A* Pathfinding Visualization Project

By: Naman Ravindra Gaonkar

Introduction

This project implements the A* pathfinding algorithm and visualizes its operation on a customizable grid using Python. The purpose is to demonstrate how A* finds the shortest path between two points (start and goal) while navigating around obstacles, with results shown using **matplotlib**. This project is built for educational purposes and to provide insights into search and optimization algorithms.

Objective

- Implement the A* pathfinding algorithm in Python.
- Allow easy configuration of grid size, start/goal points, and obstacle layout.
- Visualize the grid, obstacles, start/goal, and the computed shortest path.
- Demonstrate the algorithm's performance and correctness on sample grids.

Methodology

Tools & Libraries:

- Python 3.x
- matplotlib for visualization
- Standard libraries: **heapq** for efficient priority queues

Implementation Steps

1. Grid Design:
   A Grid class manages the 2D space, obstacles, and valid moves (up, down, left, right).
2. *A Algorithm:**
   An AStar class encapsulates the A* logic, using the Manhattan distance heuristic for grid-based movement. The algorithm tracks open/closed nodes, calculates g and f scores, and reconstructs the optimal path.
3. Visualization:
   The resulting path and grid are displayed with matplotlib, where:
   - White cells are open spaces.
   - Gray cells are obstacles.
   - Blue circle marks the start.
   - Red circle marks the goal.

- 'X's mark the computed path.

4. User Parameters:
   Grid size, start/goal positions, and obstacles can be easily changed in the __main__ block.

CODE: (Implementation Of Code)

```
import heapq
import matplotlib.pyplot as plt

class Grid:
    def __init__(self, width, height, start, goal, obstacles):
        self.width = width
        self.height = height
        self.start = start
        self.goal = goal
        self.obstacles = set(obstacles)
        self.nodes = [[(x, y) for y in range(self.height)] for x in range(self.width)]

    def is_valid(self, node):
        x, y = node
        return 0 <= x < self.width and 0 <= y < self.height and node not in self.obstacles

    def get_neighbors(self, node):
        x, y = node
        neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]
        return [n for n in neighbors if self.is_valid(n)]

class AStar:
    def __init__(self, grid):
        self.grid = grid
        self.start = grid.start
        self.goal = grid.goal

    def manhattan_distance(self, node1, node2):
        return abs(node1[0] - node2[0]) + abs(node1[1] - node2[1])
```

```
    def reconstruct_path(self, came_from, current):
        path = []
        while current in came_from:
            path.append(current)
            current = came_from[current]
        path.append(self.start)
        path.reverse()
        return path

    def find_path(self):
        open_set = []
        heapq.heappush(open_set, (0, self.start))
        came_from = {}
        g_score = {node: float('inf') for row in self.grid.nodes for node in row}
        g_score[self.start] = 0
        f_score = {node: float('inf') for row in self.grid.nodes for node in row}
        f_score[self.start] = self.manhattan_distance(self.start, self.goal)

        while open_set:
            _, current = heapq.heappop(open_set)
            if current == self.goal:
                return self.reconstruct_path(came_from, current)
            for neighbor in self.grid.get_neighbors(current):
                tentative_g = g_score[current] + 1
                if tentative_g < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g
                    f_score[neighbor] = tentative_g + self.manhattan_distance(neighbor, self.goal)
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))
        return None

def visualize_path(grid, path):
    plt.figure(figsize=(8, 8))
    for x in range(grid.width):
```

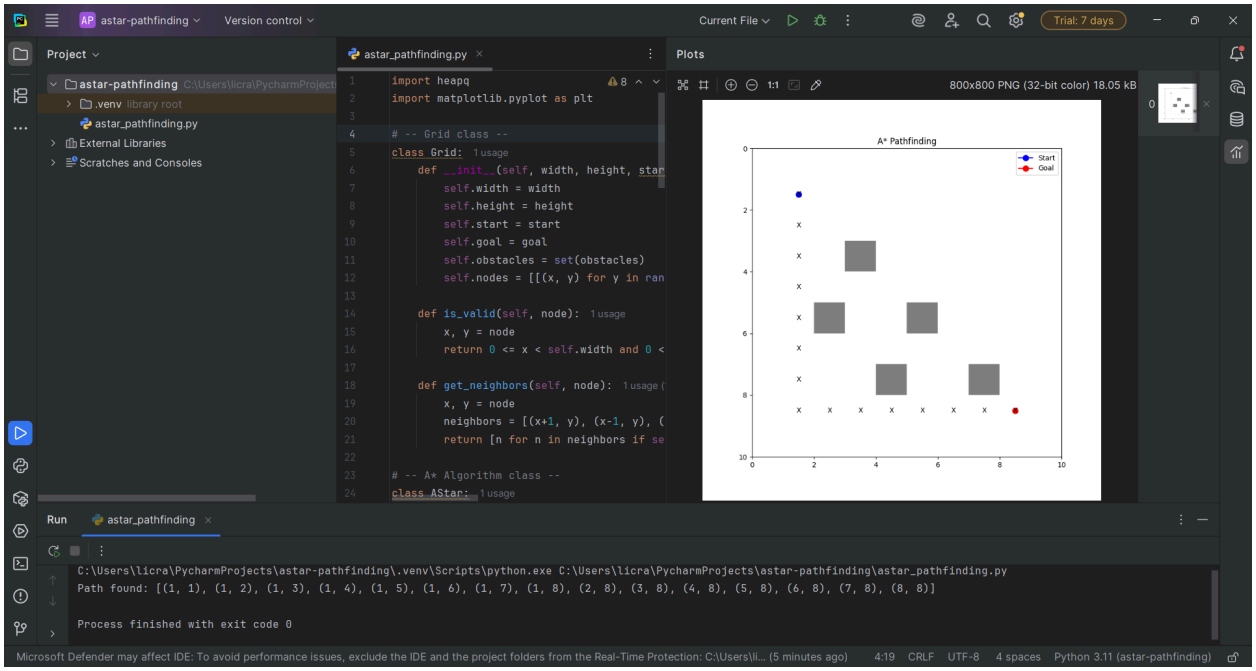
```
        for y in range(grid.height):
            color = 'gray' if (x, y) in grid.obstacles else 'white'
            plt.fill_between([x, x+1], [y, y], [y+1, y+1], color=color)
    if path:
        for node in path:
            plt.text(node[0] + 0.5, node[1] + 0.5, 'X', ha='center', va='center', fontsize=10)
    plt.plot([grid.start[0] + 0.5], [grid.start[1] + 0.5], marker='o', markersize=8, color='blue', label='Start')
    plt.plot([grid.goal[0] + 0.5], [grid.goal[1] + 0.5], marker='o', markersize=8, color='red', label='Goal')
    plt.xlim(0, grid.width)
    plt.ylim(0, grid.height)
    plt.gca().invert_yaxis()
    plt.title('A* Pathfinding')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    start = (1, 1)
    goal = (8, 8)
    obstacles = [(3, 3), (5, 5), (7, 7), (2, 5), (4, 7)]
    grid = Grid(10, 10, start, goal, obstacles)
    astar = AStar(grid)
    path = astar.find_path()
    if path:
        print("Path found:", path)
        visualize_path(grid, path)
    else:
        print("No path found.")
```

Explanation:

- The algorithm uses a min-heap as a priority queue for efficient node expansion.
- The path is reconstructed by backtracking from goal to start using the `came_from` dictionary.

- Obstacles, grid size, and coordinates are fully configurable



Results and Observations

Sample Output:

- When run, the algorithm identifies the shortest path from the start to the goal, avoiding all obstacles.
- The output plot visually represents:
    - The layout of the grid and obstacles.
    - The path found (if one exists), marked with 'X'.
    - Start and goal nodes (colored).
- If no path exists, the program notifies accordingly.

Example:
For a 10x10 grid, start at (1,1), goal at (8,8), with obstacles at several points, the algorithm computes and plots the optimal path, as shown in the attached screenshot/evidence (add a screenshot of your plot here for your report!).

Conclusion

This project successfully demonstrates the application and visualization of the A* pathfinding algorithm in Python. Through intuitive grid visualization and user-set obstacles, it provides hands-on understanding of heuristic search, efficient pathfinding, and algorithmic problem-solving. The modular design enables easy extension (e.g., to diagonal moves or weighted grids). The results confirm the correctness and utility of the A* algorithm for grid-based navigation problems.