3. Heuristic Search: A Algorithm for Route Optimization

Use Case: Design a navigation system that finds the shortest and fastest route for a delivery truck in a city.

Objective: Implement the A* algorithm to find the optimal path from a starting location to a destination, considering both distance and traffic conditions.

Heuristics: Use the Manhattan or Euclidean distance for evaluation.

Step 1: Import Libraries

Start by loading essential libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import heapq  # For priority queue implementation
```

Step 2: Define the City Grid and Cost Function

Represent the city as a 2D grid, where each cell represents a road segment.

Define two matrices: one for distances and another for traffic congestion. For simplicity, let's use random values to simulate varying traffic and distance values.

```
grid_size = (10, 10)  # Adjust grid size as needed
distance_matrix = np.random.randint(1, 10, size=grid_size)
traffic_matrix = np.random.randint(1, 5, size=grid_size)
```

Step 3: Define Heuristic Functions

Implement Manhattan and Euclidean heuristics.

```
def manhattan_distance(start, end):
    return abs(start[0] - end[0]) + abs(start[1] - end[1])

def euclidean_distance(start, end):
    return np.sqrt((start[0] - end[0])**2 + (start[1] - end[1])**2)
```

Step 4: Implement the A* Algorithm

Create the A* algorithm that calculates the cost of reaching a neighboring cell based on distance and traffic matrices. Use a priority queue to prioritize nodes with the lowest estimated cost.

```
def a_star_algorithm(start, end, distance_matrix, traffic_matrix, heuristic_func):
    rows, cols = distance_matrix.shape
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    cost_so_far = {start: 0}
```

```
    while open_set:
        current_cost, current = heapq.heappop(open_set)

        if current == end:
            break

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # Possible moves
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:
                traffic_penalty = traffic_matrix[neighbor]
                new_cost = cost_so_far[current] + distance_matrix[neighbor] + traffic_penalty

                if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + heuristic_func(neighbor, end)
                    heapq.heappush(open_set, (priority, neighbor))
                    came_from[neighbor] = current

    return came_from, cost_so_far
```

Step 5: Reconstruct and Visualize the Path

Once the optimal path is found, reconstruct and plot it on the grid for visualization.

```
def reconstruct_path(came_from, start, end):
    path = []
    current = end
    while current != start:
        path.append(current)
        current = came_from.get(current)
    path.append(start)
    path.reverse()
    return path

# Visualization
def visualize_grid(distance_matrix, path):
    plt.imshow(distance_matrix, cmap="YlGn", origin="upper")
    path_x, path_y = zip(*path)
    plt.plot(path_y, path_x, marker="o", color="red")
    plt.title("Optimal Path")
    plt.colorbar(label="Distance")
    plt.show()
```

Step 6: Run the Experiment

Define a start and end location, choose a heuristic, and run the A* algorithm to find and visualize the optimal path.

```
start = (0, 0)  # Starting point
end = (9, 9)    # Destination

# Choose heuristic: Manhattan or Euclidean
heuristic = manhattan_distance
```

```
came_from, cost_so_far = a_star_algorithm(start, end, distance_matrix, traffic_matrix,
heuristic)
path = reconstruct_path(came_from, start, end)

# Visualize the path on the grid
visualize_grid(distance_matrix, path)
```

Labset 3: Heuristic Search

In [1]:
```
# import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import heapq  # For priority queue implementation
```

In [2]:
```
# Define the city grid and cost function
grid_size = (10, 10)  # Adjust grid size as needed
distance_matrix = np.random.randint(1, 10, size=grid_size)
traffic_matrix = np.random.randint(1, 5, size=grid_size)
```

In [3]:
```
# Define heuristic functions
def manhattan_distance(start, end):
    return abs(start[0] - end[0]) + abs(start[1] - end[1])

def euclidean_distance(start, end):
    return np.sqrt((start[0] - end[0])**2 + (start[1] - end[1])**2)
```

In [4]:
```
# Implement A* Algorithm
def a_star_algorithm(start, end, distance_matrix, traffic_matrix, heuristic
_func):
    rows, cols = distance_matrix.shape
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    cost_so_far = {start: 0}

    while open_set:
        current_cost, current = heapq.heappop(open_set)

        if current == end:
            break

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # Possible moves
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:
                traffic_penalty = traffic_matrix[neighbor]
                new_cost = cost_so_far[current] + distance_matrix[neighbor]
+ traffic_penalty

                if neighbor not in cost_so_far or new_cost < cost_so_far[ne
ighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + heuristic_func(neighbor, end)
                    heapq.heappush(open_set, (priority, neighbor))
                    came_from[neighbor] = current

    return came_from, cost_so_far
```

In [5]:

```python
# Reconstruct and visualize the path
def reconstruct_path(came_from, start, end):
    path = []
    current = end
    while current != start:
        path.append(current)
        current = came_from.get(current)
    path.append(start)
    path.reverse()
    return path

# Visualization
def visualize_grid(distance_matrix, path):
    plt.imshow(distance_matrix, cmap="YlGn", origin="upper")
    path_x, path_y = zip(*path)
    plt.plot(path_y, path_x, marker="o", color="red")
    plt.title("Optimal Path")
    plt.colorbar(label="Distance")
    plt.show()
```

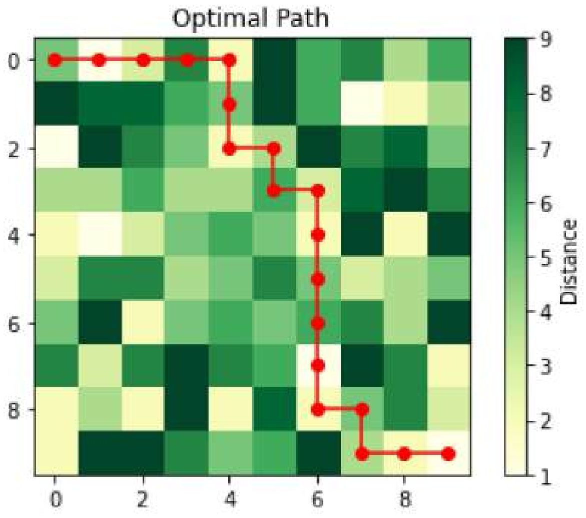
In [6]:

```python
# Run the experiment
start = (0, 0)  # Starting point
end = (9, 9)    # Destination

# Choose heuristic: Manhattan or Euclidean
heuristic = manhattan_distance

came_from, cost_so_far = a_star_algorithm(start, end, distance_matrix, traffic_matrix, heuristic)
path = reconstruct_path(came_from, start, end)

# Visualize the path on the grid
visualize_grid(distance_matrix, path)
```



In [ ]: