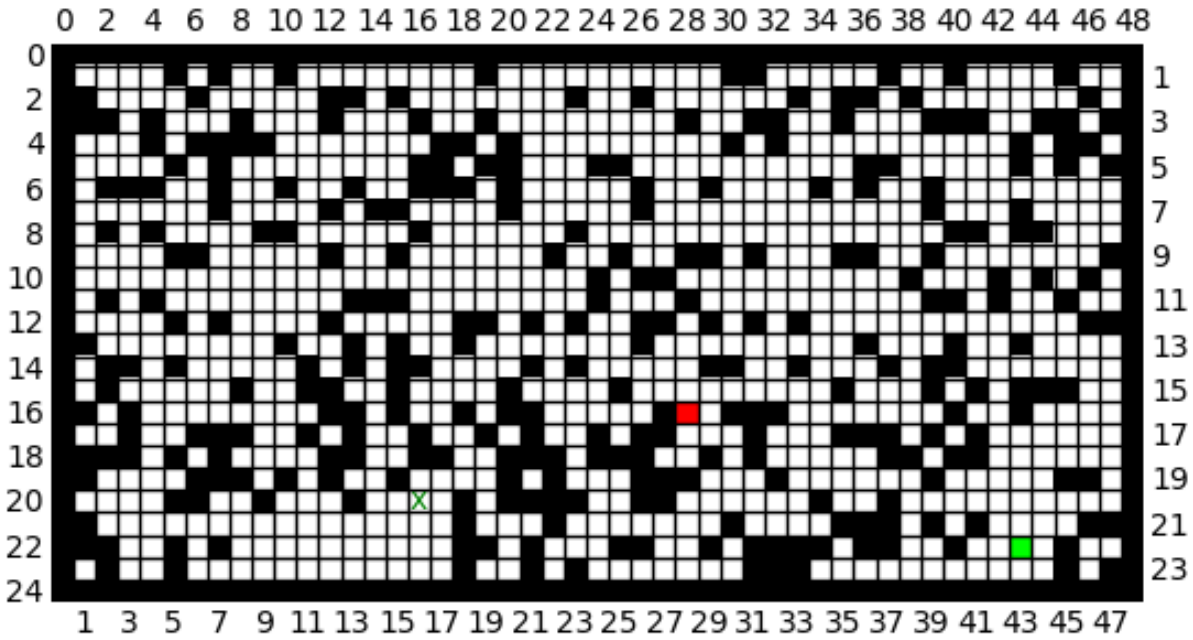
ME/CS/EE 133b Final Project

Fire Finding Rescue Robot

Isabella Pagano, Deepro Pasha, Sara Razavi

March 2025



1 Overview

Our project focuses on developing a fire-fighting robot capable of navigating a grid-based environment, rescuing a dog, and extinguishing a fire, all while avoiding dynamic randomly placed obstacles. The primary challenge is to develop an efficient path-planning algorithm that adapts to obstacles and ensures that the robot reaches its goals (both dog and fire) in the shortest possible time. This project is important as it demonstrates autonomous robotic intervention in dangerous environments.

The primary goal of the robot is to reach a designated fire location determined apriori, while navigating continuously changing obstacles. A secondary goal of the robot is to "rescue" / reach a designated dog survivor before reaching the fire. The robot first works to achieve the secondary goal, then achieve the primary goal and end execution. A grid map is thus utilized as described further in the environment setup.

The specific challenge that is tackled in this project is the implementation of the D* Lite algorithm while enforcing continuously changing obstacles and a secondary objective that needs to be accomplished. This goal is successfully achieved by the project.

2 Approach

To tackle the problem, we implemented the following:

2.1 Environment Setup

- The grid environment is initialized with walls, obstacles, a dog, and a fire.
- Nodes are created for each grid cell and connections (neighbors) are established for valid moves, which can be horizontal, vertical, or diagonal.
- The red node represents the fire node or the goal node, while the green node represents the dog node that needs to be accessed by the robot before it reaches the goal node. The square with the green X represents the robot that traverses the grid. White squares are all squares that the robot can access and traverse on, while black squares are obstacles that the robot cannot travel on.
- For each step taken by the robot, the locations of the obstacles are shuffled, thus continously changing the environment that the robot is traversing.

  The environment setup can be seen in the figure below:

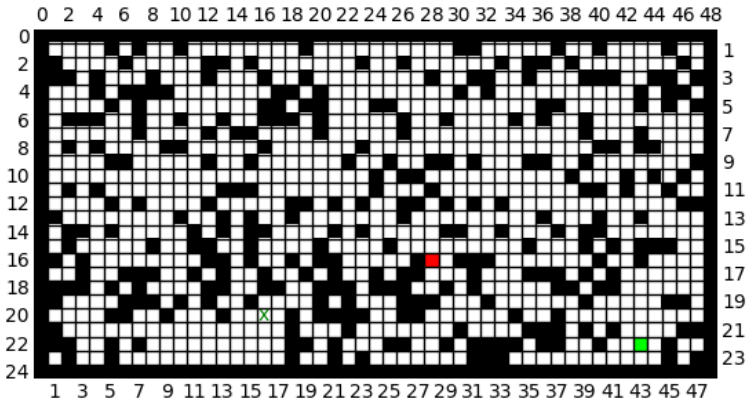


Figure 1: Environment Layout

2.2 Path Planning

- The D* Lite algorithm is used to compute the optimal path from the robot's current position to the dog and then to the fire. While functionally equivalent to the A* algorithm in the calculation of the optimal path, D* Lite differs in that the path can be recomputed with new costs assigned to existing nodes if an unknown obstacle is found on the precomputed optimal path. Given the constantly changing environment condition we applied to the problem, this made D* lite perfect for our usage cases.
- The algorithm starts from the goal node (dog or fire) and works backward to the robot's current position in order to ensure that it is optimal and adaptable to dynamic obstacles.
- Once computed, the algorithm returns a list of nodes which form the optimal path from the robot's current location to its current goal.

2.3 Dynamic Obstacle Handling

- Obstacles are randomly placed in the environment, and the robot must be able to recalculate its path when it encounters an obstacle. This can be seen in Figure 2.

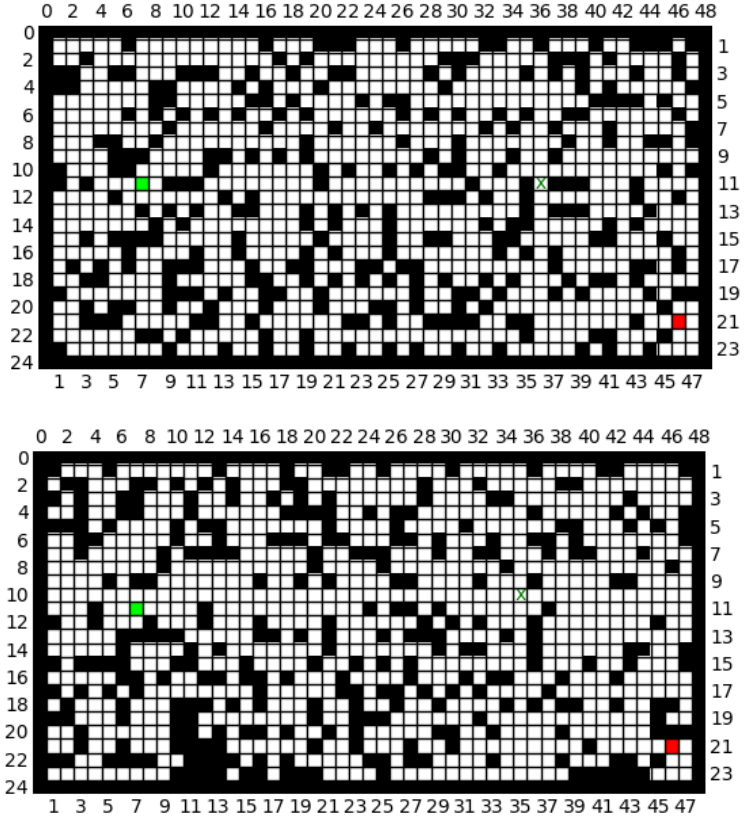


Figure 2: Randomly Placed Obstacles in the Same Round

- The algorithm updates the grid and recomputes the path it takes using the D* Lite approach. When the robot encounters an obstacle node, it triggers a reset of the available nodes in the map accordingly and then re-runs the path planning algorithm to find a new valid path to the goal.

2.4 Robot Movement

- The robot moves along the computed path to reach its defined goal. As the robot traverses the path, it takes single-step moves at a time, allowing it to check for collisions with unknown obstacles.

- The position of the robot is updated after each move, and the environment is re-evaluated for new obstacles prior to taking a step.

2.5 Visualization

- A visualization tool is used to display the grid, robot, obstacles, dog, and fire for each step taken by the robot towards its object. The output is described in the Environment setup.
- The robot's path and progress are visually tracked in order to show the robot's behavior and performance. The square X representing the robot moves through grid squares for each iteration, which allows for easy tracking. This can be seen in the figure below.

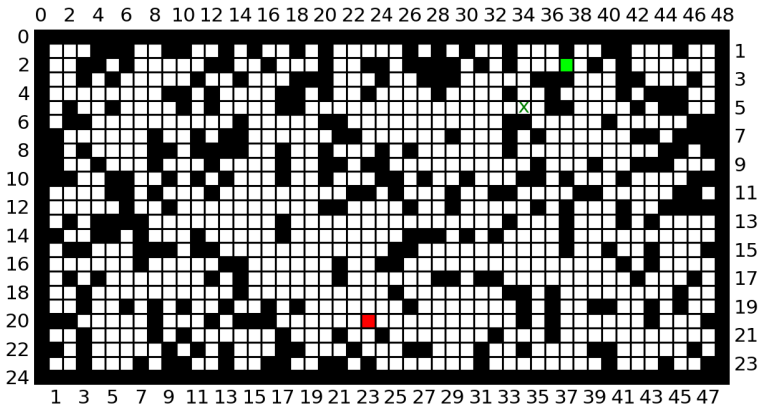


Figure 3: Square X Representation of the Robot at (34, 5)

Altogether, this approach allowed for the real-world scenario of fire seeking to be easily modeled. In addition, D* lite was able to be applied to this grid map visualization quite readily. Given the applicability of D* lite to this particular scenario, this combination was the best suited approach for this project.

3 Technical Details

3.1 Implementation

This was implemented in Python 3 and reuses the grid environment utilized in HW 5. However, most utilities presented in HW 5 have been sufficiently replaced with what is necessary for this program to run. The code is available at https://github.com/TerminalVelocityDPro/133b_Winter_2025.

3.2 Algorithm

The core of this project is the D* Lite algorithm, which is implemented as follows:

3.2.1 Node Initialization

- Each node in the grid is initialized with the following instance attributes:

  Row and Col: These store the row and column where the node is located on the grid, allowing the position of each node to be known.

Type: The node can be 'clear', 'obstacle', 'fire', or 'dog'; this allows the goal nodes to be found and for obstacle checking to occur. All nodes are initialized to 'clear' and their type is reset by various helper functions.

Neighbors: a list of all valid, clear nodes connected to the node itself; this is initialized as an empty list.

Parent: the neighbor node which connects the node to the search tree leading to the goal; this is initialized to None.

cost2Reach: this tracks the actual cost to reach the node from the goal. This is initialized to infinity.

cost: each node is defined by D* Lite to have a cost which is the sum of the cost to reach that node from the goal and the estimated cost to go until the starting node is reached. This is initialized to infinity.

Seen and Done: both are Booleans which track if the node has been processed by the D* Lite algorithm during path planning. Seen indicates whether a node has been added to the queue already or not, and done indicated when a node has been fully processed. Both are initialized to False.

- The cost function is the sum of the actual cost to reach the node from the goal (i.e. the number of steps to reach it) and the estimated cost to reach the starting node. The estimated cost is defined by the Chebyshev distance, as the robot is allowed to make diagonal moves and thus the Chebyshev distance most closely approximates the minimum number of moves the robot would make to travel to the start node from the current node. It should be noted that in convention, D* Lite starts the tree at the goal node and branches out until the starting point is found, the inverse of A* and Dijkstra's algorithms.

3.2.2 Path Computation

- The algorithm starts at the goal node and finds neighboring nodes. Then, it updates their costs and parent pointers.
- Nodes are sorted by their total cost in order to find the most promising path. Sorting by cost ensures that those most promising paths are explored first, as they are estimated to be "lowest cost" options.
- The algorithm finishes when the robot's current position is reached. Then, the path is reconstructed by following parent pointers from the start back to the goal node.

3.2.3 Dynamic Updates

- When the robot encounters an obstacle, the affected node is marked as an obstacle, and the neighbors are updated to remove the connections from the affected node to its neighbors, effectively eliminating it from all future possible paths.
- Each time an obstacle is encountered, the algorithm finds and recomputes a new path from the robot's current position to the goal to make it adaptable to a dynamic environment. This new path takes into account the now-invalid node that was previously unknown to be blocked.
- In addition, the map itself also dynamically updates, as obstacles are completely shuffled during each iteration of the robot's path-finding algorithm.

3.2.4 D* Lite Algorithm Pseudo-code

- Input:

  - `current`: The current node (starting point)
  - `goal`: The goal node
  - `nodes`: A list of all nodes in the graph
- Output:
  - A path from the `current` node to the `goal` node.

- Initialization

```
Initialize all nodes:
    For each node in nodes:
        node.seen = False
        node.done = False
        node.cost = infinity
        node.cost2Reach = infinity
        node.parent = None

Set up the goal node as the starting point for the backward search:
    goal.seen = True
    goal.cost = distance(goal, current)
    goal.cost2Reach = 0
    goal.parent = None

Initialize the priority queue (onDeck) with the goal node:
    onDeck = [goal]
```

- Main Loop

```
While the priority queue (onDeck) is not empty:
    If the current node is reached:
        Break the loop (path found).

    Pop the node with the lowest cost from the priority queue:
        node = pop(onDeck)

    For each neighbor of the current node:
        If the neighbor is already marked as "done":
            Skip this neighbor.

        Calculate the move cost:
            If diagonal move:
                move_cost = sqrt(2)
            Else:
                move_cost = 1

        Compute the temporary cost to reach the neighbor:
            tempCost2Reach = node.cost2Reach + move_cost

        Compute the heuristic cost to go from the neighbor to the current node:
            cost2Go = distance(neighbor, current)

        Compute the total estimated cost:
            totalCost = tempCost2Reach + cost2Go

        If this total cost is better than the neighbor's current cost:
            Update the neighbor's cost2Reach and total cost:
                neighbor.cost2Reach = tempCost2Reach
```

```
                neighbor.cost = totalCost

            Set the neighbor's parent to the current node:
                neighbor.parent = node

            If the neighbor is already seen:
                Remove the neighbor from the priority queue.
            Else:
                    Mark the neighbor as seen:
                    neighbor.seen = True

            Insert the neighbor back into the priority queue, maintaining order:
                insert(onDeck, neighbor)

        Mark the current node as done:
            node.done = True
```

- Path Reconstruction

```
    Reconstruct the path from the current node to the goal:
        Initialize an empty path list.
        Start from the current node's parent:
            brick = current.parent

        While brick is not None:
            Append brick to the path list.
            Move to the next parent:
                brick = brick.parent

        Return the reversed path (from current to goal).
```

3.3 Key Equations and Parameters

3.3.1 Chebyshev Distance

This was used as the heuristic for estimating the cost of the goal:

$$\text{distance}(n_1, n_2) = \max \text{distance}(n_1, n_2) = \max\left(|n_1.\text{row} - n_2.\text{row}|, |n_1.\text{col} - n_2.\text{col}|\right) \tag{1}$$

3.3.2 Cost Function

The total cost for a node is the sum of the cost to reach the node and the estimated cost to reach the goal:

$$\text{totalCost} = \text{cost2Reach} + \text{cost2Go} \tag{2}$$

3.3.3 Move Cost

- Diagonal moves have a cost of $\sqrt{2}$, and horizontal/vertical moves have a cost of 1. The cost2Reach of a node is defined as the sum of the move cost of each step taken from the goal to arrive at that node on the optimal path.

3.3.4 Relevant Helper Functions

Several helper functions were written in the mapping code in order to simplify the main loop and keep track of logic needed for value resetting. These functions are listed and described below:

- computeCurrentNode(robot,nodes): This function finds which node the robot is located at by looping through the list of nodes; it is used to update the current node for path replanning.
- checkObstacle(obstacles, node): This function checks if a node is now blocked, so the neighbors can be updated and a new path can be planned; it returns True if blocked and False if clear.
- setObstacle(node): This function sets the node type to 'obstacle' and clears the node's neighbors so it can be planned around.
- clearObstacle(node, nodes): This function checks if a node which was blocked is now clear again, and then regenerates its neighbors and resets the node type to 'clear'; it returns True is an obstacle was cleared and False if none were cleared, in order to check for path replanning.
- setFireNodes(goalmark, nodes) and setDogNodes(dogmark, nodes): These functions set the node type at the fire and dog location to 'fire' and 'dog', respectively.
- connectNodes(nodes, walls): This function sets the valid neighbors of all the nodes in the map in order to run the D* Lite algorithm.
- setUpObstacles(walls): This function initializes the random obstacles and returns a list of coordinates where each blocked node exists.
- setGoal(nodes): This function sets the goal node to first the dog node and then the fire node, if the dog has already been found; it returns the goal node, or returns nothing if no goal was found.

4 Contributions & Lessons Learned

4.1 Key Valuable Contributions

- D* Lite Implementation: Successfully implemented the D* Lite algorithm for dynamic path planning in a grid-based environment.
- Dynamic Obstacle Handling: Implemented dynamic obstacles, developed a set of checks for obstacle detection, and developed procedures for adapting to them in real time.
- Visualization: Successfully adapted a visualization class to track the robot's progress and environment changes to maintain continuity across dynamic environment changes.

4.2 Lessons Learned

- One key lesson learned was the importance of ensuring that information is tracked properly throughout the program. When issues were presented with the visualization not matching what the path was supposed to be, these issues mostly occurred due to the information in the visualization not matching the path information. This was particularly important when tracking down issues in path computation, which ended up being tied to how information was stored in the Visualization class and passed on the Robot when initialized. In doing so, another important lesson was ensuring that any heritage code is fully understood and that any additions to said code should be done with special caution.
- Another key lesson learned was regarding how adaptive the D* Lite algorithm was to finding obstacles it could collide with. Compared to other algorithms we previously saw during term such as A*, D* Lite was far better able to contend with changing obstacles while still ensuring an optimized path of traversal. This is of course mostly due to how D* Lite is designed to specifically

handle such a scenario, but we learned a lot about how these variable environmental situations can be handled in terms of path finding by implementing the D* Lite algorithm for ourselves.

4.3 Sensitivity to Parameters

- We learned that the algorithm is sensitive to the choice of heuristic (Chebyshev distance) and move costs. Using different move costs would impact the algorithm's definition of an "optimal" path, as well as impact the map's consistency and the order in which the algorithm processes nodes for path-finding. This is similar to previous implementations done in class, such as the A star implementation, though this is of course done with consideration to D* lite's particularly valuable ability to recalculate the path.
- Diagonal moves allow for more efficient paths, however, they increase computational complexity. Given the quantity of obstacles and the rapidity with which they change, allowing more possible moves seemed reasonable, as well as a better model of the "real" world within a grid. Properly modeling moves was also something that we considered before in class with regard to certain algorithm implementations of A* and similar, but our approach worked to directly model real-life actions in a very applied sense.
- The algorithm's performance depends on grid size and the number of obstacles, which affects the complexity of the pathfinding approaching by increasing the number of neighbors that needs to be considered and the number of reconsiderations that need to be made when encountering an obstacle. This is again similar to other implementations of pathfinding algorithms done in class before, but D* lite was particularly complex regarding its pathfinding and recomputation, thus giving us more experience with more complex path-finding approaches.

5 Conclusion

This project illustrates the effectiveness of the D* Lite algorithm for dynamic path planning in a fire-fighting robot scenario. The robot successfully navigates a dynamic environment while also rescuing a dog and extinguishing the fire while avoiding obstacles.

In the future, we want to test the algorithm on larger grids and more complex environments. Additionally, we want to incorporate sensor data and real-time obstacle detection to integrate this into the real world. Additionally, we might want to look into other alternative heuristics and algorithms to optimize our work.

Overall, this was a very interesting algorithm to implement, with a lot of nuance in implementation. However, its power in recomputing paths was very much demonstrated throughout our results, which we were very happy and interested to see. For others trying something similar, understanding the D* lite algorithm is critical to successfully writing and implementing the above program - this is something we ourselves focused on and we believe that this is critical to the success of the robot when it comes to its ability to adapt its path when facing an unknown situation. Understanding the program allowed us to better implement it at first and then accordingly debug it by comparing our program to the proper pseudocode and the intended results of the algorithm. True understanding is critical to implementing this type of path finding robot.

This also varied quite a lot from the homework assignments we did in class as this implementation of D* lite, although similar to A star, had the added nuance of recalculating the path when facing an unknown obstacle. This was quite different from what we had engaged with before, but ultimately, it was a very educational experience.

Thank you again!