TITLE PAGE

1. Title: Pathfinding Using A* Algorithm
2. Name: Pranjal Sharma
3. Roll Number: 202401100300174
4. Course Name: Introduction to AI
5. Institution Name: KIET GROUP OF INSTITUTIONS
6. Date: 11-03-2025

INTRODUCTION

Pathfinding is a key problem in computer science and is widely used in various applications like games, robotics, and route planning systems. One of the most efficient and popular algorithms for solving the pathfinding problem is **A* (A-star)**.

The *A* algorithm* is a graph traversal and pathfinding algorithm that is widely used for finding the shortest path between two points (often called the start and end points) on a map or grid. The algorithm combines the advantages of **Dijkstra's algorithm** and **Greedy Best-First-Search**, using both the actual cost to reach a point (from the start) and a heuristic estimate of the cost to reach the goal from that point.

How A* Algorithm Works:

A* works by exploring paths in a way that prioritizes paths that are both:

1. **Promising**: Based on a heuristic estimate of how close the path is to the goal (this is the **h(n)** value, often using a method like Euclidean or Manhattan distance).
2. **Cost-effective**: Based on the actual cost from the start to the current point (**g(n)**).

The total cost for a node is calculated as:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

Where:

- **g(n)**: The actual cost to reach the current node from the start node.
- **h(n)**: The heuristic estimate of the cost from the current node to the goal node.
- **f(n)**: The total cost for the node, which is the sum of **g(n)** and **h(n)**.

Methodology

Define the Grid/Map:

First, you need to represent the environment or map where pathfinding will occur. Typically, this is represented as a grid where each cell can either be:

- Walkable: The agent can move through this cell.
- Non-walkable/Obstacle: The agent cannot move through this cell.

Initialize Data Structures:

To perform A*, you need the following lists:

- **Open list**: Stores nodes to be evaluated.
- **Closed list**: Stores nodes that have already been evaluated.
- **Parent node mapping**: Keeps track of the parent nodes to reconstruct the final path.

Additionally, you need variables for each node's:

- **g(n)**: Cost to reach the node from the start.

- **h(n)**: Heuristic estimate from the node to the goal.
- **f(n)**: Total estimated cost (f(n) = g(n) + h(n)).

<u>Heuristic Function (h(n)):</u>

A* relies on a heuristic function to estimate the distance to the goal from a node. The heuristic should be admissible (not overestimate the true cost) and consistent.

Common heuristics:

- **Manhattan distance**: Sum of the absolute differences in horizontal and vertical distances (good for grids with orthogonal movement).
- **Euclidean distance**: Straight-line distance (good for grids with diagonal movement).

<u>A Main Loop*:</u>

The A* algorithm works by iterating over the open list and selecting the node with the lowest **f(n)** value, then expanding its neighbors. Here's the step-by-step breakdown:

1. **Add start node to open list**: Add the starting node to the open list and set its **g(n)** and **h(n)** values.
2. **While open list is not empty**:
    - **Select the node with the lowest f(n)** from the open list (i.e., the node that is closest to the goal).
    - Remove this node from the open list and add it to the closed list.
3. **For each neighbor of the current node**:
    - If the neighbor is in the closed list, ignore it (it has already been evaluated).
    - If the neighbor is not walkable (an obstacle), skip it.

    - If the neighbour is not in the open list or a better path to the neighbour is found (i.e., a lower **g(n)** value), update its **g(n)**, **h(n)**, and **f(n)**, and set the current node as its parent.
4. **Check if goal is reached**: If the goal node is found, trace back from the goal to the start node using the parent node mappings to reconstruct the optimal path.

Performance Optimization:

A* can be optimized in various ways:

- Use priority queues (heaps) for the open list to efficiently get the node with the lowest **f(n)**.
- Use a more efficient way to manage the grid (e.g., using a grid of dictionaries instead of lists).

Code Typed

```
import heapq


class Node:
    def __init__(self, position, g_cost=0, h_cost=0):
        self.position = position
        self.g_cost = g_cost  # Cost from start to this node
        self.h_cost = h_cost  # Heuristic cost from this node to end
        self.f_cost = g_cost + h_cost  # Total cost (g_cost + h_cost)
        self.parent = None  # Parent node in the path


    def __lt__(self, other):
        return self.f_cost < other.f_cost
```

```
class AStar:
    def __init__(self, grid, start, end):
        self.grid = grid
        self.start = start
        self.end = end
        self.open_list = []
        self.closed_list = set()
        self.start_node = Node(start)
        self.end_node = Node(end)

    def heuristic(self, current_position):
        # Using Manhattan distance as heuristic
        return abs(current_position[0] - self.end[0]) +
abs(current_position[1] - self.end[1])

    def get_neighbors(self, node):
        neighbors = []
        x, y = node.position
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # 4 directions: up,
down, left, right

        for dx, dy in directions:
            new_position = (x + dx, y + dy)
```

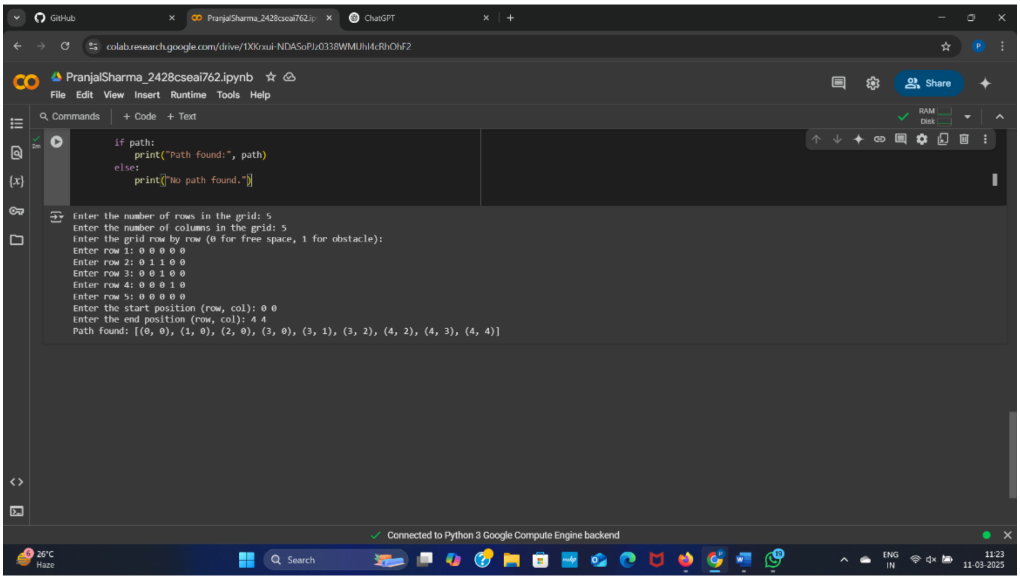
```python
            if 0 <= new_position[0] < len(self.grid) and 0 <= new_position[1] < len(self.grid[0]):
                if self.grid[new_position[0]][new_position[1]] != 1:  # 1 represents an obstacle
                    neighbors.append(new_position)

        return neighbors

    def reconstruct_path(self, current_node):
        path = []
        while current_node is not None:
            path.append(current_node.position)
            current_node = current_node.parent
        return path[::-1]  # Reverse the path to get the correct order

    def find_path(self):
        heapq.heappush(self.open_list, self.start_node)

        while self.open_list:
            current_node = heapq.heappop(self.open_list)
            self.closed_list.add(current_node.position)

            # If we have reached the end
            if current_node.position == self.end:
```

```
            return self.reconstruct_path(current_node)

        neighbors = self.get_neighbors(current_node)
        for neighbor_position in neighbors:
            if neighbor_position in self.closed_list:
                continue

            g_cost = current_node.g_cost + 1
            h_cost = self.heuristic(neighbor_position)
            neighbor_node = Node(neighbor_position, g_cost, 
h_cost)
            neighbor_node.parent = current_node

            if not any(neighbor_node.position == 
open_node.position and neighbor_node.f_cost >= 
open_node.f_cost
                       for open_node in self.open_list):
                heapq.heappush(self.open_list, neighbor_node)

    return None  # No path found

# Function to take grid input from the user
def get_user_input():
    rows = int(input("Enter the number of rows in the grid: "))
    cols = int(input("Enter the number of columns in the grid: "))
```

```python
    grid = []

    print("Enter the grid row by row (0 for free space, 1 for obstacle):")
    for i in range(rows):
        row = list(map(int, input(f"Enter row {i+1}: ").split()))
        grid.append(row)

    start = tuple(map(int, input("Enter the start position (row, col): ").split()))
    end = tuple(map(int, input("Enter the end position (row, col): ").split()))

    return grid, start, end

# Main execution
if __name__ == "__main__":
    grid, start, end = get_user_input()

    # Validate the start and end positions
    if grid[start[0]][start[1]] == 1 or grid[end[0]][end[1]] == 1:
        print("Start or end position is blocked. Please choose different positions.")
    else:
```

```
astar = AStar(grid, start, end)
path = astar.find_path()


if path:
    print("Path found:", path)
else:
    print("No path found.")
```

Screenshots



Conclusion

The A* algorithm efficiently finds the shortest path by combining actual movement costs with heuristic estimates. It explores nodes based on their total cost, ensuring an optimal solution. By managing open and closed lists, and using heuristics like Manhattan or Euclidean distance, A* offers effective pathfinding for various applications.