**Design and Analysis of Algorithm Assignment**

**Navigation Path Optimization for Autonomous Drones**

**Objective**

Develop and analyze path-planning algorithms to help autonomous drones navigate through urban environments. The focus is on finding optimal paths while minimizing energy consumption and avoiding restricted zones.

**Deliverables**

1. Code implementations for brute-force, Dijkstra’s, and A\* path planning algorithms.
2. Comparative analysis report on time complexity and algorithm performance.
3. Performance graphs showing the impact of algorithms on battery usage and travel time.

**1. Brute-Force Path Optimization**

**Objective**

Implement a brute-force algorithm that checks all possible routes to find the shortest path between waypoints.

**Instructions**

1. Use the itertools.permutations function to generate all possible paths between waypoints.
2. Calculate the total distance for each path, including the return to the starting point.
3. Find the path with the minimum cost and output it.

**Code Implementation**

from itertools import permutations

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

def brute\_force\_path(graph):

waypoints = range(len(graph))

min\_path\_cost = float('inf')

best\_path = None

for perm in permutations(waypoints):

current\_cost = 0

for i in range(len(perm) - 1):

current\_cost += graph[perm[i]][perm[i+1]]

current\_cost += graph[perm[-1]][perm[0]]

if current\_cost < min\_path\_cost:

min\_path\_cost = current\_cost

best\_path = perm

return best\_path, min\_path\_cost

best\_path, min\_path\_cost = brute\_force\_path(graph)

print("Best Path:", best\_path)

print("Minimum Path Cost:", min\_path\_cost)

**Analysis**

* **Time Complexity**: The brute-force method has a time complexity of O(n!)O(n!)O(n!), which is infeasible for large numbers of waypoints.
* **Example Calculation**: For 10 waypoints, the number of possible paths is 10!10!10!, or approximately 3.6 million paths.

**2. Dijkstra’s Algorithm for Shortest Path**

**Objective**

Implement Dijkstra’s algorithm to find the shortest path from a starting waypoint to all other waypoints.

**Instructions**

1. Use a priority queue (using Python’s heapq library) to efficiently retrieve the minimum distance nodes.
2. For each node, update the shortest known distance to its neighboring nodes.
3. Return the distances from the start to each node.

**Code Implementation**

import heapq

def dijkstra(graph, start):

num\_nodes = len(graph)

distances = [float('inf')] \* num\_nodes

distances[start] = 0

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_node = heapq.heappop(priority\_queue)

if current\_distance > distances[current\_node]:

continue

for neighbor, weight in enumerate(graph[current\_node]):

if weight > 0:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

distances\_from\_start = dijkstra(graph, start=0)

print("Shortest distances from start:", distances\_from\_start)

**Analysis**

* **Correctness**: Dijkstra’s algorithm guarantees the shortest path for graphs with non-negative weights.
* **Time Complexity**: 𝑂((V+E) log V), where V is the number of vertices and E is the number of edges.

**3. A Algorithm for Shortest Path\***

**Objective**

Implement A\* algorithm to find the optimal path from a start node to a goal node using a heuristic function.

**Instructions**

1. Define a heuristic function that estimates the distance from each node to the goal.
2. Adapt Dijkstra’s algorithm by adding the heuristic to prioritize paths that appear promising.
3. Return the path and total cost from the start to the goal.

**Code Implementation**

from math import sqrt

def heuristic(node, goal, coordinates):

(x1, y1) = coordinates[node]

(x2, y2) = coordinates[goal]

return sqrt((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2)

def a\_star(graph, start, goal, coordinates):

num\_nodes = len(graph)

open\_set = [(0, start)]

g\_costs = {node: float('inf') for node in range(num\_nodes)}

g\_costs[start] = 0

f\_costs = {node: float('inf') for node in range(num\_nodes)}

f\_costs[start] = heuristic(start, goal, coordinates)

came\_from = {}

while open\_set:

\_, current = heapq.heappop(open\_set)

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

return path[::-1], g\_costs[goal]

for neighbor, weight in enumerate(graph[current]):

if weight > 0:

tentative\_g\_score = g\_costs[current] + weight

if tentative\_g\_score < g\_costs[neighbor]:

came\_from[neighbor] = current

g\_costs[neighbor] = tentative\_g\_score

f\_costs[neighbor] = tentative\_g\_score + heuristic(neighbor, goal, coordinates)

heapq.heappush(open\_set, (f\_costs[neighbor], neighbor))

return None, float('inf')

coordinates = [(0, 0), (1, 2), (3, 1), (4, 3)]

path, cost = a\_star(graph, start=0, goal=3, coordinates=coordinates)

print("A\* Path:", path)

print("Path Cost:", cost)

**Analysis**

* **Correctness**: With an admissible heuristic, A\* guarantees the shortest path.
* **Time Complexity**: Similar to Dijkstra’s, approximately O((V+E)logV), but heuristic-dependent.

**4. Comparative Analysis Report**

**Content**

* **Algorithm Comparisons**: Discuss each algorithm’s complexity, scalability, and suitability for real-time pathfinding in urban grids.
* **Battery and Travel Time**: Simulate performance data, comparing battery usage and travel time for each algorithm.

**Suggested Metrics**

1. **Energy Consumption**: Measure and graph the energy used over various path lengths and grid sizes.
2. **Travel Time**: Graph travel times based on reached waypoints and restricted zones.

**5. Performance Graphs and Charts**

Use libraries such as matplotlib to plot:

* **Energy vs. Path Length**: Show how each algorithm affects battery usage as path length increases.
* **Time vs. Complexity**: Display execution times for each algorithm across varying waypoint configurations.

**Execution Instructions**

* **Environment**: Use Python, with dependencies such as networkx (for graph representation), matplotlib (for visualizations), and numpy (for matrix handling).
* **Instructions**:
  1. Clone the project repository.
  2. Install dependencies with pip install -r requirements.txt.
  3. Run each algorithm’s test script, following comments in the code to observe outputs.