INSYDE.IO

Step 1: Generating a Synthetic City Map

1.1 What is a Synthetic City Map?

A **city map** is represented as an **N×N grid**, where each cell represents:

- 0 → **Open road (valid path)**
- 1 → **Obstacle (building, river, restricted area)**
- 2 → **Start point (entry to city/road)**
- 3 → **End point (destination/exit point)**

This allows us to **simulate a road layout with obstacles and find an optimized path using AI**.

1.2 How We Generate the Grid in Python?

- We define the **grid size** (N×N matrix).
- We **randomly assign obstacles (1s) using NumPy**, ensuring 70% roads and 30% obstacles.
- We place a **start point (2) in the top-left** and an **end point (3) in the bottom-right**.

📌 **Code for Generating the City Map:**

```
import numpy as np

import matplotlib.pyplot as plt


def generate_city_map(grid_size=10, obstacle_prob=0.3):

    np.random.seed(42)  # Fix randomness for reproducibility
```

```
    city_map = np.random.choice([0, 1], size=(grid_size, grid_size), p=[1-obstacle_prob, obstacle_prob])
    city_map[0][0] = 2  # Start (S)
    city_map[grid_size-1][grid_size-1] = 3  # End (E)
    return city_map


grid_size = 10
city_map = generate_city_map(grid_size)


# Displaying the grid
plt.imshow(city_map, cmap="coolwarm", origin="upper")
plt.title("Generated City Map")
plt.show()
```

1.3 Explanation of the Code

✅ **np.random.choice([0, 1], size=(grid_size, grid_size), p=[0.7, 0.3])** → Generates a **random city map** with 70% roads and 30% obstacles.
✅ **city_map[0][0] = 2** → Marks the **start point** at the top-left.
✅ **city_map[grid_size-1][grid_size-1] = 3** → Marks the **end point** at the bottom-right.
✅ **Matplotlib is used to visualize the grid**, where **blue cells are roads, red cells are obstacles**.

🛠 *Step 2: Selecting A Search for Path Optimization**

*2.1 Why A Search?**

*A Search** is one of the most **efficient pathfinding algorithms** used in **Google Maps, GPS systems, and robotics**. It works by:

1. Exploring paths **based on cost (distance traveled) and heuristics (estimated distance to goal)**.
2. Always choosing the **best path towards the goal** while avoiding obstacles.

*2.2 How A Search Works?**

- Each cell has:
    - G(x): **Cost from start to current cell**.
    - H(x): **Estimated cost from current cell to goal (Manhattan distance)**.
    - F(x): **Total cost →** F(x)=G(x)+H(x)F(x) = G(x) + H(x)F(x)=G(x)+H(x)
- The algorithm **expands the lowest-cost path first** until it reaches the

Goal

*🛠 Step 3: Implementing A Search Algorithm**

```
import heapq
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right


def heuristic(a, b):
    """Calculate Manhattan distance heuristic."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


def a_star_search(city_map, start, end):
    """Find the shortest path using A* Search."""
    grid_size = city_map.shape[0]
    open_list = []
```

```
    heapq.heappush(open_list, (0, start))

    came_from = {start: None}
    g_score = {pos: float("inf") for pos in np.ndindex(city_map.shape)}
    g_score[start] = 0

    f_score = {pos: float("inf") for pos in np.ndindex(city_map.shape)}
    f_score[start] = heuristic(start, end)

    while open_list:
        _, current = heapq.heappop(open_list)

        if current == end:
            path = []
            while current:
                path.append(current)
                current = came_from[current]
            return path[::-1]

        for d in DIRECTIONS:
            neighbor = (current[0] + d[0], current[1] + d[1])
            if (0 <= neighbor[0] < grid_size and 0 <= neighbor[1] < grid_size and
city_map[neighbor] != 1):
                tentative_g_score = g_score[current] + 1
                if tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
```

```
            f_score[neighbor] = tentative_g_score + heuristic(neighbor, end)
            heapq.heappush(open_list, (f_score[neighbor], neighbor))

    return None  # No path found

# Running A* Search
start = (0, 0)
end = (grid_size - 1, grid_size - 1)
optimal_path = a_star_search(city_map, start, end)
```

3.1 Explanation of the Code

✅ **Manhattan distance (heuristic)** guides the algorithm.
✅ **heapq.heappush(open_list, (f_score[start], start))** → Uses a priority queue to process the lowest-cost node first.
✅ **The algorithm reconstructs the optimal path and returns it.**

🛠 Step 5: Visualizing the Optimized Road Network

```
from google.colab.patches import cv2_imshow  # Colab fix for cv2.imshow()
import cv2

def visualize_city_map(city_map, optimal_path, cell_size=50):
    grid_size = city_map.shape[0]
    img = np.ones((grid_size * cell_size, grid_size * cell_size, 3), dtype=np.uint8) * 255

    # Draw grid and obstacles
```

```
    for i in range(grid_size):
        for j in range(grid_size):
            color = (255, 255, 255)  # White for roads
            if city_map[i][j] == 1:
                color = (0, 0, 0)  # Black for obstacles
            elif (i, j) == start:
                color = (0, 255, 0)  # Green for start
            elif (i, j) == end:
                color = (0, 0, 255)  # Red for end

            cv2.rectangle(img, (j * cell_size, i * cell_size),
                          ((j + 1) * cell_size, (i + 1) * cell_size), color, -1)

    # Draw optimal path
    if optimal_path:
        for (i, j) in optimal_path:
            cv2.circle(img, (j * cell_size + cell_size // 2, i * cell_size + cell_size // 2),
                       10, (0, 255, 255), -1)

    # Display image in Colab
    cv2_imshow(img)

# Call visualization
visualize_city_map(city_map, optimal_path)
```

5.1 Explanation

✅ **Obstacles are drawn in black**, roads in white.

✅ **Start (green), End (red), Path (yellow dots)** are marked.

OUTPUT: