# Micromouse Competition

Matias Cinera
Computer Science Department, *University of South Florida*
Tampa, Florida, 33612, USA
cinera@usf.edu

## I. Introduction

This document is the report of my 6th lab from my Spring 2023 class CDA-6626 Autonomous Robots, as a graduate student. The individuals who are overviewing this project are our professor Dr. Alfredo Weitzenfeld (Professor in the department of Computer Science) and Mr. Chance Hamilton (Teaching Assistant & Graduate Student)

### A. Global Reference Frame

The objective of this lab exercise is to create a map of a world with walls, with dimensions of n by n. Afterwards, the task is to develop a logical method to navigate this world from a starting point to a destination point. For illustrative purposes, **I will be using a 4x4 world to demonstrate my implementation of the mapping algorithm and robot motions.**

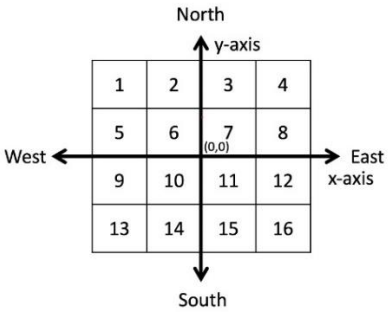


**Figure 1: Global reference frame and grid cell numbering scheme used to illustrate the implementation.**

## II. World Representation

The following terms will be used to explain my implementation:

- **Node / Tile**: A single cell on the grid (**Figure 1**)
- **Neighbors**: Any adjacent node, do not consider diagonal nodes as adjacent.
- **Robot Pose**: An object that keeps track of the robot's current x, y, theta, and current node/tile.

### A. Data Structures

- **Grid – 2d array:** A 2d array is used to represent the grid and keeps track of the visited tiles. An **undiscovered** tile has a value of 0, and a **discovered** one has a value of 1.

```
grid = [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]
```

**Figure 2: Grid 2d array, all tiles are initialized as undiscovered.**

- **Tile coordinates – 1d array:** Each entry in the 1d array holds 4 points which correspond to one tile. Since all **tiles** are squares of the same size, their corners (top left, top right, bottom left, bottom right) can be generated and stored in an array. By using the tile coordinates along with the **Robot Pose**, it's possible to update the current **tile** in the grid from undiscovered to discovered.
- **Adjacency List:** While traversing the grid, the algorithm will also build an adjacency matrix. This data structure is not used for traversing the grid, it is only used to map the world. Thus, the Adjacency list is mainly used for path planning. The algorithm will only add nodes when the robot moves from one **node** to another**.** The Adjacency list will be composed of a tile as a key, and a list of valid neighbors as the value of the key. **Valid neighbors** are neighbors from the current node that are not **blocked by any wall**. In **Figure 3**, Tiles 8 and 11 are not added to 7 neighbors since they are blocked by walls.

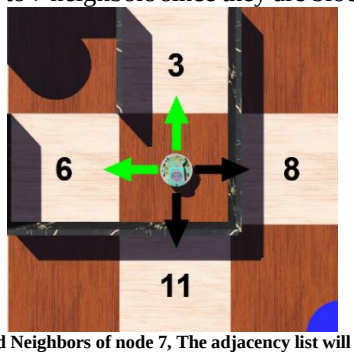


**Figure 3: Valid Neighbors of node 7, The adjacency list will add the key value pair: AdjList[Tile 7] = [Tile 3, Tile 6].**

## III. Mapping the World

To map the world, I employed an A-star algorithm and utilized the grid to track the discovered and undiscovered tiles. Additionally, I maintained a stack of motions to facilitate backtracking. I implemented an A-star search algorithm for Micromouse due to the time constraint of only having up to 10 minutes to map the world and locate the goal. However, finding the goal is more important than mapping the entire world. Given that the goal is situated at the center of the maze, A-star can efficiently map the tiles closest to the goal, making it an optimal choice.

### A. Discovered and Undiscovered Tiles

To keep track of the explored tiles I used the grid (2d array), as explained before a discovered tile will have the value of 1, and an undiscovered one will be 0. The algorithm also considers valid neighbors before traversing, meaning that the robot can only move from one tile to another if the target tile is **undiscovered** and **not blocked by a wall**.

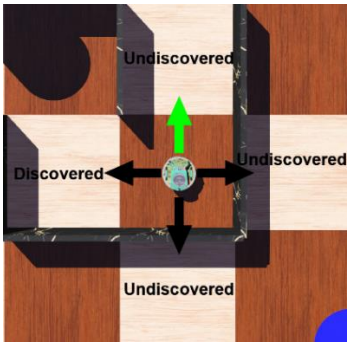

**Figure 4: The robot can only traverse to Tile 3, since Tile 6 is already discovered and tiles 8&11 are blocked by walls.**

*B.* Backtracking

Rather than utilizing a stack of nodes to traverse, I opted to use a stack of **motions**. Within my algorithm, I defined **motions** as the actions taken by the robot to transition between tiles.

**Figure 5: Forward motion of 10**

In **Figure 5**, the motion performed was moving forward 10 inches, thus this will be added to the stack: Stack = [ Straight Motion 10 Inches, …]. Backtracking happens when the robot cannot traverse to any tile, meaning when all neighbor tiles are **discovered** or **blocked by a wall**. When this happens pop the stack and reverse the motion poped. For example, if the stack's top had "Straight Motion 10 Inches", the robot will perform "Backwards-Straight Motion 10 Inches", by doing this the robot will backtrack to the last tile. The algorithm will keep backtracking until the current tile has at least one node that is **undiscovered** and **not blocked by a wall**. As seen in **Figure 6,** once the robot moves to tile 1, no valid tiles it can traverse to. Thus, it backtracks to tile 3, in which tile 7 is undiscovered and is not blocked by any wall.

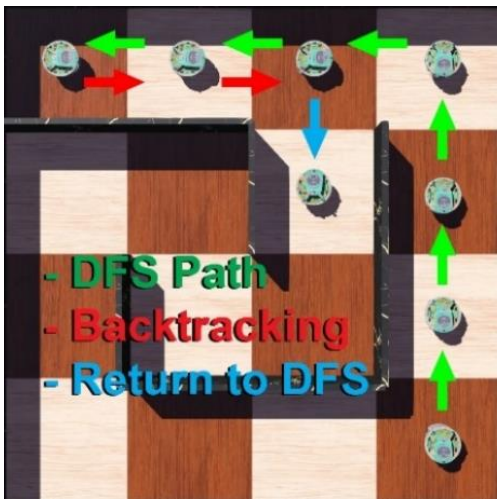


**Figure 6: Backtracking- DFS illustration, there is no goal in this map. The exploration is not affected by A-star**

*C.* A-star mapping

While conducting a DFS traversal, the exploration order of nodes can be arbitrary, such as up, down, left, and right. Nonetheless, if the location of the goal is known, *my algorithm* can calculate the distance from each *neighbor* tile to the goal. This way, *the algorithm* can identify the *tile closest* to the goal. As a result, whenever I have multiple options to choose from, I would select the adjacent tile that is closer to the goal.

*D.* Problems with A-star mapping

While A-star is optimal in terms of traversing to a neighbor node closest to the goal, optimal path to reach the goal for the MicroMouse competition. In my algorithm, once the goal is found I stop mapping, thus any other possible path to the goal is not explored. As seen in path of **Figure 7** [14, 15, 16, 12, 8, 4, 3, 7, 11], the algorithm will traverse from tile 14 to tile 15 (instead of tile 13) since that's its closest neighbor to the goal. However, this is clearly not the optimal path to tile 11. This are the drawbacks of a greedy algorithm (A-star), picking the closest neighbor to the goal will not guarantee the optimal path,

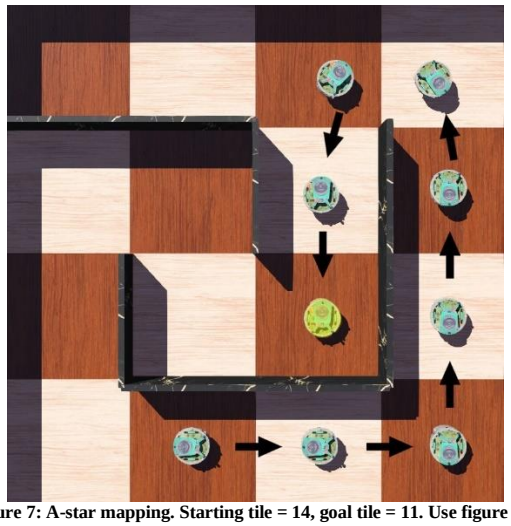

**Figure 7: A-star mapping. Starting tile = 14, goal tile = 11. Use figure 1 for referencing the tiles.**

*E.* Updating the global reference frame

The **Robot Pose** is used to update the **grid** (2d arr) data structure (note: The grid keeps track of the discovered nodes). Thus, when reading the distance sensors to check for walls, the robot needs to remap the readings to adjust to the global frame. For example, if the Robots current angle is 270°, its frontal sensor will be pointing South in the global frame

*F.* Traversing using local reference frame

The local frame is used to traverse through the maze. Although the readings of the robots are converted to update the global frame, the local frame is used to decide which node to travel to. For example, in **Figure 3**, (assuming only tile 7 is discovered), the robot can traverse to tiles 3 and 6. If given the choice, my algorithm will pick the neighbor tile north of the local frame. If that is not possible pick either of the sides, east or west.Grid Size & Starting node

*G.* Algorithm

The following algorithm asummes that all tiles in the maze can be explored. It also does not go into detail on how to perform the motion, and how to link the local frame to the global frame when traversing from one tile to another. This micromouse implementation assumes the starting location of the robot, thus A-star like search can be implemented. Assume the world has not been mapped in any-way, only assume the starting location the size of the world.

### H. Pseudo-code

```
motion_stack = create an empty stack
A-star_mapping()
    neighbor_tiles = get neighbor tiles of the current
      tile from the Robot Pose
    Adjacency List[current tile] = neighbors not blocked by walls
    if there are no valid neighbors in neighbor_tiles:
        Pop the motion_stack
        Use the motion popped to backtrack
    else:
        Traverse to the neighbor closest to the goal
        by only performing forward, or 90° in place motions
        append the performed motions to the motion_stack
    end if

main()
    cur_node = current node from the Robot Pose object
    while cur_node != goal_node:
        A-star_mapping()
    end while
        Print mapping time
        Save the Adjacency List
        Exit program
```

**Figure 8: A-star mapping algorithm**

## IV. Fastest Run

### A. BFS

After mapping the world, the goal of the "fastest run" is to find the shortest path from the starting tile to the goal (center of the maze). To achieve this, I implemented a BFS algorithm to find the shortest path between two nodes in a graph. Since I already have an Adjacency List, I could derive the shortest path by passing the starting and goal tiles to a BFS algorithm.

### B. Traversing a Path

I implemented specific algorithm to traverse a path in the mapped world. Instead of traversing the path with only forward motions and 90° turns, I used circular motions to optimize for a faster traversal. The motion functions were based the the motion library of Chance Hamilton[2], and adjusted for better accuracy. My algorithm has 3 different scenarios:

- **$\pi$ circular motions**: Half-circle motions happen when the next four traversing nodes form a square. Although, in this configuration making circular motions starting at the center of a tile will not be possible, since the sides of the robot will collide with the walls. Thus, the robot first moves forward to the edge of the current tile, then performs a circular motion to the edge of the target tile.

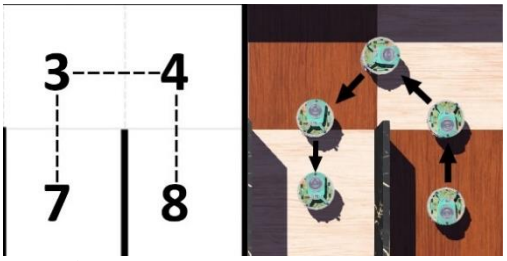


**Figure 9: Half-Circle Motion**

- **$\pi/2$ circular motions**: Quarter-circle motions happen when the next three traversing nodes form a curve. Only perform this motion if half-circle motions are not possible.

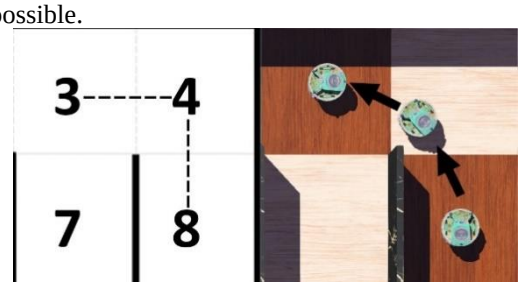


**Figure 10: Quarter-Circle Motion, path = [8, 4, 3]**

- **Forward**: If both circular motions cannot be performed, just perform a forward motion to the next tile (as seen in **Figure 5**).

## V. Conclusions

### A. Implementation

I think the algorithms implemented were optimal for the problem presented. However, my implementation was not. When implementing the mapping algorithm (DFS) I used a grid (2d array) and a list of tile coordinates (1d arr) along with an Adjacency List. However, the Adjacency List was not used in any way for mapping. Thus, I had 2 sets of data structures that essentially represented the same thing. Consequently, if given the chance to re-implement this lab, I would only use an Adjacency List to map the world.

My A-star mapping implementation cannot be run multiple times. When traversing the maze, I assume the world has not been mapped in any way. Because of this, it's possible I won't be able to find the most optimal path as in Figure 7. Thus, I plan to only run the mapping algorithm once and hopefully find the goal in under 10 minutes. When testing on sample worlds, the average time to find the goal was ~4 minutes.

### B. Problem with sensors

Another issue was the inaccurate lidar sensors. Sometimes when checking for walls in the maze, the lidar readings will return values that are not possible. For example, if the robot was 4 inches away from a wall, lidar would instead return an "inf" reading. This wrong reading implies that there is no wall, thus, it breaks the algorithm. The puck lidar sensors do have dead zones, this could have been the issue.

### C. Motions Accuracy

For both tasks (Mapping, Fastest-run) I had to fine-tune the motions performed by the robot. The margin of error for the motions had to be very low in order to traverse the map. This could be solved with a wall-follower implementation, however, I think it would be slower than the current one.

## References


[1] Webots Reference Manual. Webots. (n.d.). Retrieved 04, 2023, from https://www.cyberbotics.com/doc/guide/imu-sensors?version=develop

[2] Hamilton, C. [ChanceHamilton59]. Retrieved 04, 2023, GitHub. https://github.com/ChanceHamilton59