NED University of Engineering & Technology

Computer Science & Information Technology Department

Smart Route: Finding the Best Way to NED

Course Title: Design & Analysis of Algorithm
Course Code: CT-363
Instructor: Dr. Usman Amjad

Group Members:

Maham Tahir (DT-001)
Afifa Siddiqui (DT-003)
Ashna Shaikh (DT-019)
Sabrina Shahzad (DT-026)

1- Introduction

1.1 Background

In a city as intricate as Karachi, navigating daily commutes can be quite difficult — particularly when selecting the most effective route. This initiative seeks to enhance travel for four individuals starting from North Nazimabad, North Karachi, Gulshan-e-Iqbal, and Water Pump, all of whom are headed to NED University. With four potential entry gates and diverse traffic patterns, the objective is to identify the shortest and quickest route for each person utilizing various search algorithms. By representing Karachi as a graph, we evaluate and contrast DFS, BFS, Dijkstra's, and A* algorithms to provide informed gate suggestions.

1.2 Problem Statement

The traffic conditions in Karachi can fluctuate greatly at different times of the day, and many navigation systems do not effectively consider changing congestion levels when proposing routes. The goal is to develop a system that not only identifies the quickest route but also adapts to traffic conditions, providing alternative options when traffic is heavy. A smart solution is required to emulate this situation and assist users in making educated decisions about their commutes.

1.3 Objectives

The goal is to represent the road network of Karachi as a weighted graph that includes both distance and congestion factors. We aim to implement and evaluate the effectiveness of BFS, DFS, Dijkstra, and A* algorithms. The objective is to determine the quickest route to NED University from various starting locations. We plan to provide alternative routing options dynamically when congestion levels are high. Additionally, we will visualize route suggestions and accommodate input from multiple sources.

1.4 Technologies Used

- **Python:** Core development language.
- **NetworkX:** For graph modeling and manipulation.
- **Matplotlib:** For visualizing the graph and computed paths.
- **Flask:** Web-friendly backend for interactive deployment.
- **Random module:** Simulates real-time congestion.

2- Methodology

2.1 Graph Construction

The roadway system of Karachi was modeled as a weighted undirected graph utilizing the NetworkX library. Every node symbolizes a place (for example, North Karachi, Gulshan), while each edge signifies a road segment containing two attributes.

- Distance (in km)
- Congestion factor (randomized between 0.1 to 0.8)

The total travel time is calculated dynamically using both distance and congestion:

Time = (Distance / 30) * 60 * (1 + Congestion)

2.2 Congestion Simulation

To emulate actual traffic conditions, congestion values are generated randomly with Python's random module. These values are refreshed prior to the execution of each algorithm to represent changing traffic circumstances.

2.3 Algorithm Implementation

Four pathfinding algorithms were implemented and compared:

- **BFS (Breadth-First Search):** Explores all neighbors at the current depth before moving deeper.
- **DFS (Depth-First Search):** Explores as far as possible down each branch before backtracking.
- **Dijkstra's Algorithm:** Finds the shortest path considering cumulative travel time as cost.
- **A* Algorithm:** Enhances Dijkstra with a heuristic (currently zero for uniform behavior).

2.4 Alternate Path Detection

If a route contains a section with congestion exceeding 0.4, the system activates a backup procedure. A new graph is created by excluding edges with high congestion, and Dijkstra's algorithm is executed once more to propose an alternative path.

2.5 Visualization

Matplotlib and NetworkX were used to visualize:

- The full road network with edge weights.
- Paths generated by each algorithm.
- Alternate paths when congestion is high.
- Multi-source to single-destination routing (highlighting all commuters heading to NED).

3- Results & Discussions

3.1 Path and Time Analysis

Each algorithm was tested to find the best route from four different locations (North Karachi, North Nazimabad, Gulshan-e-Iqbal, Water Pump) to NED University. The algorithms considered both distance and congestion. Below is the travel time (in minutes) calculated for each algorithm from a single source:

(**From:** North Nazimabad **To:** NED University main gate)

Algorithm	Path Time (in mins)	Alternate Path Time (if congested)
BFS	28 mins	N/A
DFS	28 mins	N/A
Dijkstra	25.8 mins	26.8 mins
A*	25.8 mins	26.8 mins

3.2 Evaluation

- BFS and DFS performed adequately but did not guarantee the shortest time due to their unweighted nature and traversal logic.
- Dijkstra and A* consistently found the fastest paths, incorporating congestion dynamically.
- Alternate Path Logic was effectively triggered for congestion thresholds > 0.4, rerouting users through less congested roads even if slightly longer.

4- Conclusion & Future Work

4.1 Conclusion

This project showcased an intelligent and efficient routing system aimed at navigating the intricate traffic patterns of Karachi. By modeling the city's roads as a graph and incorporating congestion-aware logic with classic pathfinding algorithms (BFS, DFS, Dijkstra, and A*), we successfully delivered precise and optimized route suggestions to NED University. Among all the algorithms, Dijkstra and A* excelled by providing the shortest travel times and adapting dynamically to fluctuating congestion levels. Additionally, the provision of alternative route suggestions further improved the system's practicality and reliability for users.

4.2 Future Work

To enhance the system's robustness and enable real-time functionality, the following enhancements could be implemented:

- **Real-Time Traffic Data:** Integrate live traffic information sourced from APIs like Google Maps or Mapbox.
- **Heuristic Improvements for A*:** Enhance the A* algorithm by incorporating geographic heuristics, such as the Haversine formula.
- **User Preferences:** Allow users to choose their priority, whether it be the shortest distance, lowest traffic, or least travel time.
- **Mobile Application Development:** Create a mobile-compatible interface for users commuting.
- **Expanding Reach:** Develop the system to cover more areas within Karachi or other cities.

5- Modular Overview of the Routing Algorithm Implementation

5.1 Importing Required Libraries

- **import networkx as nx:** Used to create, manipulate, and analyze complex networks represented as graphs (nodes and edges). Essential for modeling Karachi's routes as a graph.
- **import matplotlib, matplotlib.use('Agg'):** Configures Matplotlib to use a non-GUI backend (`'Agg'`) for rendering plots in server environments (like Flask), where display windows are not available.
- **import matplotlib.pyplot as plt:** Used for generating and saving graph visualizations including paths, nodes, and congestion annotations.
- **import heapq:** Provides a priority queue (min-heap) implementation, used in Dijkstra's and A* algorithms for efficient path cost retrieval.
- **import random:** Generates random congestion values to simulate live traffic conditions dynamically.
- **import os:** Enables interaction with the operating system, such as creating folders and managing file paths for saving visualizations.
- **import re:** Used to clean algorithm names for safe file naming (e.g., removing special characters from graph image filenames).

5.2 Graph Setup

This part describes the road network in terms of a graph, with each node indicating a location and each edge comprising a tuple (distance, congestion_level). Each key signifies a starting location. Each inner dictionary outlines the destinations that can be accessed from that source, along with their corresponding distance and congestion level (ranging from 0 to 1).

```python
# ---------------- GRAPH SETUP ----------------
graph = {
    'North Karachi': {'North Nazimabad': (7, 0.2), 'Water Pump': (5, 0.5), 'Gulshan': (9, 0.3),
                      'NED Main Gate': (12, 0.4), 'NED Maskan Gate': (12, 0.4),
                      'NED Staff Gate': (12, 0.4), 'NED Visitors Gate': (12, 0.4)},
    'North Nazimabad': {'Water Pump': (3, 0.3), 'Gulshan': (6, 0.6),
                        'NED Main Gate': (10, 0.4), 'NED Maskan Gate': (10, 0.4),
                        'NED Staff Gate': (10, 0.4), 'NED Visitors Gate': (10, 0.4)},
    'Water Pump': {'Gulshan': (4, 0.4), 'NED Main Gate': (6, 0.5),
                   'NED Maskan Gate': (6, 0.5), 'NED Staff Gate': (6, 0.5),
                   'NED Visitors Gate': (6, 0.5)},
    'Gulshan': {'NED Main Gate': (3, 0.3), 'NED Maskan Gate': (2, 0.3),
                'NED Staff Gate': (2, 0.3), 'NED Visitors Gate': (2, 0.3)},
}
```

5.3 Simulating Congestion

Randomly updates the congestion values to simulate real-time traffic fluctuations.

```python
def simulate_live_congestion():
    for source in graph:
        for dest in graph[source]:
            distance = graph[source][dest][0]
            graph[source][dest] = (distance, round(random.uniform(0.1, 0.8), 2))
```

5.4 Travel Time Calculation

Calculates time by adjusting base speed (30 km/h) with the congestion factor.

```python
# Function to update travel time considering congestion
def update_travel_time(distance, congestion):
    return round((distance / 30) * 60 * (1 + congestion), 2)
```

5.5 Building the NetworkX Graph

Creates a weighted graph using NetworkX where weights represent travel times.

```python
# Build the graph
G = nx.Graph()
for source, destinations in graph.items():
    for dest, (distance, congestion) in destinations.items():
        time = update_travel_time(distance, congestion)
        G.add_edge(source, dest, weight=time)
```

5.6 Search Algorithms

5.6.1 Basic Algorithms

5.6.1.1 Breadth-First Search Algorithm

Breadth-First Search (BFS) explores a graph level by level starting from the source node.It uses a queue to visit the nearest unvisited neighbors first, ensuring the shortest path in terms of number of steps, but not necessarily the least cost/time.

Steps:

1- Start at the source node.
2- Visit all immediate neighbors.
3- Add them to a queue.
4- Repeat for each neighbor until the goal is reached.

Pseudocode:

```
FUNCTION BFS(start,
  goal): queue = [(start,
  [start], 0)] visited =
  SET()

  WHILE queue is not empty:
    vertex, path, cost =
    queue.pop(0)

    IF vertex == goal:
      RETURN (path,
      cost)

    visited.ADD(vertex)

    FOR neighbor IN graph.GET(vertex,
      {}): IF neighbor NOT IN visited:
        distance, congestion = graph[vertex][neighbor]
        travel_time = UPDATE_TRAVEL_TIME(distance, congestion)
        queue.APPEND((neighbor, path + [neighbor], cost + travel_time))

  RETURN (None,
+infinity) END
FUNCTION
```

Limitations:

- Ignores edge weights (like traffic congestion or travel time).
- May choose a longer time path even if it has fewer steps.

```python
# Algorithms
def bfs(start, goal):
    queue = [(start, [start], 0)]
    visited = set()
    while queue:
        vertex, path, cost = queue.pop(0)
        if vertex == goal:
            return path, cost
        visited.add(vertex)
        for neighbor in graph.get(vertex, {}):
            if neighbor not in visited:
                distance, congestion = graph[vertex][neighbor]
                travel_time = update_travel_time(distance, congestion)
                queue.append((neighbor, path + [neighbor], cost + travel_time))
    return None, float('inf')
```

Explores all neighbors at the current depth before going deeper, ensuring the shortest path in terms of hops, not weight.

5.6.1.2 Depth-First Search Algorithm

Depth-First Search (DFS) explores as deep as possible along a path before backtracking. It uses a stack (or recursion) to go down one branch fully before trying alternatives.

Steps:

1- Start at the source node.
2- Visit one neighbor and keep going deeper.
3- Backtrack when no unvisited neighbors remain.
4- Continue until the goal is found.

Pseudocode:

```
FUNCTION DFS(start,
    goal): stack = [(start,
    [start], 0)] visited =
    SET()

    WHILE stack is not empty:
        vertex, path, cost =
        stack.POP()

        IF vertex == goal:
            RETURN (path,
            cost)

        IF vertex NOT IN
            visited:
            visited.ADD(vertex)

            FOR neighbor IN graph.GET(vertex,
                {}): IF neighbor NOT IN visited:
                    distance, congestion =
graph[vertex][neighbor]
                    travel_time =
UPDATE_TRAVEL_TIME(distance,
congestion)
                    stack.APPEND((neighbor,
path + [neighbor], cost + travel_time))

    RETURN (None,
+infinity) END
FUNCTION
```

Limitation:

- Does not guarantee the shortest or optimal path.
- Can get stuck in long or inefficient branches.

```python
def dfs(start, goal):
    stack = [(start, [start], 0)]
    visited = set()
    while stack:
        vertex, path, cost = stack.pop()
        if vertex == goal:
            return path, cost
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in graph.get(vertex, {}):
                if neighbor not in visited:
                    distance, congestion = graph[vertex][neighbor]
                    travel_time = update_travel_time(distance, congestion)
                    stack.append((neighbor, path + [neighbor], cost + travel_time))
    return None, float('inf')
```

Explores as far as possible along each branch before backtracking. Not optimal for the shortest travel time.

5.6.2 Advanced Algorithms

5.6.2.1 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a starting node to all other nodes in a weighted graph with non-negative edge weights.

Steps:

1- Start at the source node and set its cost to 0; all others to infinity.
2- Use a priority queue to always explore the node with the smallest known cost.
3- For each neighbor, calculate the new cost: new_cost = current_cost + edge_weight
4- If the new_cost is smaller, update it and continue.
5- Repeat until the destination is reached.

Pseudocode:

```
FUNCTION   DIJKSTRA(start,
   goal):  priority_queue  =  [(0,
   start, [start])] visited = SET()

   WHILE priority_queue is not empty:
      cost, vertex, path = HEAPOP(priority_queue)

      IF vertex == goal:
         RETURN (path,
         cost)

      IF vertex NOT IN
         visited:
         visited.ADD(verte
         x)

         FOR neighbor IN
            graph.GET(vertex, {}): IF
            neighbor NOT IN visited:
               distance, congestion = graph[vertex][neighbor]
               travel_time = UPDATE_TRAVEL_TIME(distance, congestion)
               HEAPPUSH(priority_queue, (cost + travel_time, neighbor, path +
               [neighbor]))

   RETURN (None,
+infinity) END
FUNCTION
```

Why It's Good:

- **Always gives the shortest path** in weighted graphs.
- Takes **congestion (travel time)** into account through edge weights.

```
def dijkstra(start, goal):
    queue = [(0, start, [start])]
    visited = set()
    while queue:
        cost, vertex, path = heapq.heappop(queue)
        if vertex == goal:
            return path, cost
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in graph.get(vertex, {}):
                if neighbor not in visited:
                    distance, congestion = graph[vertex][neighbor]
                    travel_time = update_travel_time(distance, congestion)
                    heapq.heappush(queue, (cost + travel_time, neighbor, path + [neighbor]))
    return None, float('inf')

```

Finds the shortest path considering weights (travel time). Greedy algorithm.

5.6.2.2 A Algorithm*

A* is an informed search algorithm that finds the shortest and fastest path by combining actual cost and estimated cost.

Formula:

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

- **g(n)** = actual cost from the start to current node `n`
- **h(n)** = heuristic (estimated cost from `n` to goal)

Steps:

1- Start at the source node and initialize cost values.
2- Use a **priority queue** based on the lowest `f(n)`.
3- At each step, expand the node with the **lowest estimated total cost**.
4- Continue until the goal is reached.

Pseudocode:

```
FUNCTION HEURISTIC(node, goal):
  RETURN distance_estimate_between(node,
goal) END FUNCTION

FUNCTION A_STAR(start, goal):
  priority_queue = [(0 + HEURISTIC(start, goal), 0, start, [start])]
  visited = SET()

  WHILE priority_queue is not empty:
    est_total, cost, vertex, path = HEAPPOP(priority_queue)

    IF vertex == goal:
      RETURN (path,
      cost)

    IF vertex NOT IN
      visited:
      visited.ADD(verte
      x)

      FOR neighbor IN
        graph.GET(vertex, {}): IF
        neighbor NOT IN visited:
          distance, congestion = graph[vertex][neighbor]
          travel_time = UPDATE_TRAVEL_TIME(distance, congestion)
          new_cost = cost + travel_time
          estimate = new_cost + HEURISTIC(neighbor, goal)
```

HEAPPUSH(priority_queue, (estimate, new_cost, neighbor, path + [neighbor]))

RETURN (None, +infinity) END FUNCTION

Why It's Effective:

- Faster than Dijkstra when a good heuristic is used.
- Finds optimal paths with fewer computations.
- Best choice when speed and accuracy are both needed.

```python
def heuristic(node, goal):
    return 0

def a_star(start, goal):
    queue = [(0 + heuristic(start, goal), 0, start, [start])]
    visited = set()
    while queue:
        est_total, cost, vertex, path = heapq.heappop(queue)
        if vertex == goal:
            return path, cost
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in graph.get(vertex, {}):
                if neighbor not in visited:
                    distance, congestion = graph[vertex][neighbor]
                    travel_time = update_travel_time(distance, congestion)
                    est = cost + travel_time + heuristic(neighbor, goal)
                    heapq.heappush(queue, (est, cost + travel_time, neighbor, path + [neighbor]))
    return None, float('inf')
```

Enhancement over Dijkstra by using a heuristic. In this case, **heuristic()** is set to 0 (i.e., behaves like Dijkstra).

5.7 Graph Visualization

Draws the entire graph with edge weights and saves the image.

```python
def visualize_graph():
    pos = nx.spring_layout(G, seed=42)
    plt.figure(figsize=(10, 8))
    nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=2000, font_size=10)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels={k: f"{v:.1f}" for k, v in labels.items()})
    plt.title("Graph Visualization")

    # Save graph as PNG in static folder
    os.makedirs("static/algos", exist_ok=True)
    file_path = "static/algos/graph_visualization.png"
    plt.savefig(file_path)
    plt.close()
    return file_path
```

5.8 Path Visualization

Highlights the path found by the algorithm and optionally marks an alternate route if available.

```python
def visualize_path(path, algo_name, total_time, alt_path=None, alt_time=None):
    pos = nx.spring_layout(G, seed=42)
    plt.figure(figsize=(10, 8))
    nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=2500, font_size=12)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels={k: f"{v:.1f}" for k, v in labels.items()})
    nx.draw_networkx_nodes(G, pos, nodelist=path, node_color='orange', node_size=3000)

    if alt_path:
        nx.draw_networkx_nodes(G, pos, nodelist=alt_path, node_color='red', node_size=2000, alpha=0.6)
        edges_in_alt = list(zip(alt_path, alt_path[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=edges_in_alt, width=2, edge_color='red', style='dashed')

    plt.title(f"{algo_name} Path: {path} | Time: {total_time} min"
              + (f"\nAlternate Path: {alt_path} | Time: {alt_time} min" if alt_path else ""))


    safe_algo_name = re.sub(r'[^a-zA-Z0-9_-]', '_', algo_name)  # replaces anything not a-z, A-Z, 0-9, _
    filename = f"{safe_algo_name}.png"
```

5.9 Congestion Handling

If congestion exceeds 0.4 on any segment, it filters out highly congested paths and reruns Dijkstra to find an alternate.

```python
# Function to find an alternate path if congestion is high
def find_alternate_path_if_needed(path, algo_func, start, goal):
    congested = False
    for i in range(len(path)-1):
        u, v = path[i], path[i+1]
        congestion = graph[u][v][1]
        if congestion > 0.4:
            congested = True
            break
    if congested:
        print("⚠ High congestion detected! Searching for alternate path...")
        new_graph = {
            u: {v: (d, c) for v, (d, c) in dest.items() if c <= 0.4}
            for u, dest in graph.items()
        }
        path_alt, time_alt = dijkstra_custom(new_graph, start, goal)
        return path_alt, time_alt
    return None, None
```

5.10 Alternate Dijkstra (Custom Graph)

Dijkstra re-run on a filtered graph excluding high congestion edges.

```python
# Dijkstra for custom graph considering congestion
def dijkstra_custom(custom_graph, start, goal):
    queue = [(0, start, [start])]
    visited = set()
    while queue:
        cost, vertex, path = heapq.heappop(queue)
        if vertex == goal:
            return path, cost
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in custom_graph.get(vertex, {}):
                if neighbor not in visited:
                    distance, congestion = custom_graph[vertex][neighbor]
                    travel_time = update_travel_time(distance, congestion)
                    heapq.heappush(queue, (cost + travel_time, neighbor, path + [neighbor]))
    return None, float('inf')
```

5.11 Multi-Source Visualization

Highlights multiple starting points and one shared destination (NED).

```python
def visualize_multi_source_graph(sources, destination):
    pos = nx.spring_layout(G, seed=42)
    plt.figure(figsize=(10, 8))
    node_colors = []
    for node in G.nodes:
        if node in sources:
            node_colors.append('lightgreen')
        elif node == destination:
            node_colors.append('red')
        else:
            node_colors.append('skyblue')

    nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=2500, font_size=11)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels={k: f"{v:.1f}" for k, v in labels.items()})
    plt.title(f"Multi-Source Visualization\nSources: {', '.join(sources)} | Destination: {destination}")

    os.makedirs("static/algos", exist_ok=True)
    file_path = "static/algos/multi_source_graph.png"
    plt.savefig(file_path)
    plt.close()
    return file_path
```

5.12 Flask-Friendly Processing

Combines all algorithm outputs, alternate paths, and visualization generation. Designed to work with a Flask frontend.

```python
# Flask-friendly function to process start, goal, and sources for multi-source
def process_paths(start, goal, sources_input=None):
    if start not in G.nodes or goal not in G.nodes:
        return "Invalid input! Please check the location names."

    results = []
    for algo_name, algo_func in [('BFS', bfs), ('DFS', dfs), ('Dijkstra', dijkstra), ('A*', a_star)]:
        path, total_time = algo_func(start, goal)
        alt_path, alt_time = find_alternate_path_if_needed(path, algo_func, start, goal)
        path_file = visualize_path(path, algo_name, total_time, alt_path, alt_time)
        results.append({
            'algorithm': algo_name,
            'path': path,
            'total_time': total_time,
            'alt_path': alt_path,
            'alt_time': alt_time,
            'path_image': path_file  # Include path image URL
        })

    if sources_input:
        multi_sources = [s.strip() for s in sources_input.split(',') if s.strip() in G.nodes]
        if start not in multi_sources:
            multi_sources.insert(0, start)
        if multi_sources:
            multi_source_file = visualize_multi_source_graph(multi_sources, goal)
            results.append({'multi_source_graph': multi_source_file})  # Include multi-source graph URL

    return results
```

6- Comparison of Routing Algorithms: Best Performance

The clear winner here is Dijkstra's algorithm and by extension A* since its heuristic is zero.

1- Considers True Travel Time

- BFS and DFS ignore edge weights (they treat every step as equal), so they both return a 28 min route even though faster routes exist.
- Dijkstra and A* explicitly use the congestion-adjusted travel-time weights, yielding the true shortest-time path of 25.8 min.

2- Guaranteed Optimality

- Dijkstra's algorithm probably finds the minimum-cost path in a weighted graph. Neither BFS or DFS can guarantee that when weights vary.

3- Alternate-Path Handling

- Even when forced into a low-congestion alternate, Dijkstra's reroute is 26.8 min, still better than BFS/DFS's 28 min.

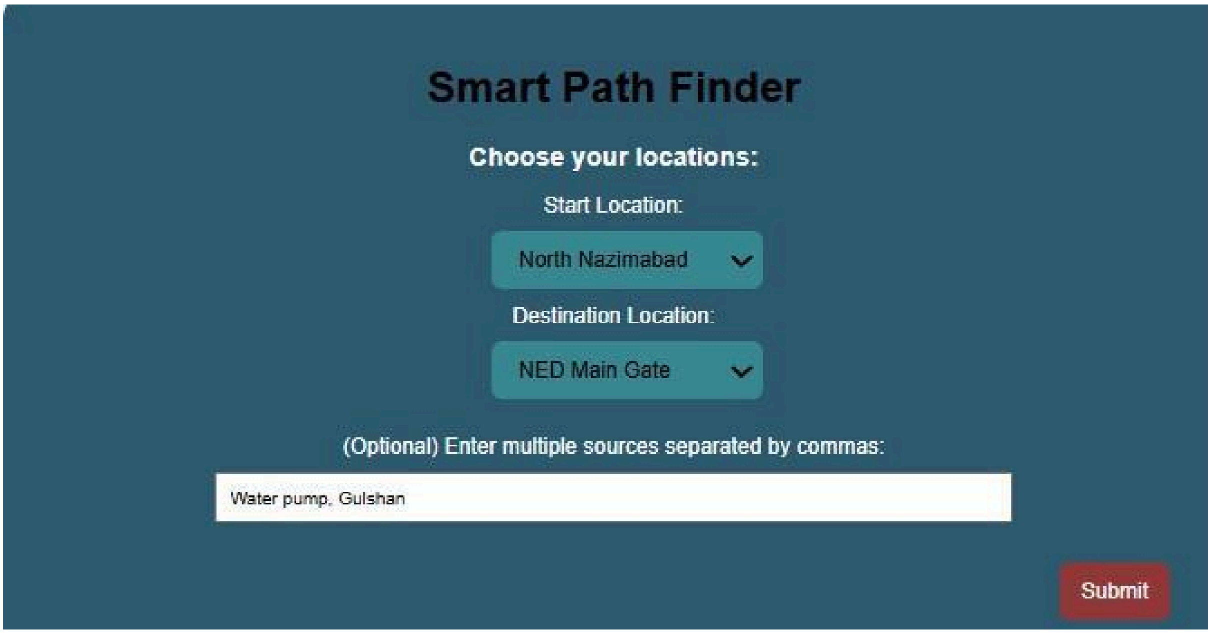
4- A Equivalence & Extensibility*

- With a zero heuristic, A* degenerates to Dijkstra—so it also finds 25.8 min.
- **Why use A***? If you later supply an admissible heuristic (e.g., straight-line distance), A* can explore far fewer nodes and run faster, while retaining optimality.

Best Pick:

Dijkstra (or A* with no heuristic) for guaranteed shortest travel time of **25.8 min**.
If performance/scalability becomes an issue on large graphs, switch to **A*** with a good heuristic to speed up the search without sacrificing optimality.

7- Output


Smart Path Finder
Choose your locations:
Start Location:
North Nazimabad
Destination Location:
NED Main Gate
(Optional) Enter multiple sources separated by commas:
Water pump, Gulshan
Submit

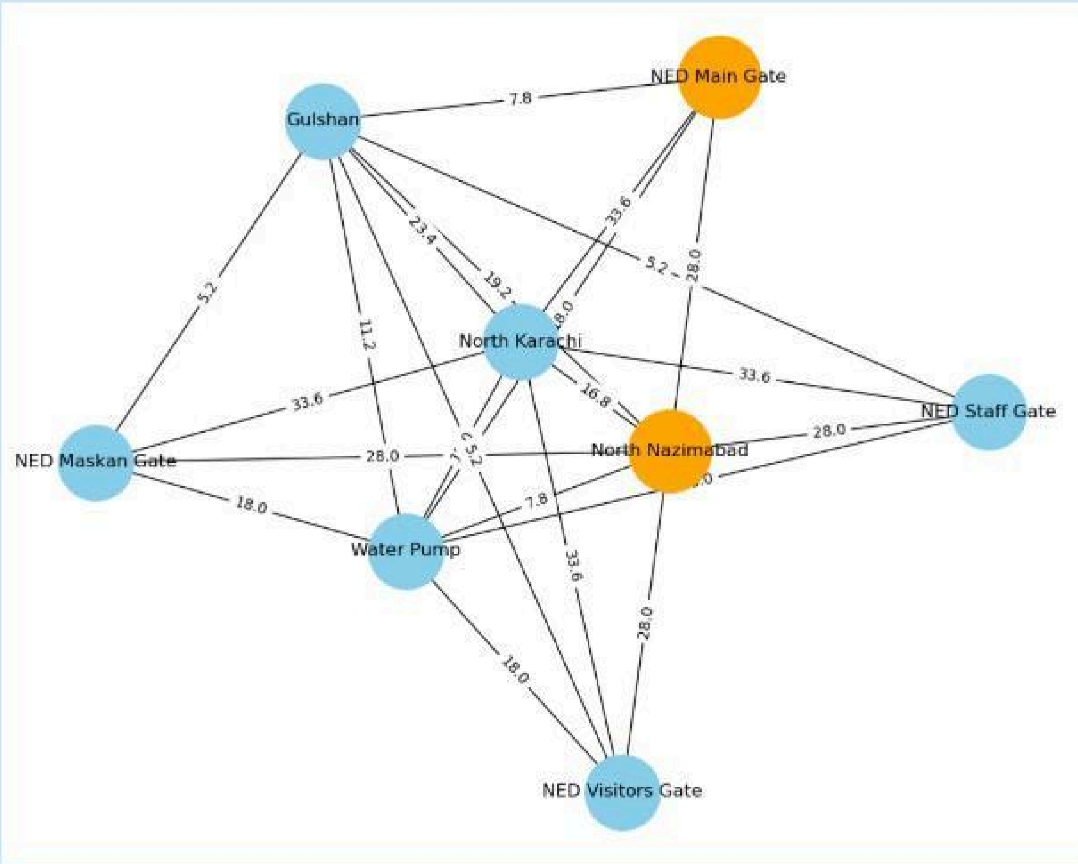
BFS:

Path: North Nazimabad → NED Main Gate

Total Time: 28 mins

Alternate Path: N/A

Alternate Time: null mins



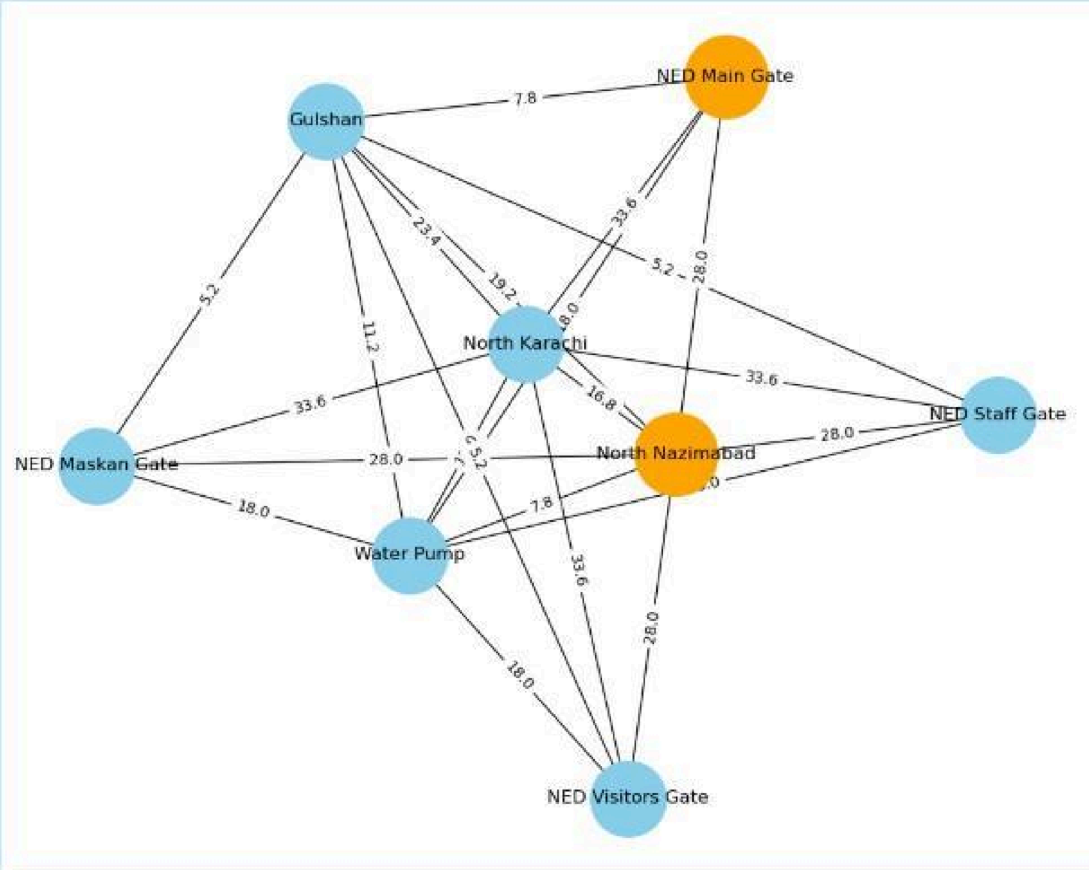
DFS:

Path: North Nazimabad → NED Main Gate

Total Time: 28 mins

Alternate Path: N/A

Alternate Time: null mins



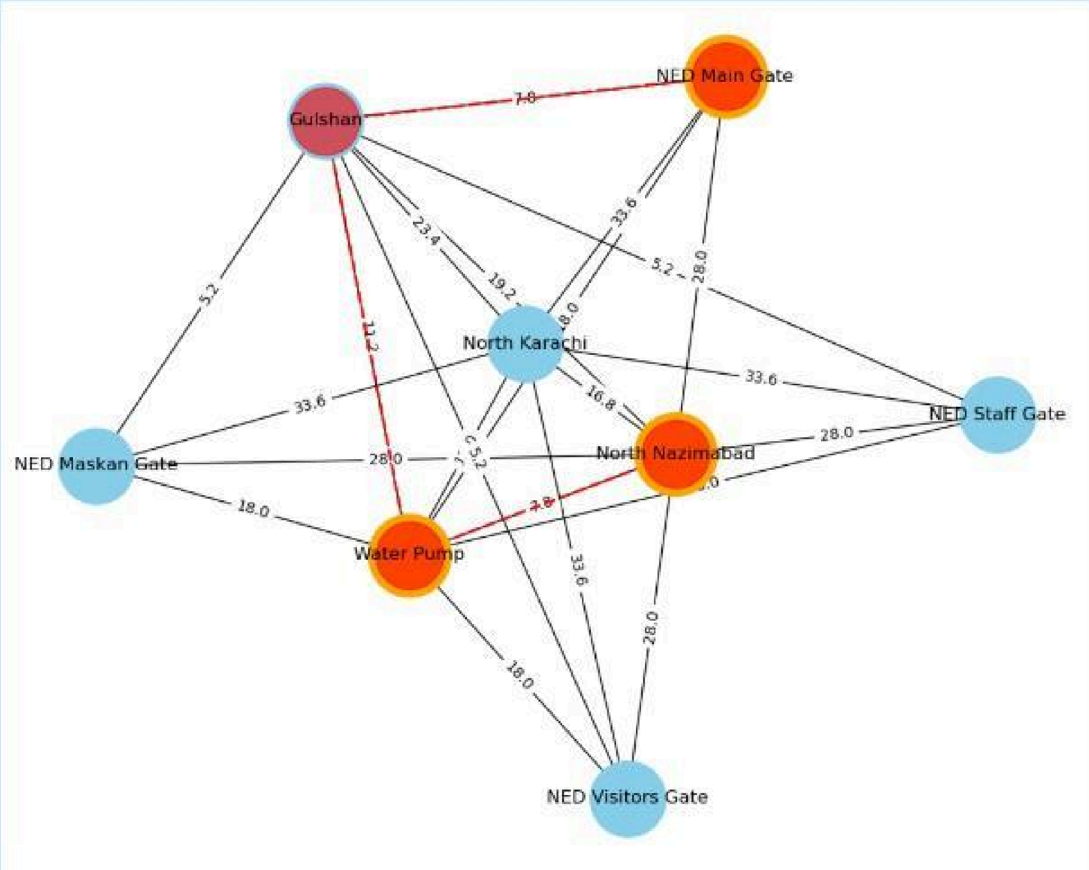
Dijkstra:

Path: North Nazimabad → Water Pump → NED Main Gate

Total Time: 25.8 mins

Alternate Path: North Nazimabad → Water Pump → Gulshan → NED Main Gate

Alternate Time: 26.8 mins



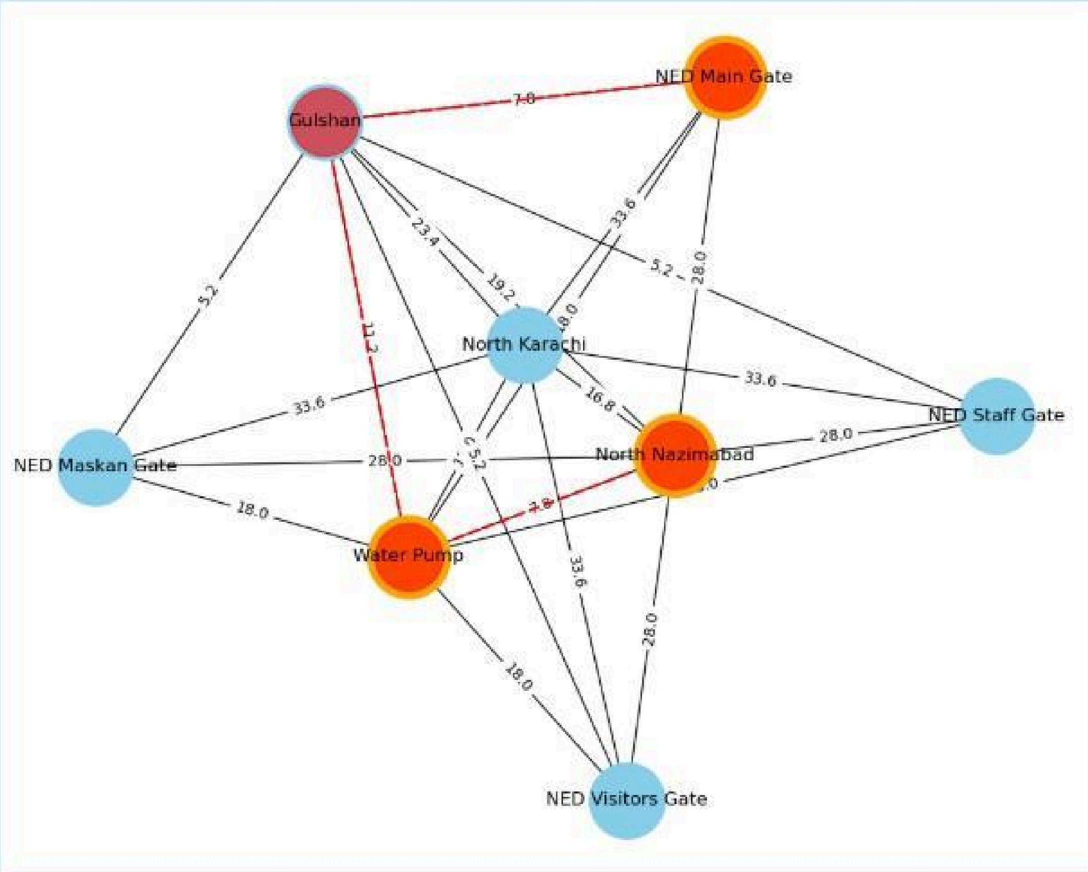
A*:

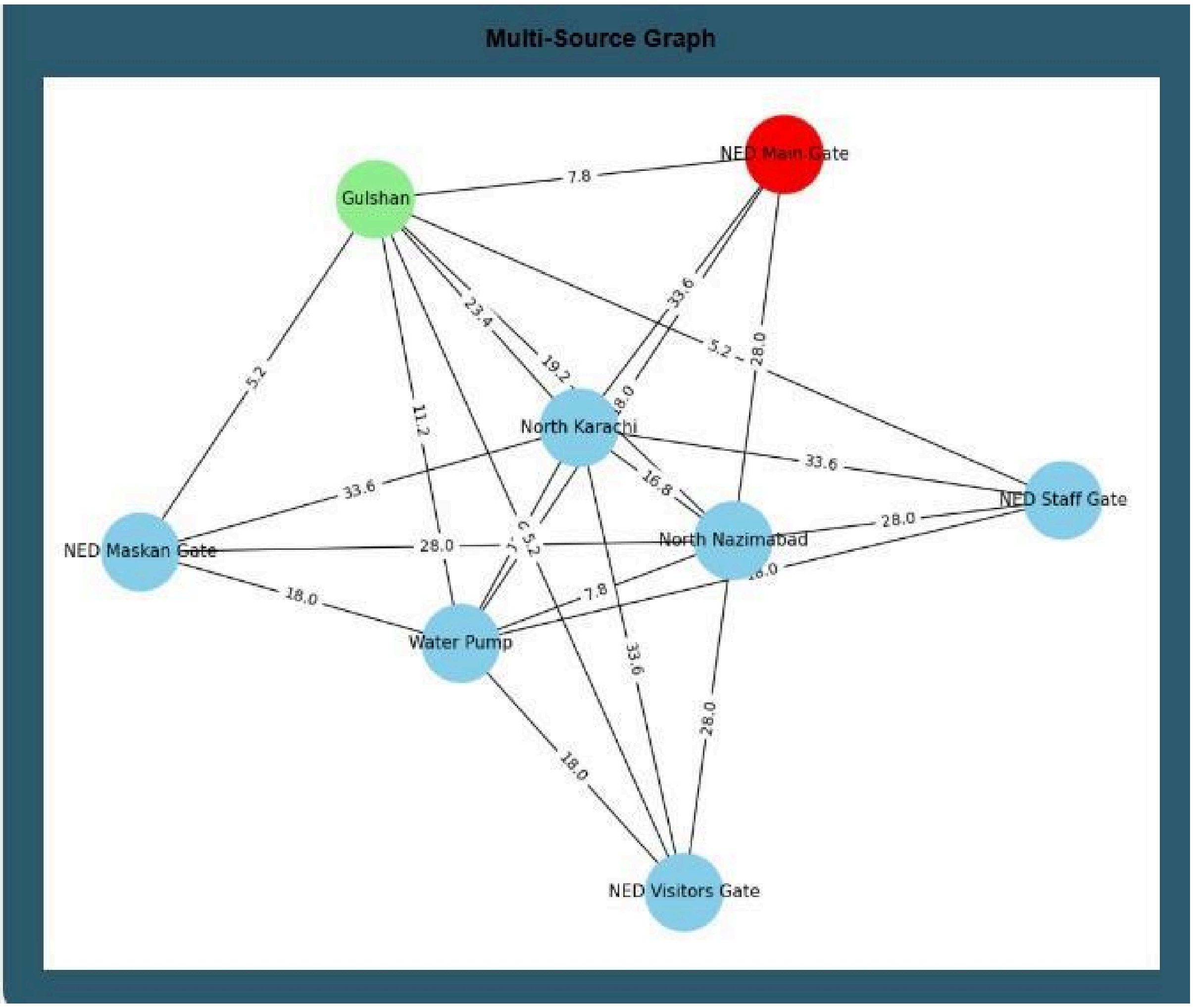
Path: North Nazimabad → Water Pump → NED Main Gate

Total Time: 25.8 mins

Alternate Path: North Nazimabad → Water Pump → Gulshan → NED Main Gate

Alternate Time: 26.8 mins




Multi-Source Graph
Gulshan
NED Main Gate
North Karachi
NED Staff Gate
North Nazimabad
NED Maskan Gate
Water Pump
NED Visitors Gate
7.8
23.4
33.6
19.2
5.2
28.0
5.2
11.2
18.0
33.6
33.6
16.8
28.0
28.0
5.2
7.8
18.0
33.6
28.0
18.0