Mapping & Path Planning

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.

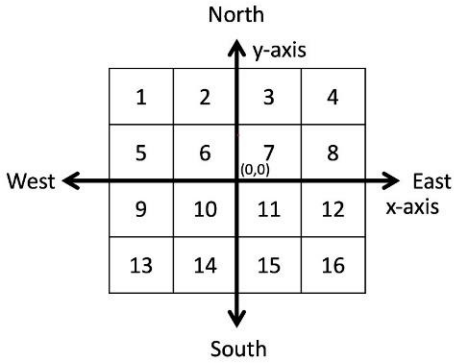

Figure 1: Global reference frame and grid cell numbering scheme used to illustrate the implementation.

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.

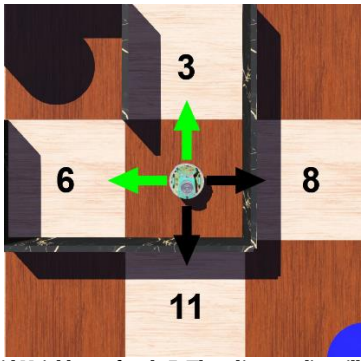

Figure 3: Valid Neighbors of node 7, The adjacency list will add the key value pair: AdjList[Tile 7] = [Tile 3, Tile 6].

III. Mapping the World

To map the world, I employed a DFS algorithm and utilized the grid to track the discovered and undiscovered tiles. Additionally, I maintained a stack of motions to facilitate backtracking.

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**.

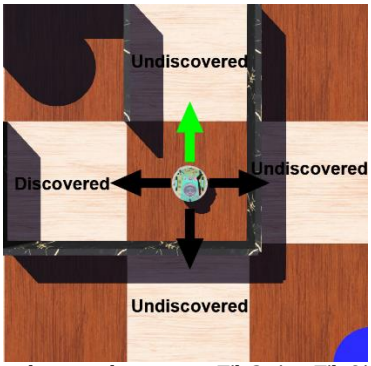

Figure 4: The robot can only traverse to Tile 3, since Tile 6 is already discovered and tiles 8&11 are blocked by walls.

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.

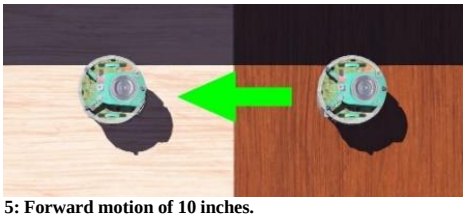

Figure 5: Forward motion of 10 inches.

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 illustration.

C. 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

D. 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

E. Starting location

In my algorithm a starting node is not required to map the world. The algorithm only requires knowledge of the size of the **grid**, which is then quadrupled. For instance, if the original matrix is 4x4, then an 8x8 matrix will be used instead. This allows for the initial position of the robot to be set as the center of the new grid. As the grid is explored, there is no risk of surpassing the boundaries of the matrix. After the robot has finished exploring/mapping the world, the graph is then normalized.

F. 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. Since the starting location is not needed to map the world, a normalization function is needed for the **adjacency list.** The normalization function just reformats the nodes to the initial global reference frame (**Figure 1**).

Pseudo-code:

```
motion_stack = create an empty stack
DFS_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 any of the available neighbors
        by performing forward, or 90° in place motions
        append the performed motions to the motion_stack
    end if

main()
    target_nodes = number of tiles in maze
    while visited nodes != target_nodes:
        DFS_mapping()
    end while
        Print mapping time
        Normalize the Adjacency List
        Save the Adjacency List
        Exit program
```

IV. Path Planning

A. 4-point Wave-Front Planner

Once the mapping is finished, use the adjacency list to build a 4-point wave-front planner. For my implementation, the starting node has a value of 0 instead of 2 and walls are represented with "|" instead of a 1. The only reason for the changes was to simplify the code. Regarding the algorithm, I retrieve the starting node neighbors and perform a BFS traversal. Whenever a new node is being explored, its Wave-Front value will be calculated and stored. After the BFS traversal finishes, all nodes will have a cost (value previously calculated) in relation to the starting node. The cost is just the number of nodes in the shortest path between the starting node and the target node.

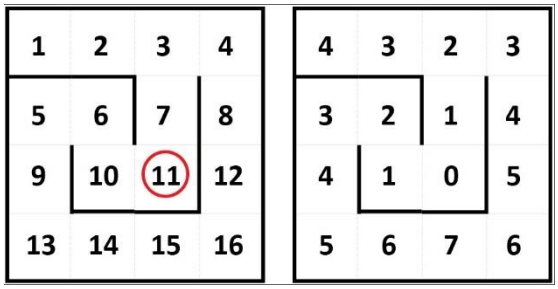

Figure 7: Illustration of wave front planner starting at tile 11, starting node is represented with a 0 instead of a 2.

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. My algorithm has 3 different scenarios:

- **π circular motions**: Half-circle motions happen when the next 4 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.

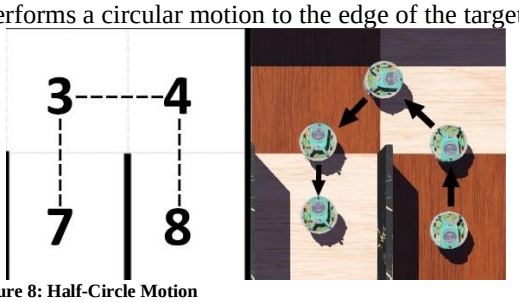

Figure 8: Half-Circle Motion

- **π/2 circular motions**: Quarter-circle motions happen when the next 3 traversing nodes form a curve. Only perform this motion if half-circle motions are not possible.

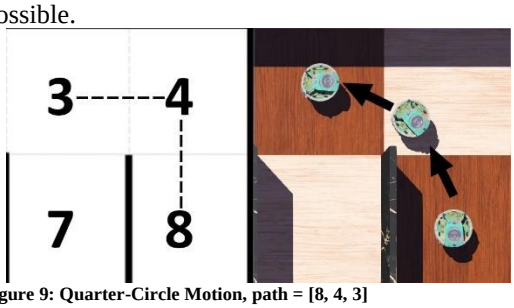

Figure 9: Quarter-Circle Motion, path = [8, 4, 3]

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

V. Conclusions

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.

Another issue were 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" readings. This wrong reading implies that there is no wall, thus, it breakes the algorithm. The epuck lidar sensors do have a deadzones, this could have been the issue.

References

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