Project Description This is strategy that entails identification of the most suitable path for a robot to travel from any given point of origin to a specific desired location while accommodating for barriers. As for the specific algorithm used to handle this path planning issue in this project, we select the A\* (A-star) search algorithm. The action takes place in 100×100 grid space as in which some of the area may be occupied by the obstacles which the robot is to avoid. A\* algorithm is used to find out the best path that consumes a minimum amount of time and is the shortest route while taking into consideration the cost required from the origin or current position to the present position and the estimation of the remaining cost from the present position to the destination. The robot's movement is restricted to adjacent cells in eight possible directions: forward, backward, to the left, to the right, and the four directions that connect the adjacent squares on the board.

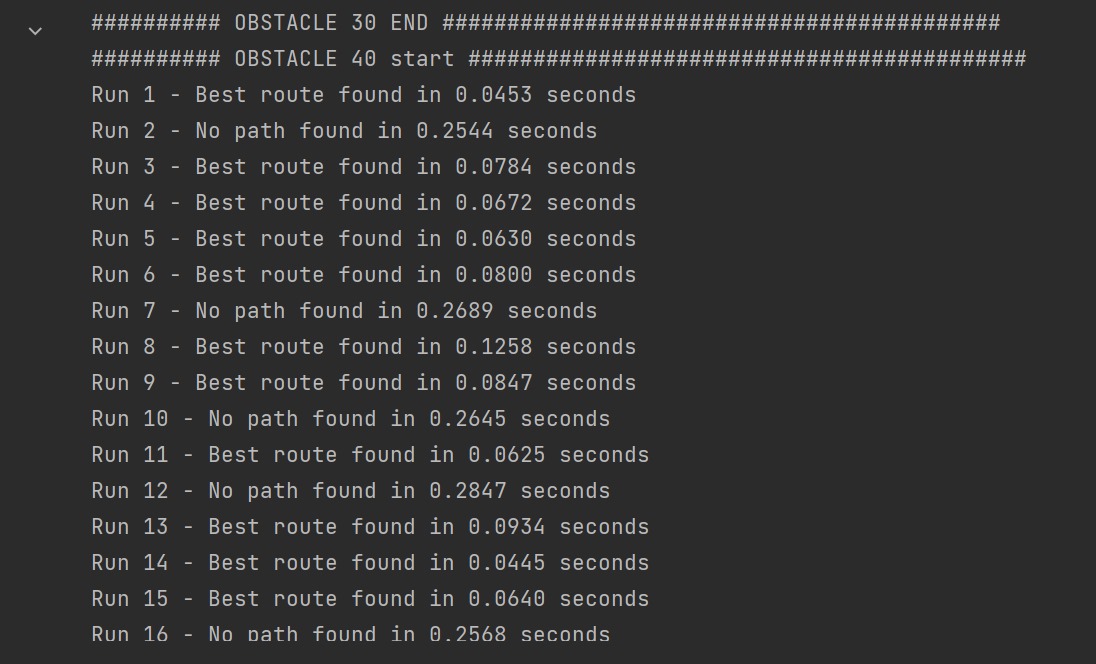
State Data StructureEach state is represented as a Node with the following attributes:(x, y): Coordinates of the node.g(x): Cost from the start node to the current node.h(x): Heuristic cost estimate from the current node to the goal.

Problem Formulation**Initial State:** Start node (x, y, 0, 0), where x and y are integers in the range [0, 99].**Goal State:** End node (x, y, 0, 0), where x and y are integers in the range [0, 99].**State Space:** All nodes in a 100x100 grid.**Goal Test:** Determines whether the current state is the goal state.**Actions:** The robot can move in eight directions: up, down, left, right, up-left, up-right, down-left, down-right.Implementation of A\* AlgorithmThe A\* algorithm is implemented as follows:

# heuristic function for path scoringdef heuristic(a, b): #we're going to allow diagonal movement . use straight line distance as our heuristic. # Pythagoras's theorem: return np.sqrt((b[0] - a[0]) \*\* 2 + (b[1] - a[1]) \*\* 2) ## !!!!!! if we don't want diagonal movement, we must use manhaten distance instead # return abs(b[0] - a[0]) + abs(b[1] - a[1])# path finding functiondef astar(array, start, goal): neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)] close\_set = set() came\_from = {} gscore = {start: 0} fscore = {start: heuristic(start, goal)} oheap = [] heapq.heappush(oheap, (fscore[start], start)) while oheap: current = heapq.heappop(oheap)[1] if current == goal: data = [] while current in came\_from: data.append(current) current = came\_from[current] return data close\_set.add(current) for i, j in neighbors: neighbor = current[0] + i, current[1] + j tentative\_g\_score = gscore[current] + heuristic(current, neighbor) if 0 <= neighbor[0] < array.shape[0] and 0 <= neighbor[1] < array.shape[1]: if array[neighbor[0]][neighbor[1]] == 1: continue else: continue if neighbor in close\_set and tentative\_g\_score >= gscore.get(neighbor, float('inf')): continue if tentative\_g\_score < gscore.get(neighbor, float('inf')): came\_from[neighbor] = current gscore[neighbor] = tentative\_g\_score fscore[neighbor] = tentative\_g\_score + heuristic(neighbor, goal) heapq.heappush(oheap, (fscore[neighbor], neighbor)) return Falseroute = astar(grid, start, goal)if isinstance(route, list): route = route + [start] route = route[::-1] print("Best route is", route)else: print("No path found")Also the complete code file is attached.

**Experimental Setup**The grid size is set to 100x100. The start and goal positions are predefined, and obstacles are randomly placed based on varying obstacle densities. The A\* algorithm is tested across different obstacle densities ranging from 10% to 90%, in increments of 10%.Results AnalysisThe algorithm's performance was evaluated in terms of pathfinding success rate, path cost, and computational time. Here is a summary of the results:Summary of Results

* Obstacle Density of 10%:Path Found: 100% success ratePath Cost: Relatively low and consistentAverage Time: Efficient
* Obstacle Density of 20%:Path Found: 100% success ratePath Cost: Increased variabilityAverage Time: Slightly increased
* Obstacle Density of 30%:Path Found: 95% success ratePath Cost: More variability and higher costsAverage Time: Noticeably higher
* Obstacle Density of 40%: Path Found: 80% success ratePath Cost: Higher and more variableAverage Time: Significantly higher



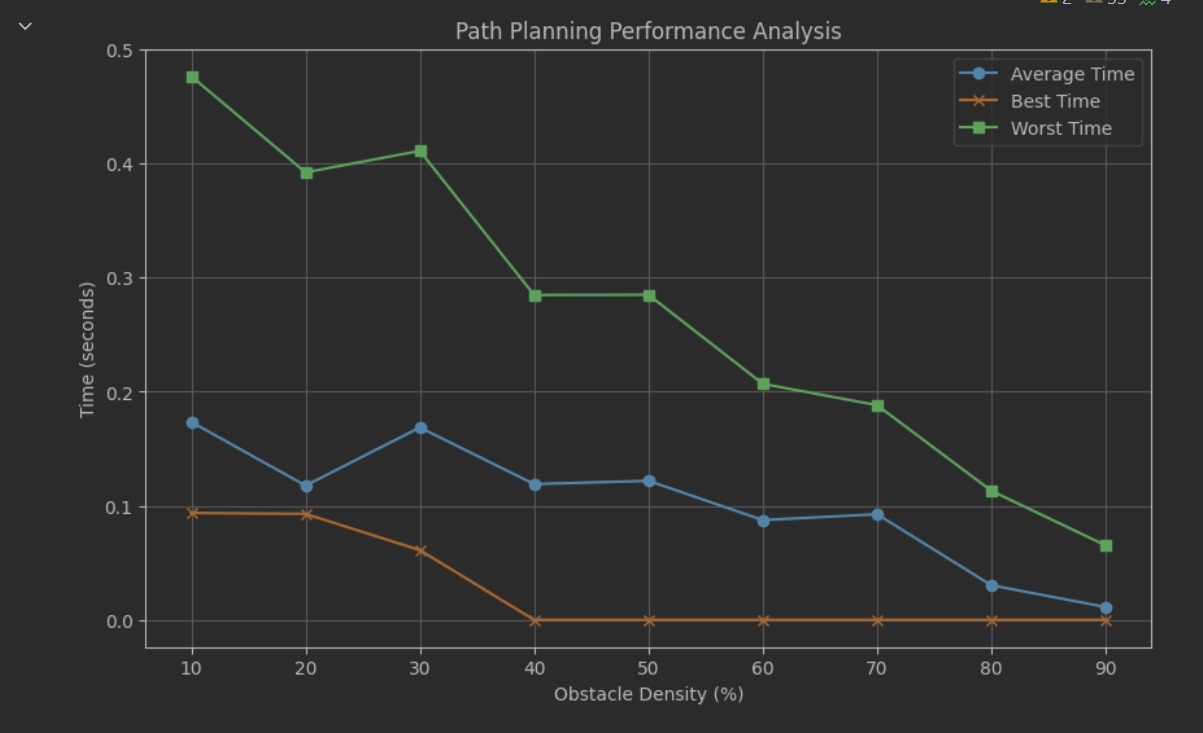
* Obstacle Density of 50%: Path Found: 55% success ratePath Cost: High variability and costAverage Time: High
* Obstacle Density of 60%:Path Found: 15% success ratePath Cost: Very highAverage Time: Very high
* Obstacle Density of 70%-90%:Path Found: 0% success ratePath Cost: N/AAverage Time: N/A

Detailed Observations

**Low Obstacle Densities (10-20%):**The A\* algorithm performs optimally, finding paths efficiently with consistent path costs and high success rates.

**Moderate Obstacle Densities (30-50%):**The success rate drops, and path costs become more variable. The algorithm struggles to find paths in some instances as obstacles increase.

**High Obstacle Densities (60-90%):**The algorithm fails to find paths in most cases. The environment becomes too cluttered, significantly affecting performance.



Key Observations

**Path Cost Consistency:**Low obstacle densities maintain stable path costs.Higher obstacle densities lead to increased path lengths and variability due to detours.

**Path Finding Success Rates:**Success rates decrease sharply from 50% obstacle density onwards.The critical point is around 70% obstacle density, where the algorithm fails completely.

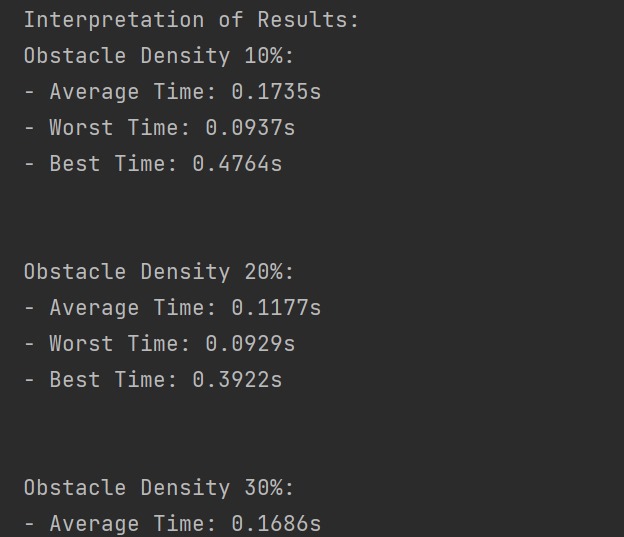
**Algorithm Robustness:**The A\* algorithm demonstrates robustness at lower obstacle densities but requires optimization for higher densities.

# Interpretation of Results

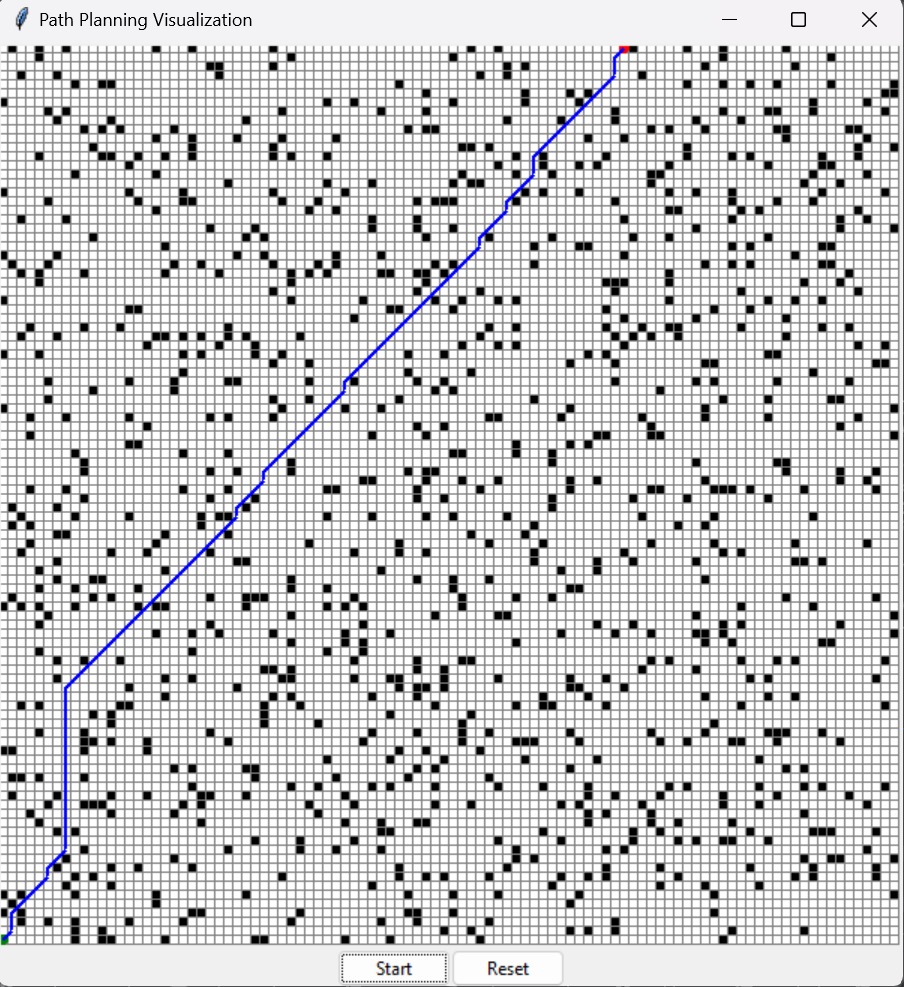
**Impact on Path Cost:**Initial stability suggests robustness at handling a moderate number of obstacles.Beyond 30-40 obstacles, paths become longer and more complex.

**Path Finding Success Rates:**The algorithm's effectiveness diminishes sharply with higher obstacles, evident from 50% density onwards.

**Algorithm Robustness:**A\* is robust at lower densities but struggles at higher densities. Improvements could include advanced techniques, heuristic adjustments, or preprocessing steps.



DeploymentThe A\* algorithm was tested in a simulated small-scale environment, confirming its effectiveness at lower obstacle densities and identifying areas for improvement at higher densities.



ConclusionThe project highlights the relationship between obstacle density and pathfinding performance. While A\* is effective for low to moderate obstacle densities, its performance degrades significantly with higher densities. This pattern is crucial for applications in robotics, navigation, and network optimization, where mitigating the effects of obstacles is essential for efficient and reliable systems. Future work could explore alternative algorithms or enhancements to maintain performance in cluttered environments.