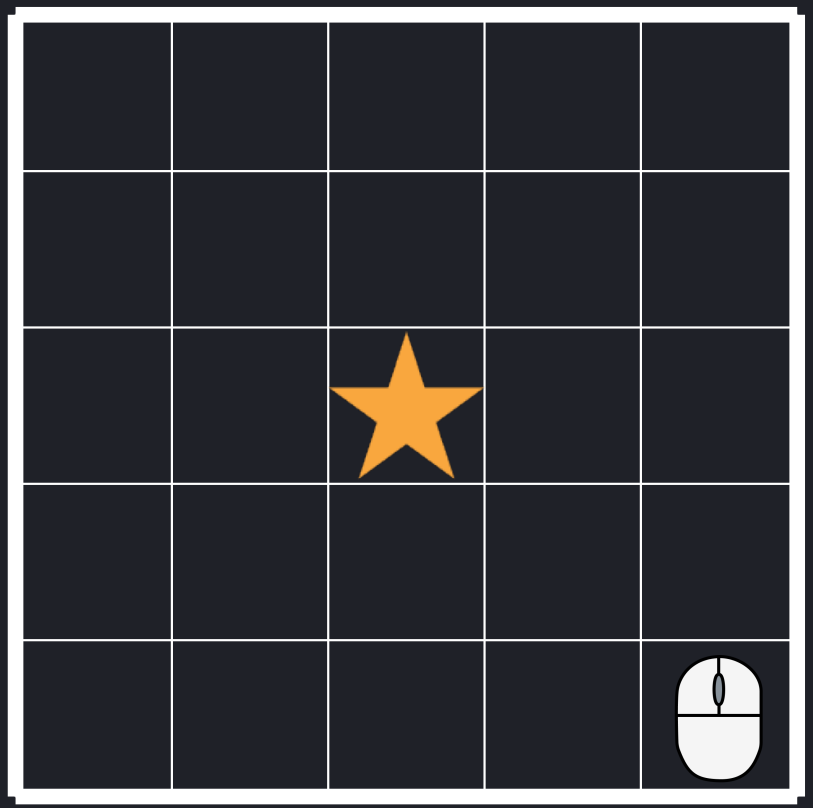
This is a visualization of an arbitrary graph that we could make into a network using the Flood Fill algorithm, where the green circles are vertices, and the white lines are edges.

Algorithm Steps

1. **Identify Start and Destination:** Choose the start vertex and the destination vertex. Assign a weight of 0 to the destination.
2. **Initialize Destination Edges:** Assign a weight of 1 to all edges directly connected to the destination vertex.
3. **Propagate Weights:** For each vertex reached by a weighted edge, assign a weight of 2 to all its unweighted edges, which are now highlighted in purple.
4. **Iterate and Assign Weights:** Continue the process of propagating the weights from vertices that have just been assigned a weight, incrementing the weight by 1 each time until all edges in the graph are weighted.

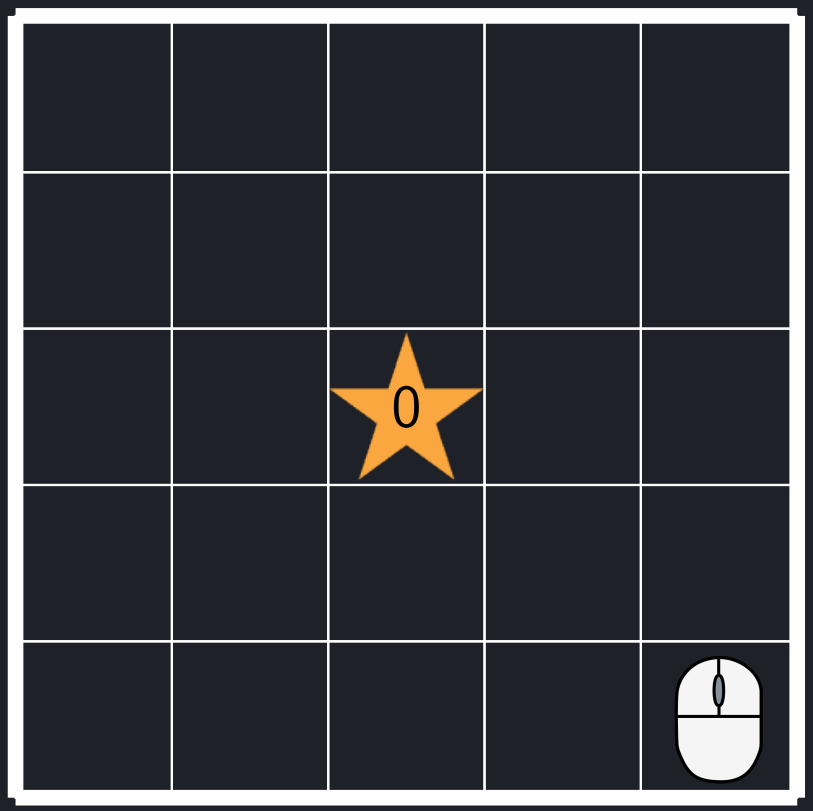
Step 1

- For simplicity and clarity we will depict our Network as a grid rather than a graph, with a final destination (marked by the star) and a start (marked by the top facing mouse)



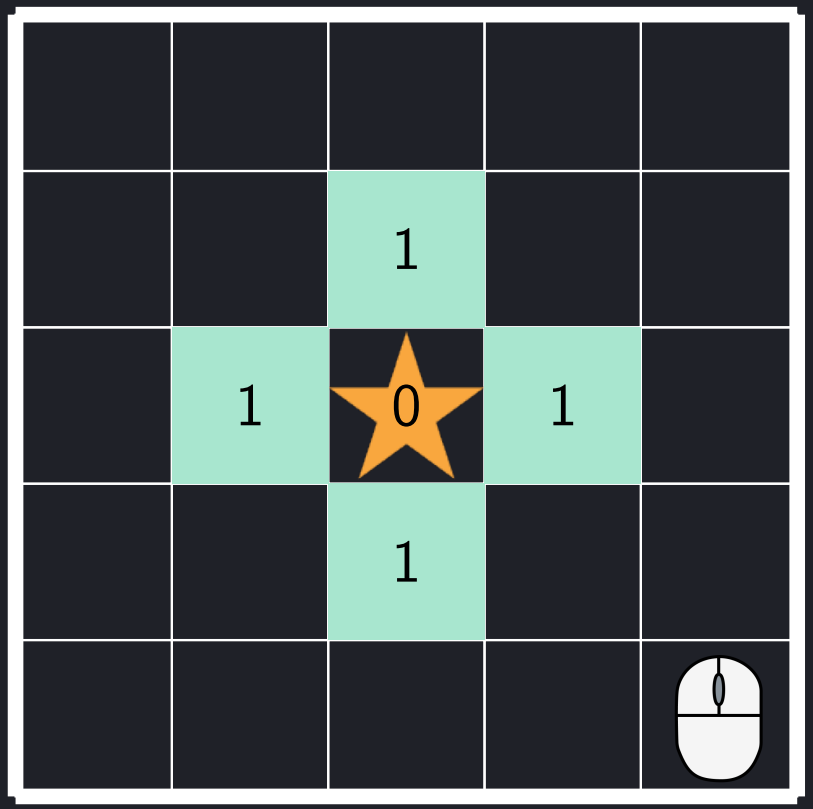
Step 2

- The destination is always assigned a weight or k-value of 0, because you cannot get any closer to the end goal.



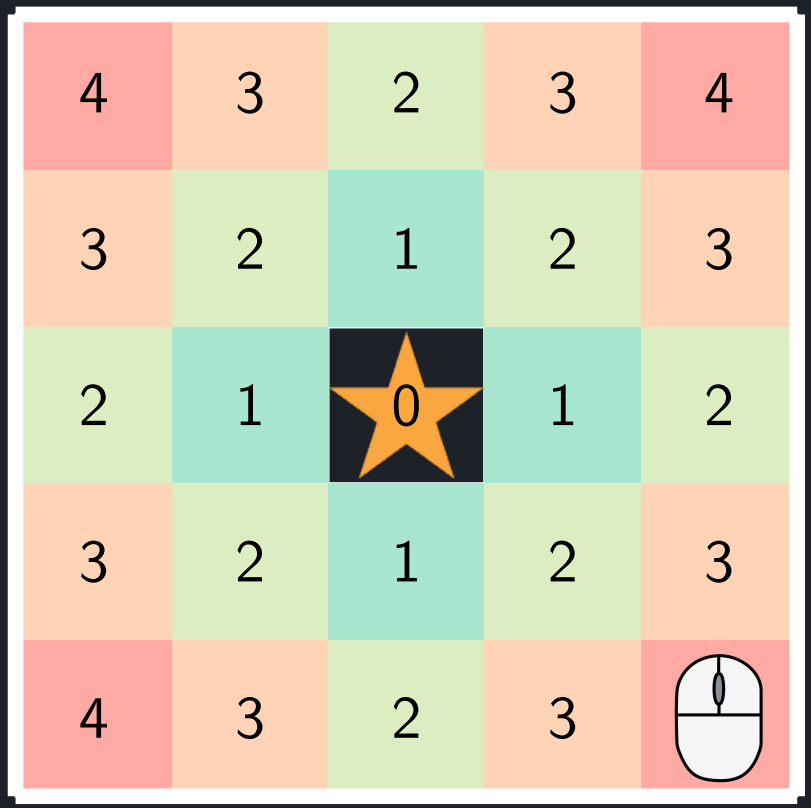
Step 3

- All squares that are next to the destination increase by 1
- For this specific implementation we will not move diagonally or consider diagonal squares.



Step 4 ...

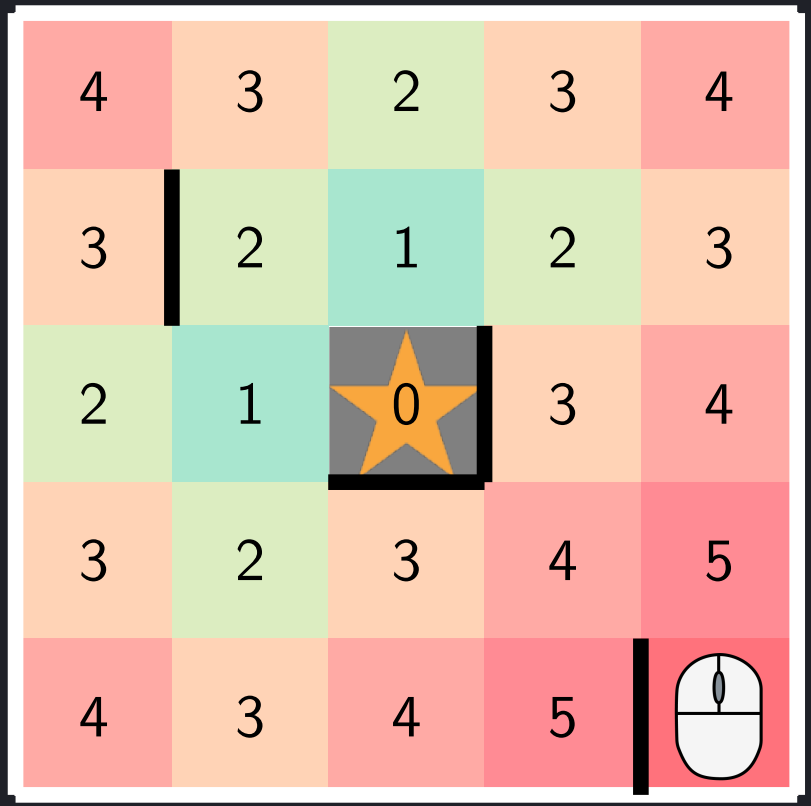
- Continue increasing k by 1 until all squares have been assigned a value



- Note: this grid is "blank" and has no obstacles or walls. Which in graph form would be represented by disconnected edges.

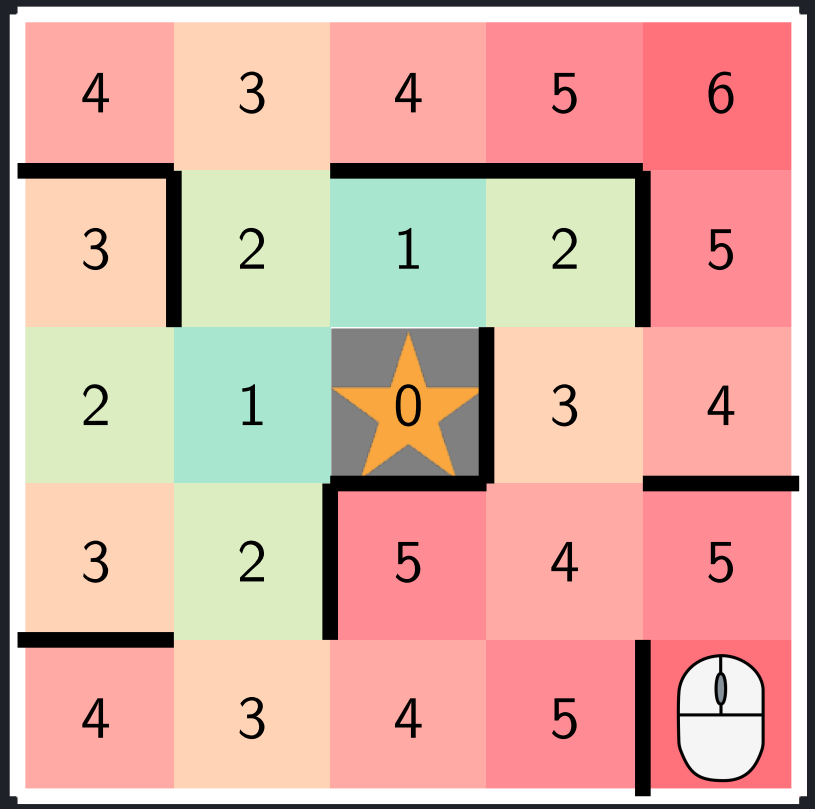
Recalculate k-Values

- Let's add a few walls and see how the k-values change



- This doesn't quite look like a maze yet, we should add more obstacles

Add More Walls



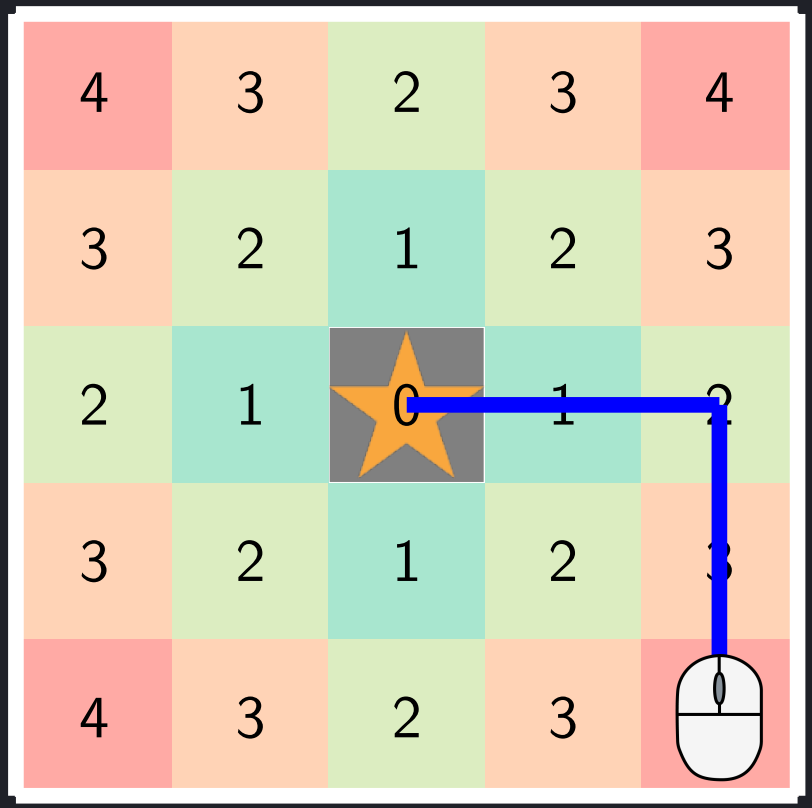
- This looks more like a maze and our Network is complete
- As you can see the k-values have changed significantly now that there are more obstacles

Finding the Shortest Path

Now that our network is finished, we can continue to use the Flood Fill algorithm to find the shortest path:

1. **Move to Lesser Value:** From the starting position, move to the square that has a lesser value than the current position.
2. **Resolve Ties with Straight Path:** If there is a tie, meaning two or more squares have the same value, prioritize the path that continues in the same direction as the previous step.

Shortest Path [No Walls]

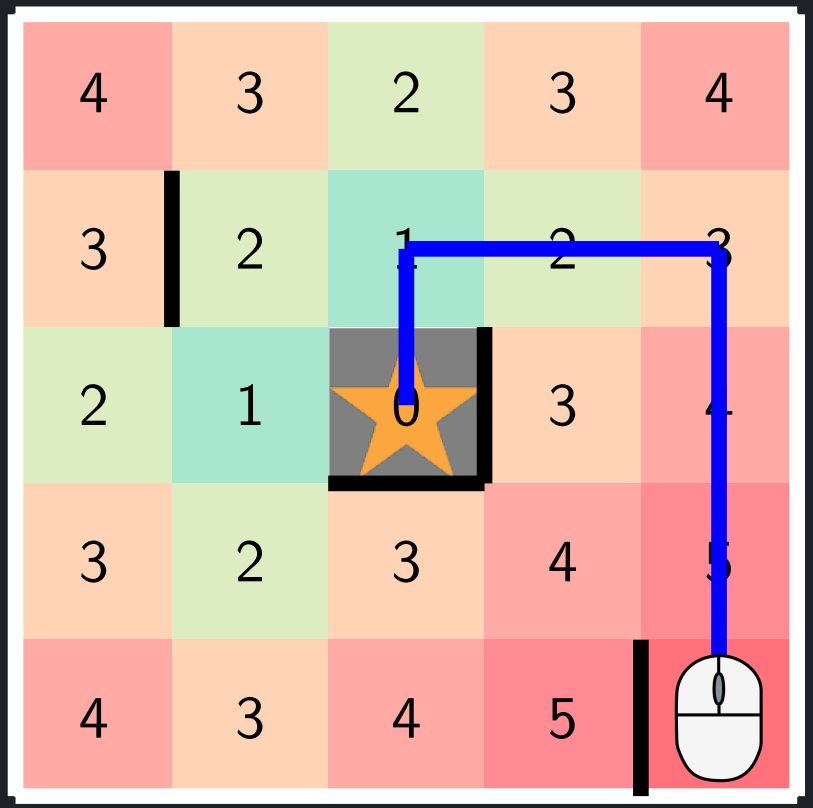


Based on our previous work we have this network of weighted paths. To find the shortest path:

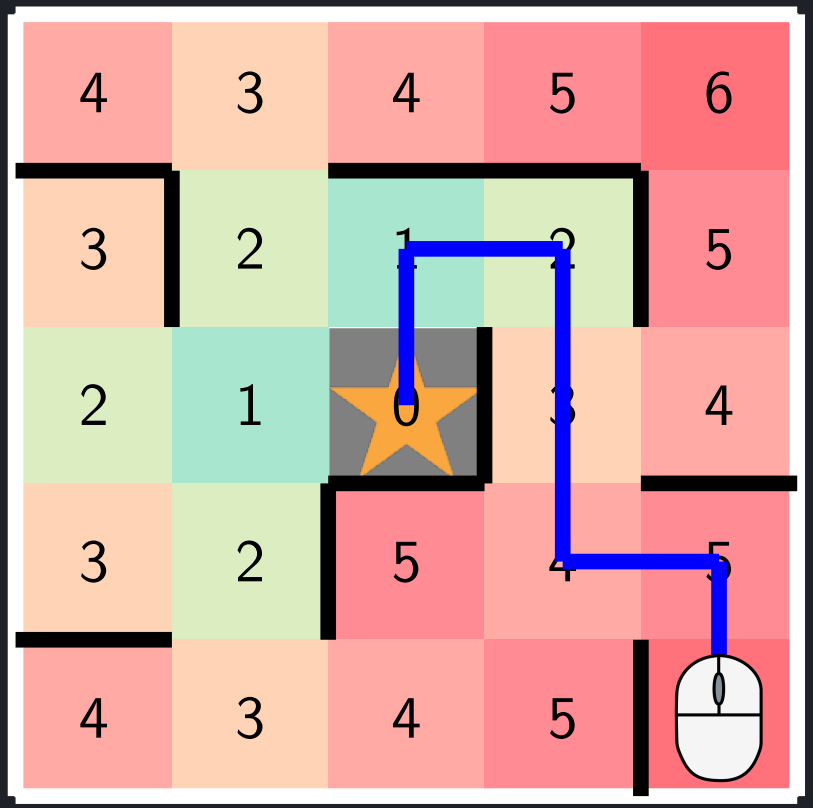
1. Move from the starting position to an adjacent square with a lower weight.
2. In the case of equal weights, choose the direction that is the straightest.

Follow these steps until you reach the destination, which is marked by the lowest weight on the network.

Shortest Path [Some Walls]



Shortest Path [More Walls]



Research

For our research we created a Micromouse Simulator. It is a small program that finds the shortest path in a randomly generated maze using the the flood fill algorithm.

- We modeled our approach after the work of four other papers. The first three describe coding the flood fill algorithm and the last describes creating a maze simulation. [2] [3] [4] [1] .

Generating a Maze

- The approach we took to generating a random maze was using a recursive Depth First Search (DFS) algorithm.
- This is the common approach used when programming mazes.
- The basic idea is that that we explore the deepest node on the graph, so the furthest one away, and then backtrack until we hit a node that we have already explored but have not branched to. Then we follow that branch and come back, and continue this until all nodes have been explored.

DFS Visualizer

This YouTube video shows a visualization of what the maze generation looks likes. Linked Here .

Demo

Try it Yourself (link)

Code Overview

- Written with JavaScript (backend) , CSS and HTML (GUI)
- Maze is randomly generated using a recursive Depth First Search (DFS) Algorithm
- Shortest Path is solved using Flood Fill Algorithm
  - The source code for the program can be found at `this Github Repository` .

Flood Fill

Flood Fill uses a recursive stack approach. We push neighboring cells onto the stack as we explore and pop them off when we're done. The stack ultimately holds our shortest path.

Flood Fill for Shortest Path Finding:

- `floodFill()` is a recursive function that searches for the shortest path from a starting point to a destination within a maze. It performs this by advancing to adjacent cells with lower weights and backtracking when no further progression is possible.
- It uses a `visited` flag within each cell to keep track of the cells that have already been explored, preventing the algorithm from revisiting the same cell and thus efficiently finding the shortest path.

Code Snippet

```
124  function floodFill(x, y, finishX, finishY, path = []) {
125      if (x < 0 || x >= mazeColumns || y < 0 || y >= mazeRows || maze[y][x].visited) {
126          return false;
127      }
128
129      maze[y][x].visited = true;
130      path.push({ x, y });
131
132      if (x === finishX && y === finishY) {
133          console.log("Finish reached at:", x, y);
134          return path;
135      }
136
137      console.log("Visiting:", x, y);
138
139      // Try each direction
140      if (!maze[y][x].top && floodFill(x, y - 1, finishX, finishY, path)) return path;
141      if (!maze[y][x].right && floodFill(x + 1, y, finishX, finishY, path)) return path;
142      if (!maze[y][x].bottom && floodFill(x, y + 1, finishX, finishY, path)) return path;
143      if (!maze[y][x].left && floodFill(x - 1, y, finishX, finishY, path)) return path;
144
145      // Backtrack: Remove the last element if all directions are blocked
146      path.pop();
147      return false;
148  }
```

DFS for Maze Carving:

- `generateMazeDFS()` is a function that uses Depth-First Search (DFS) to carve out a random maze by knocking down walls between adjacent cells. It ensures randomness by shuffling the order of directions before each recursive call.
- Walls are removed by updating both the maze data structure and the GUI, showing a real-time carving process on the display.

Code Snippet

```
// The recursive DFS function
function carvePath(x, y) {
    // Mark the current cell as visited
    maze[y][x].visited = true;

    let directions = [0, 1, 2, 3];
    shuffle(directions); // Shuffle directions to ensure randomness

    // Explore neighbors
    for (let i = 0; i < directions.length; i++) {
        let nextX = x + dx[directions[i]];
        let nextY = y + dy[directions[i]];

        // Check if the neighbor is within bounds and not visited
        if (isInBounds(nextX, nextY) && !maze[nextY][nextX].visited) {
            // Carve a path between the current cell and the neighbor
            if (directions[i] === 1) { // Right
                document.getElementById(`cell-${y}-${x}`).style.borderRight = "none";
                document.getElementById(`cell-${nextY}-${nextX}`).style.borderLeft = "none";
                maze[y][x].right = false;
                maze[nextY][nextX].left = false;
            } else if (directions[i] === 3) { // Left
                document.getElementById(`cell-${y}-${x}`).style.borderLeft = "none";
                document.getElementById(`cell-${nextY}-${nextX}`).style.borderRight = "none";
                maze[y][x].left = false;
                maze[nextY][nextX].right = false;
            } else if (directions[i] === 0) { // Up
                document.getElementById(`cell-${y}-${x}`).style.borderTop = "none";
                document.getElementById(`cell-${nextY}-${nextX}`).style.borderBottom = "none";
                maze[y][x].top = false;
                maze[nextY][nextX].bottom = false;
            } else if (directions[i] === 2) { // Down
                document.getElementById(`cell-${y}-${x}`).style.borderBottom = "none";
                document.getElementById(`cell-${nextY}-${nextX}`).style.borderTop = "none";
                maze[y][x].bottom = false;
                maze[nextY][nextX].top = false;
            }

            carvePath(nextX, nextY); // Recursive call
        }
    }
}

carvePath(0, 0); // Start carving from the top-left corner
}
```

GUI Code:

- The GUI represents the maze in a visual format using HTML and CSS. The maze is generated dynamically by creating a table with cells corresponding to the maze's grid structure.
- The `animateMouse()` function visualizes the pathfinding process by moving an icon through the cells that constitute the shortest path found by the flood fill algorithm, offering an animated solution to the maze.

Future Work

- Our project could be improved by adapting our algorithm to incorporate diagonal movement.
- The traditional Micromouse competition will have mice that are programmed to move on diagonals, but we omitted this feature for the sake of simplicity and time constraints.

Thank You!
Are there any questions?

[1] Martin Komák and Elena Pivarčiová. "Creating a Simulation Environment for the Micromouse". English. In: *TEM Journal* 11.1 (Feb. 2022). Copyright - © 2022. This work is published under https://creativecommons.org/licenses/by-nc-nd/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2022-09-06, pp. 479–483. URL: http://login.ezproxy.lib.vt.edu/login?url=https://www.proquest.com/scholarly-journals/creating-simulation-environment-micromouse/docview/2702222085/se-2.

[2] Swati Mishra and Pankaj Bande. "Advanced Algorithms for Micro Mouse Maze Solving". In: *Proceedings of the 2009 International Conference on Embedded Systems & Applications, ESA 2009, July 13-16, 2009, Las Vegas Nevada, USA*. Ed. by Hamid R. Arabnia and Ashu M. G. Solo. CSREA Press, 2009, pp. 78–84.

[3] Swati Mishra and Pankaj Bande. "Maze Solving Algorithms for Micro Mouse". In: *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*. 2008, pp. 86–93. DOI: 10.1109/SITIS.2008.104.

[4] Semuil Tjiharjadi, Marvin Wijaya, and Erwin Setiawan. "Optimization Maze Robot Using A* and Flood Fill Algorithm". In: *International Journal of Mechanical Engineering and Robotics Research* 6 (Sept. 2017), pp. 366–372. DOI: 10.18178/ijmerr.6.5.366-372.

[5] A. Tucker. *Applied Combinatorics, 6th Edition*. Online access: Center for Open Education Open Textbook Library. Wiley, 2012. ISBN: 9781118210116. URL: https://books.google.com/books?id=hdgbAAAAQBAJ.

[6] Bob White. *APEC MicroMouse Contest Rules*. 2004. URL: https://www.thierry-lequeu.fr/data/APEC/APEC_MicroMouse_Contest_Rules.html.