# Real-Time Route Optimization Between Cities Using Spatio-Temporal Dijkstra with Weather and Traffic Data

## 1 Problem

Design and implement a novel real-time pathfinding algorithm that computes the **safest and shortest** route between two cities. Your algorithm must dynamically integrate the following key sources of information:

1. **Google Maps data** to understand the geospatial layout and road network structure,
2. **Real-time weather conditions**, including hazards such as storms, snow, and poor visibility that may impact road safety, and
3. **Live traffic information** to avoid congestion and minimize travel time.

The objective is to develop a robust and adaptive algorithm that can adjust the computed route in response to dynamically changing environmental and traffic conditions. The solution should emphasize both safety and efficiency by incorporating real-time data fusion, intelligent decision-making, and continuous route re-evaluation.

1. The design of the core algorithm, including data fusion methods,
2. The decision-making strategy used to balance competing objectives (e.g., safety, distance, and time),
3. The definition and use of heuristic functions guiding the pathfinding process.

Additionally, you must provide a comprehensive, step-by-step example demonstrating how the algorithm identifies an optimal path from a source city to a destination city. The example should demonstrate the algorithm's dynamic behavior in response to changing weather or traffic data, and explain how route recalculation occurs in real-time.

This assignment should be written with the clarity and depth expected in a professional research or engineering context. Emphasis should be placed on technical soundness, innovation, and effective communication of complex algorithmic behavior.

Creating a real-time search algorithm for discovering a safe and efficient path between two cities using Google Maps data represents a challenging and ambitious project. An effective approach involves combining Dijkstra's algorithm, a method for determining the shortest path, with a spatio-temporal algorithm that takes traffic information into account. Below, we outline the steps to develop such an algorithm.

## 2 Solution

### Methodology

#### Dijkstra's Algorithm:

Dijkstra's algorithm is a fundamental graph search technique designed to identify the shortest path between two nodes within a graph containing weighted edges. This algorithm finds widespread application in network routing, navigation, transportation, and logistics problem-solving scenarios.

When the sum of the distance from node $u$ to node $v$ ($\text{dist}(u)$) and the length of the edge connecting $u$ and $v$ ($\text{len}(u, v)$) is less than the current distance from node $v$ ($\text{dist}(v)$), the algorithm updates $\text{dist}(v)$ as follows:

$$\text{dist}(v) = \text{dist}(u) + \text{len}(u, v)$$

**Spatio-Temporal Algorithm:**

The term "spatio-temporal" pertains to data or phenomena exhibiting variations both in spatial (location-based) and temporal (time-based) dimensions. This concept finds extensive use in geographic information systems (GIS) and various domains for analyzing and modeling data featuring both spatial and temporal attributes.

## Graph Representation:

The graph consists of five nodes (cities) with the following adjacency matrix representing edge weights:

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
|---|---|---|---|---|---|
| $v_1$ | 3 | 8 | $\infty$ | 1 | 7 |
| $v_2$ | $\infty$ | 0 | $\infty$ | 1 | 7 |
| $v_3$ | $\infty$ | 4 | 6 | $\infty$ | $\infty$ |
| $v_4$ | 2 | $\infty$ | 5 | 0 | $\infty$ |
| $v_5$ | $\infty$ | $\infty$ | $\infty$ | 6 | 0 |

City A is Node 1, and City B is Node 4.

## Weather and Traffic Data:

**Weather Data:**

$$\text{weather\_data} = \{(1, 2) : 1; \quad (2, 4) : 3; \quad (1, 3) : 2; \quad (3, 4) : 1; \quad (4, 5) : 4\}$$

**Traffic Data:**

$$\text{traffic\_data} = \{(1, 2) : 2; \quad (2, 4) : 5; \quad (1, 3) : 3; \quad (3, 4) : 4; \quad (4, 5) : 2\}$$

## Shortest Path Result:

**Shortest distance from City 1 to City 4: 10 units**

**Paths from City 1 to City 4:**

1. City 1 → City 2 → City 4: Total Distance = 15 units
2. City 1 → City 2 → City 5: Total Distance = 19 units
3. City 1 → City 3 → City 2 → City 4: Total Distance = 26 units
4. City 1 → City 3 → City 2 → City 5: Total Distance = 30 units
5. City 1 → City 5: Total Distance = 10 units

## Code Explanation Step-by-Step:

1. **Importing `heapq`**: The function begins by importing the `heapq` module, which provides a heap-based priority queue. This structure helps efficiently select nodes with the smallest total weights in Dijkstra's algorithm.
2. **Function Parameters**:
   (a) `adjacency_matrix`: 2D matrix representing spatial distances between cities.

(b) `start`: Starting node.

(c) `spatio_temporal_data`: Dictionary to adjust weights based on time-dependent factors.

(d) `weather_data`: Dictionary assigning weights based on weather conditions.

(e) `traffic_data`: Dictionary assigning weights based on traffic intensity.

3. **Initialization**:

   (a) `num_nodes`: Total number of nodes in the graph.

   (b) `distances`: Dictionary of shortest distances, initialized to infinity except for the start node (0).

   (c) `priority_queue`: Heap queue initialized with the start node.

4. **Dijkstra's Algorithm Execution**:

   (a) Loop while priority queue is not empty.

   (b) Extract node with smallest weight.

   (c) Skip if its current path cost exceeds already found shortest path.

   (d) Iterate over neighbors and compute new total weight including spatial, temporal, weather, and traffic data.

   (e) Update neighbor's distance if new path is shorter, and push to priority queue.

5. **Return**: The function returns a dictionary containing the shortest distances from the start node to all other nodes.

## Code Example:

```
def get_successors(state):
    # Initialize an empty list to store valid successor states and their costs
    successors = []

    # Iterate through each cell in the 3x3 puzzle grid
    for i in range(3):
        for j in range(3):
            # Check if the current tile is the empty space (0)
            if state[i][j] == 0:
                # Explore all possible movements from the MOVES list
                for dx, dy in MOVES:
                    # Compute the new coordinates after moving
                    new_x, new_y = i + dx, j + dy

                    # Check if the new coordinates are within puzzle bounds
                    if is_valid_move(new_x, new_y):
                        # Determine move cost: diagonal = DIAGONAL_COST, else 1.0
                        cost = DIAGONAL_COST if (dx != 0 and dy != 0) else 1.0

                        # Make a deep copy of the state to avoid mutation
                        new_state = [list(row) for row in state]

                        # Swap the empty tile with the adjacent tile
                        new_state[i][j], new_state[new_x][new_y] = \
                            new_state[new_x][new_y], new_state[i][j]

                        # Convert the state to a hashable tuple and append with
                            ↪ cost
                        successors.append((tuple(map(tuple, new_state)), cost))

    # Return the list of all valid successor states and their move costs
    return successors
```

### 2.0.1 Dijkstra's Algorithm with Weather and Traffic Weights:

The following function integrates spatial distances, weather conditions, and traffic levels into the Dijkstra algorithm:

```
def dijkstra_with_weather_traffic(adjacency_matrix, start, weather_data,
    ↪ traffic_data):
    import heapq

    num_nodes = len(adjacency_matrix)
    distances = {node: float('inf') for node in range(1, num_nodes + 1)}
    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 in range(1, num_nodes + 1):
            if adjacency_matrix[current_node - 1][neighbor - 1] == float('inf'):
                continue

            spatial_distance = distances[current_node] + adjacency_matrix[
                ↪ current_node - 1][neighbor - 1]

            # Get weights from weather and traffic dictionaries
            weather_weight = weather_data.get((current_node, neighbor), 0)
            traffic_weight = traffic_data.get((current_node, neighbor), 0)

            # Combine all weights
            total_weight = spatial_distance + weather_weight + traffic_weight

            if total_weight < distances[neighbor]:
                distances[neighbor] = total_weight
                heapq.heappush(priority_queue, (total_weight, neighbor))

    return distances
```

## Recursive Path Finding Function:

The function below recursively explores all paths from a starting node to an end node without revisiting nodes. It helps in scenarios where you want to analyze all possible routing paths.

```
def find_all_paths(graph, start, end, path=[]):
    path = path + [(start, 0)]  # Append current node to path with dummy weight

    if start == end:
        return [path]

    if start not in graph:
        return []

    paths = []
    for node in graph[start]:
        if node not in [vertex for vertex, _ in path]:
            new_paths = find_all_paths(graph, node, end, path)
            for new_path in new_paths:
```

```
                paths.append(new_path)

    return paths
```

**Explanation of Function Logic:**

1. `path = path + [(start, 0)]`: Appends the current node and a dummy weight to track the route.
2. `if start == end`: Base case. If the current node is the destination, return the current path.
3. `if start not in graph`: If the start node has no outgoing edges, return an empty list.
4. `for node in graph[start]`: Iterate through all neighboring nodes.
5. `if node not in [vertex for vertex, _ in path]`: Prevent cycles by avoiding nodes already in the path.
6. Recursive call: `new_paths = find_all_paths(graph, node, end, path)` explores paths from the neighbor onward.
7. `paths.append(new_path)`: Append each valid new path to the result list.
8. `return paths`: After exploring all neighbors, return the collected paths.

### 2.0.2 Function to Find All Paths Between Two Nodes:

This function recursively searches all possible paths from a starting city to a destination, avoiding cycles.

```
def find_all_paths(graph, start, end, path=[]):
    path = path + [(start, 0)]
    if start == end:
        return [path]
    if start not in graph:
        return []
    paths = []
    for node in graph[start]:
        if node not in [vertex for vertex, _ in path]:
            new_paths = find_all_paths(graph, node, end, path)
            for new_path in new_paths:
                paths.append(new_path)
    return paths
```

### 2.0.3 Function to Print All Paths with Weather and Traffic Weights:

This function finds and prints all valid paths between two cities, computing associated metrics such as spatial distance, weather effects, and traffic congestion.

```
def find_and_print_path(graph, start, end, adjacency_matrix, weather_data,
    ↪ traffic_data):
    all_paths = find_all_paths(graph, start, end)
    if all_paths:
        print(f"Shortest distance from City {start} to City {end}: {
            ↪ shortest_distances[end]} units")
        print(f"All possible paths from City {start} to City {end}:")

        for path in all_paths:
            total_distance = 0
            total_weather = 0
            total_traffic = 0
            path_info = []
```

```
            for (i, _), (j, _) in zip(path, path[1:]):
                distance = adjacency_matrix[i - 1][j - 1]
                weather_info = weather_data.get((i, j), 0)
                traffic_info = traffic_data.get((i, j), 0)

                total_distance += distance
                total_weather += weather_info
                total_traffic += traffic_info

                path_info.append(f'City {i} ({distance} units, Weather: {
                    ↪ weather_info}, Traffic: {traffic_info})')

            city_units_string = ' -> '.join(path_info)
            print(f"Path: {city_units_string}, Unit distance: {total_distance},
                ↪ Weather: {total_weather}, Traffic: {total_traffic}, Total
                ↪ Distance: {total_distance + total_weather + total_traffic} units
                ↪ ")
    else:
        print(f"No paths found from City {start} to City {end}.")
```

**Main Execution Block:**

This block initializes the adjacency matrix and data dictionaries, calculates the shortest path using Dijkstra's algorithm with weather and traffic, and prints detailed information for each route.

```
# Example adjacency matrix
adjacency_matrix = [
    [0, 3, 8, float('inf'), 4],
    [float('inf'), 0, float('inf'), 1, 7],
    [float('inf'), 4, 6, float('inf'), float('inf')],
    [2, float('inf'), 5, 0, float('inf')],
    [float('inf'), float('inf'), float('inf'), 6, 0]
]

# Weather conditions
weather_data = {
    (1, 2): 1,
    (2, 4): 3,
    (1, 3): 2,
    (3, 4): 1,
    (4, 5): 4
}

# Traffic conditions
traffic_data = {
    (1, 2): 2,
    (2, 4): 5,
    (1, 3): 3,
    (3, 4): 4,
    (4, 5): 2
}

# Start and end nodes
start_node_city_A = 1
end_node_city_B = 4

# Compute the shortest path considering all weights
```

```
shortest_distances = dijkstra_with_weather_traffic(adjacency_matrix,
    ↪ start_node_city_A, weather_data, traffic_data)

# Check if the path exists
if shortest_distances[end_node_city_B] == float('inf'):
    print(f"No path exists from City {start_node_city_A} to City {end_node_city_B
        ↪ }.")
else:
    # Build a graph from an adjacency matrix
    graph = {}
    for i in range(len(adjacency_matrix)):
        for j in range(len(adjacency_matrix[i])):
            if adjacency_matrix[i][j] != float('inf'):
                If i + 1 is not in the graph:
                    graph[i + 1] = []
                graph[i + 1].append(j + 1)

    # Print all path info
    find_and_print_path(graph, start_node_city_A, end_node_city_B,
        ↪ adjacency_matrix, weather_data, traffic_data)
```

## Note

This paper presents a comprehensive framework for real-time route optimization that prioritizes both safety and efficiency by integrating spatio-temporal data, weather conditions, and traffic information into an enhanced Dijkstra-based algorithm. The approach demonstrates how intelligent data fusion and adaptive path recalculation can significantly improve travel outcomes in dynamic environments. Future work may involve extending this model to larger, real-world transportation networks, incorporating predictive analytics, and leveraging machine learning to further enhance decision-making under uncertainty.