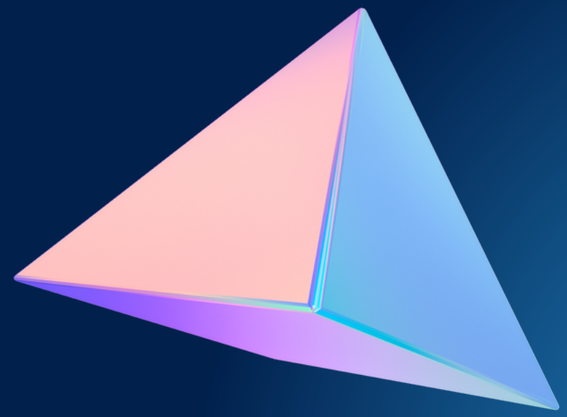
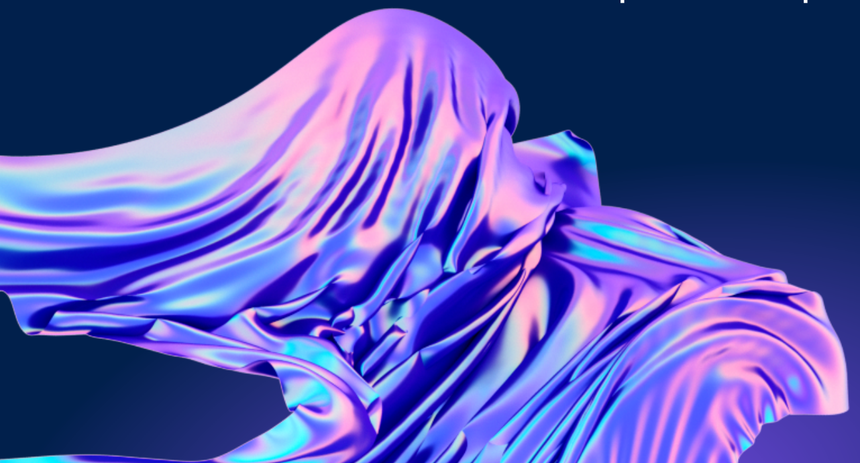
Path Planning project

Presentation

objective



The objective of this project is to design and implement a path planning algorithm that enables a robotic system to navigate efficiently and safely through an environment while avoiding obstacles dynamically. The algorithm will utilize data structures like graphs and priority queues to determine the optimal path from a starting point to a goal location.



Project Scope

- Graph Generation: Create a graph representation of the environment, where each node represents a potential position for the robot.
- Obstacle Detection: Implement obstacle detection mechanisms to identify and mark obstacles in the environment.
- Path Planning Algorithm: Design a path planning algorithm that utilizes the graph and prioritizes paths based on various criteria such as distance, cost, and obstacle avoidance.

Program Structure and Components

- **Cell Structure:** The **Cell** structure represents a cell in the grid. It contains information about the cell's row and column indices, distance from the source cell, and a pointer to its parent cell.
- **Distance Calculation:** The **calculateDistance** function calculates the Manhattan distance between two cells using their row and column indices.
- **Path Reconstruction:** The **reconstructPath** function reconstructs the path from the destination cell to the source cell by traversing the parent pointers. It marks the cells on the path as **2** in the grid.
- **Dijkstra's Algorithm:** The **findPath** function implements Dijkstra's algorithm to find the shortest path from the source cell to the destination cell. It uses a priority queue to store the cells being explored, prioritizing cells with shorter distances. The algorithm iteratively explores neighboring cells, updates their distances, and terminates when the destination cell is reached or all reachable cells have been explored.

Input and Output

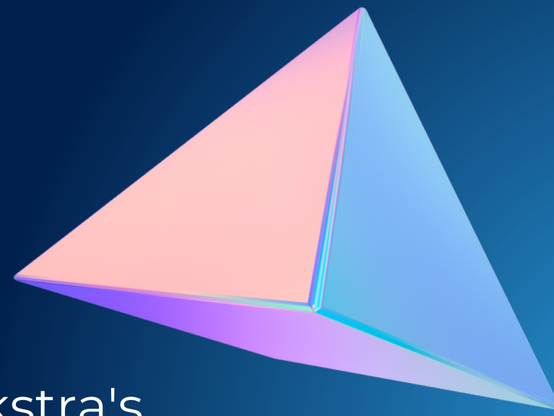
```
Obstacle Detection Environment:
. . X . . . X . . .
. . . . . . . . . .
. X X . X X . X X .
. X . . . . . . . .
. . . X . . . X . .
. . . X . . . X . .
. . . X . . . X . .
. . . X . . . X . .
Enter source coordinates (row col): 0 0
Enter destination coordinates (row col): 5 5
Path: Grid with Path:
P X X . . . X . . .
P P P P . . . . . .
P X X P X X . X X .
. X . P P . . . . .
. . . X P P . X . .
. . . X . P . X . .
. . . X . . . X . .
. . . X . . . X . .

--------------------------------
Process exited after 12.97 seconds with return value 0
Press any key to continue . . .
```

- The Grid is initialized with obstacles represented by **1** and empty cells represented by **0**.
- The program prints the obstacle detection environment grid to display the initial setup.
- After finding the shortest path using Dijkstra's algorithm, the program prints the updated grid with the path marked as **P**.
- If the source or destination coordinates are invalid (outside the grid boundaries), an appropriate error message is displayed.

Limitations and Possible Improvements

- The program assumes a fixed grid size of 8 rows and 10 columns. Modifying the program to handle dynamically-sized grids could enhance its flexibility.
- Currently, the program only considers horizontal and vertical movement between neighboring cells. Including diagonal movement could provide more path options.
- The program does not account for obstacles that may change dynamically during runtime. Adding support for dynamically updating obstacles could make the program more interactive and practical.

Conclusion



in conclusion, the program successfully implements Dijkstra's algorithm to find the shortest path between a source and destination cell in a grid. It effectively utilizes a priority queue and various supporting functions to calculate distances, check cell validity, and reconstruct the path. With further enhancements, such as handling dynamic obstacles and grid size flexibility, the program could be more versatile and adaptable to real-world scenarios.