Tank Robot Pathfinding with A* and Greedy Search Algorithms

Problem formulation

The project involves implementing a pathfinding solution for a tank robot equipped with an Arduino Uno and a gyroscope for precise navigation. The robot operates on a 5x5 grid matrix where obstacles (tanks cannot go) are marked as 1 and free cells (tank can go) as 0. Starting from the top-left corner (0,0) and aiming for a designated end state (determined by us), the goal is to find the most optimal path while avoiding obstacles. This task represents a pathfinding problem, specifically a graph search task, which will be solved using two algorithms: A* and Greedy Search.

EDA and data preprocessing

In this project, we did not use any external dataset because the grid-based environment for our robot navigation was predefined. Instead of relying on real-world or collected data, we manually designed the matrix that represents the environment, including traversal costs and obstacles. This predefined grid allowed us to focus on implementing and testing pathfinding algorithms like A* and Greedy Best-First Search in a controlled environment. As a result, there was no need for traditional data preprocessing or exploratory data analysis (EDA), since the grid structure and its values were already known and directly used as inputs for the algorithms.

Implementation

Hardware Set Up:

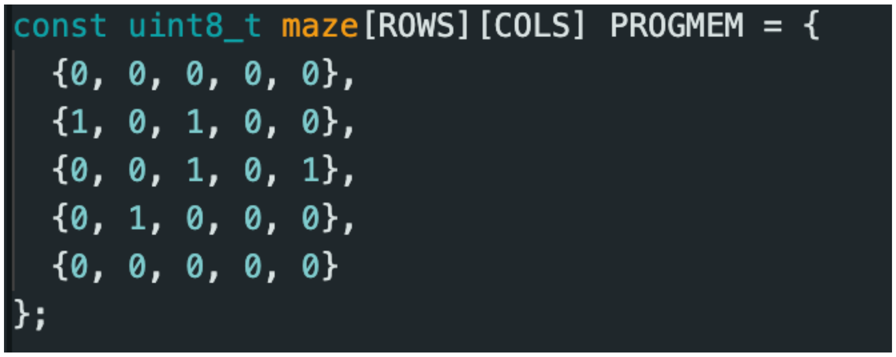
We build our tank from scratch by connecting all components ourselves. At its core is the Arduino Uno, which acts as the main brain, handling all the sensor inputs and controlling the motors. The robot is equipped with four DC motors connected to the Arduino, allowing it to move forward, backward, and turn with ease. To keep track of its direction, we added an MPU6050 gyroscope, which communicates with the Arduino through the I2C interface using the Wire library. This gyroscope provides real-time feedback on the robot's orientation, ensuring that its turns are precise and movements stay stable. A simple push-button lets users start the pathfinding process with just a press, thanks to built-in pull-up resistors that make the button reliable. We also included adjustable motor speed controls, allowing us to fine-tune how responsive and dynamic the robot's movements are.

Main Algorithm:

For this project, we implemented the A* algorithm to enable our tank robot to navigate a predefined 5x5 grid efficiently. The A* algorithm was chosen due to its ability to find the shortest possible path by considering both the actual movement cost from the start point (gCost) and the estimated cost to reach the destination (hCost) using a heuristic. This dual consideration ensures that the robot not only moves towards the goal but does so in the most optimal manner, balancing speed and efficiency.

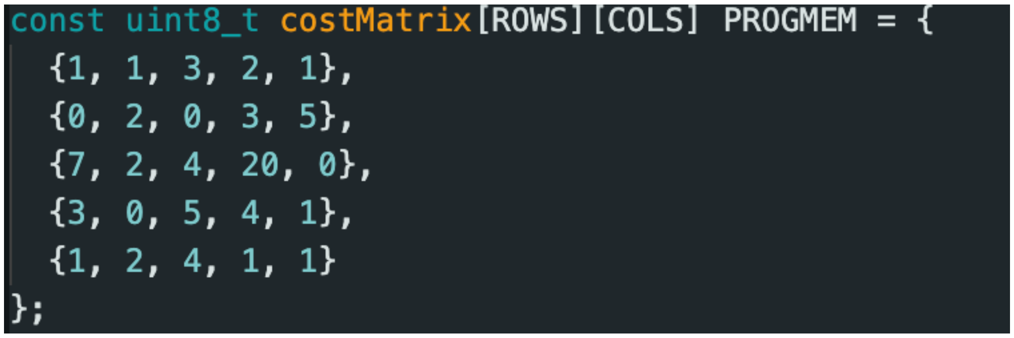
Setting Up the Environment Representation:

- Grid Definition: We defined the environment as a 5x5 grid, where each cell represents a possible position the robot can occupy. The grid was implemented using two matrices stored in PROGMEM to conserve the limited SRAM available on the Arduino Uno:
    - Maze Matrix (maze): Indicates the presence of obstacles (1) and free cells (0). This matrix helps the robot identify which paths are navigable:



```
const uint8_t maze[ROWS][COLS] PROGMEM = {
  {0, 0, 0, 0, 0},
  {1, 0, 1, 0, 0},
  {0, 0, 1, 0, 1},
  {0, 1, 0, 0, 0},
  {0, 0, 0, 0, 0}
};
```

    - Cost Matrix (costMatrix): Assigns a traversal cost to each cell, with higher values representing more challenging or costly paths. We defined costs randomly for demonstration. Specifically, we set obstacles and costs so that when we execute two different algorithms (A* and Greedy), paths are different:

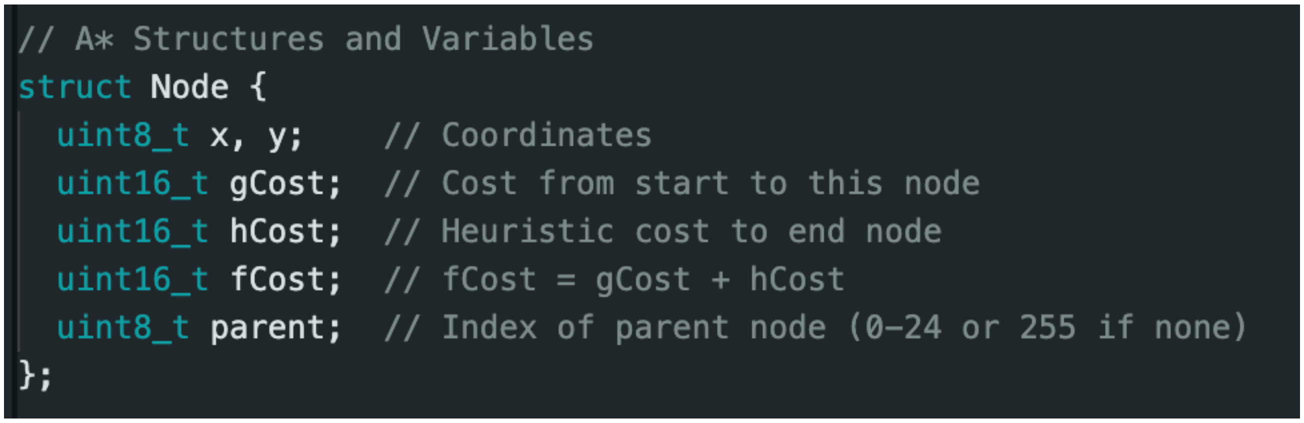


```
const uint8_t costMatrix[ROWS][COLS] PROGMEM = {
  {1, 1, 3, 2, 1},
  {0, 2, 0, 3, 5},
  {7, 2, 4, 20, 0},
  {3, 0, 5, 4, 1},
  {1, 2, 4, 1, 1}
};
```

    - Start and End Points: The robot's journey begins at the top-left corner of the grid (0,0) and aims to reach the bottom-right corner (4,4).

    - **The environment in this project is considered a static, discrete, deterministic, and fully observable grid-based environment.**

- Designing Data Structures:
    - Node Structure: We defined a Node struct containing the coordinates (x, y), movement costs (gCost, hCost, fCost), and a parent index to trace the path back to the start point. This structure is pivotal for the A* algorithm to evaluate and compare different paths.



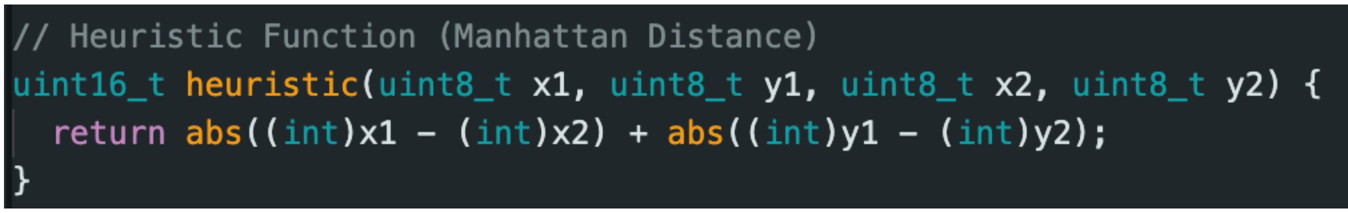
```
// A* Structures and Variables
struct Node {
  uint8_t x, y;     // Coordinates
  uint16_t gCost;  // Cost from start to this node
  uint16_t hCost;  // Heuristic cost to end node
  uint16_t fCost;  // fCost = gCost + hCost
  uint8_t parent;  // Index of parent node (0-24 or 255 if none)
};
```

- Open and Closed Lists: Implemented as arrays, the open list holds nodes that are yet to be evaluated, while the closed list contains nodes that have already been assessed. Managing these lists efficiently is crucial for the algorithm's performance.

- Path Array: Once the algorithm identifies the goal, it reconstructs the path by tracing back through the parent nodes and stores this sequence in the path array.

- Implementing the A* Algorithm:

  - Heuristic Function: We employed the Manhattan distance as our heuristic, calculating the sum of the absolute differences in the x and y coordinates between any node and the goal. This choice aligns well with grid-based movement, providing a straightforward and effective estimation:



```
// Heuristic Function (Manhattan Distance)
uint16_t heuristic(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2) {
  return abs((int)x1 - (int)x2) + abs((int)y1 - (int)y2);
}
```

  - Starting the Algorithm (Algorithm Initialization):
    - We began by setting up our starting point:
      - gCost: This is the actual cost to move from the start to the current point. For the start node, this cost is 0 because we're already there.
      - hCost: This is our estimated cost to reach the goal from the current point, calculated using the Manhattan distance (Heuristic).
      - fCost: This is the total estimated cost to reach the goal, which is simply gCost + hCost.

    Note: By adding the start node to our "open list" (a list of nodes we need to explore), we kick off the search for the best path.

  - The Main Process (Search Loop) -- > This is the heart of the A* algorithm, where the magic happens. It involves several key steps:
    - Picking the Best Node to Explore Next (Node Selection): From our open list, we always choose the node with the lowest fCost. This means we're picking the node

that seems closest to the goal when considering both the cost so far and our estimate.

    - Checking if We've Reached the Goal (Goal Check): After selecting a node, we checked if it's our end point. If it is, we stop the search because we've found our path.
    - Looking at Nearby Points (Neighbor Exploration): For the current node, we examined its four neighbors: up, down, left, and right. We check: Is the neighbor inside our grid? We don't want to go outside our defined area, Is the neighbor a free space? We can't move through walls or obstacles, Have we already evaluated this neighbor? If it's in the closed list, we've already considered it.
    - Calculating Costs: Tentative gCost: We added the movement cost from our costMatrix to the current node's gCost to get the tentative cost to reach this neighbor.
    - Updating the Open List: If the neighbor is already in the open list and the new path is cheaper: We update its costs and set the current node as its parent; If the neighbor is not in the open list: We add it to the open list with its calculated costs and parent information.
  - Building the Path (Path Reconstruction): Once we reached the end point, we traced back from the goal to the start by following each node's parent.

Evaluation of Algorithm:

1. Path Verification: We printed the reconstructed path to the Serial Monitor, ensuring that the sequence of steps logically progresses from the start to the goal without unnecessary detours. Visualizing the path steps on the Serial Monitor confirmed that the algorithm correctly traced the optimal route, ensuring the integrity of the pathfinding process.

2. Real-World Testing and Optimization: In addition, we uploaded the code to Arduino with different paths several times and run our code on the matrix prepared by us. We optimized movements and logic accordingly after each run and made sure our tank performs well on. Movements and perfectly find each optimal path with A*.

3. Comparison with Greedy Best-First Search: To further evaluate the effectiveness of our A* implementation, we also implemented the Greedy Best-First Search algorithm and compared its performance against A* on the same grid setups.

Comparison Algorithm:

In addition to A*, for comparison purpose, we also implemented Greedy Best-First Search. For our Greedy Best-First Search approach, we decided to simplify the pathfinding process by focusing solely on the estimated distance to the goal, known as the heuristic cost (hCost). Unlike the A* algorithm, which considers both the actual cost from the start point (gCost) and the estimated cost to reach the goal, Greedy Search only looks at how close each option appears to the destination based on the Manhattan distance.

In practical terms, this means that at every step, the algorithm picks the path that seems closest to the goal without taking into account movement cost. By doing this, managing the list of possible paths becomes much simpler and requires less computational power.

Experiments

We ran several experiments with different mazes (still 5x5 matrix but we changed obstacles to make sure our algorithm works well) and cost matrices. Here, we put last experiment with the details. Maze we used for this experiment is the following:

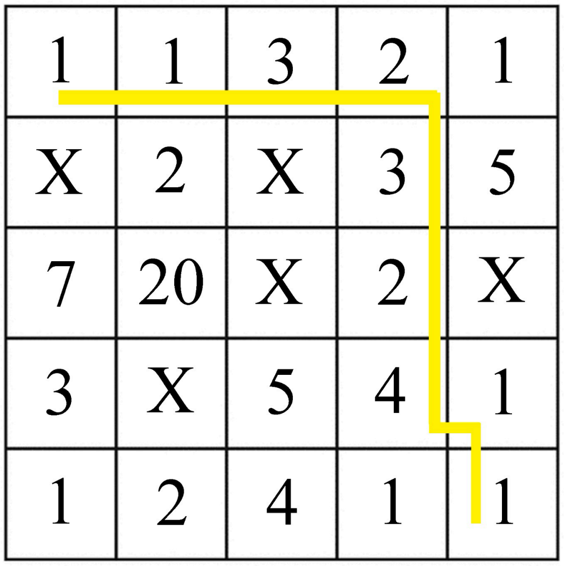
X		X		
		X		X
	X			

Here, X shows that there is obstacle on this cell and tank should avoid to enter this cell. Other cells are free and can be used when path is being found. Then, we added costs to our matrix (Costs are added randomly):

1	1	3	2	1
X	2	X	3	5
7	20	X	2	X
3	X	5	4	1
1	2	4	1	1

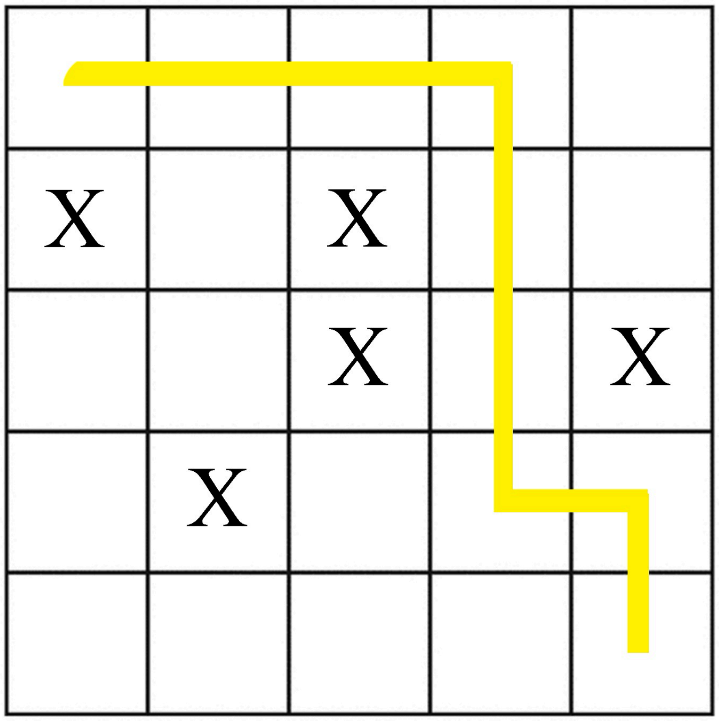
In this case, the most optimal path with A* would be the following:

Optimal path: (0,0) → (0,1) → (0,2) → (0,3) → (1,3) → (2,3) → (3,3) → (4,3) → (4,4):



1	1	3	2	1
X	2	X	3	5
7	20	X	2	X
3	X	5	4	1
1	2	4	1	1

However, since it is also the shortest path, Greedy Best-First Search algorithm also gives the same path. Here we put visualization of it (we removed costs since Greedy Best-First Search uses only heuristic):

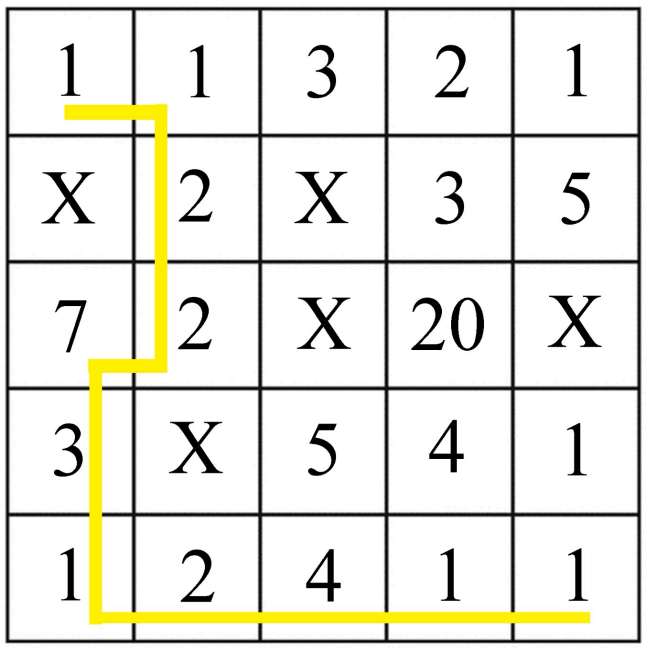


X		X		
		X		X
	X			

To be able to compare and see the difference, we changed our cost matrix:

1	1	3	2	1
X	2	X	3	5
7	2	X	20	X
3	X	5	4	1
1	2	4	1	1

Here, we gave more cost to the cell (2,3). Therefore, our A* uses different path but Greedy Best-First remains the same. Here is the new path found by A*:



1	1	3	2	1
X	2	X	3	5
7	2	X	20	X
3	X	5	4	1
1	2	4	1	1

Discussion of results:

The results of our project demonstrate the strengths and trade-offs of using A* and Greedy Best-First Search algorithms for pathfinding in a grid-based environment. In the initial experiment, where traversal costs were evenly distributed, both A* and Greedy Search produced the same path since the shortest path also happened to be the least costly. This highlights how Greedy Search, by relying solely on the heuristic (Manhattan distance), can sometimes match A*'s results when the cost distribution is simple.

However, in the modified experiment, we increased the traversal cost of a specific cell (2,3), introducing a scenario where the shortest path was no longer the most optimal path. A* successfully adjusted for this by incorporating both the movement cost (gCost) and the heuristic (hCost), thereby finding an alternative, less costly path. In contrast, Greedy Best-First Search continued to prioritize the heuristic alone, ignoring the increased movement cost, and therefore failed to adapt.

This comparison illustrates that while Greedy Search is faster and simpler due to its focus on the heuristic, it risks producing suboptimal solutions when traversal costs vary significantly. A*, on the other hand, is more computationally intensive but reliably finds the most cost-efficient path, making it a better choice for environments where costs and obstacles are not uniform.