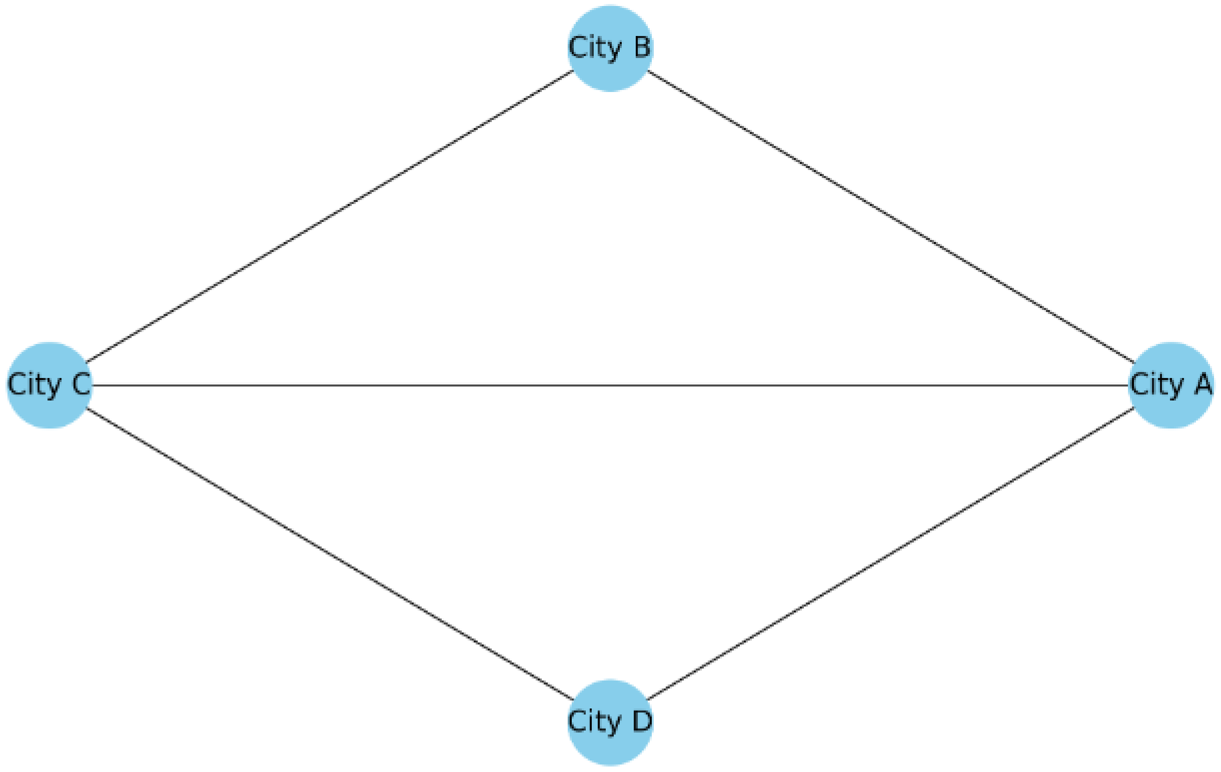
Dynamic Pathfinding with Real-time Data (DPRD) Algorithm

1. Understanding

In developing a novel algorithm aimed at identifying the most efficient route between two cities, the foundational step involves constructing a graphical representation that accurately models the network of cities and the pathways that connect them. This graphical model is composed of various nodes and edges, where each node represents a distinct city within the network, and each edge symbolizes the road or pathway linking these nodes. Visualize this as a network of roads interconnecting a series of cities.

For example:



2. Algorithm Development

Initial Thought process

In graphs, particularly when finding the shortest path between nodes, the challenge often lies in optimizing the route based on edge weights. These weights typically represent distances or costs associated with traversing from one node to another. Using algorithms like Dijkstra's, the objective is straightforward: to find the path that cumulatively incurs the minimum weight, thereby finding the shortest or least costly route between two points. Similarly, in the context of a road network, cities are considered as nodes, with the connecting roads as edges, and the distances between cities can be considered as edge weights.

However, the usual way of just using fixed distances to find paths doesn't fully grasp the changing nature of real-world travel. Traffic and weather conditions add complexity and can change a lot, greatly affecting which routes are best to take. Recognizing this issue led to the creation of a new idea: to develop an algorithm that is strong but also adaptable, capable of taking into account traffic and weather changes as it finds the shortest path.

3. Algorithm: Dynamic Pathfinding with Real-time Data

This new algorithm is designed to change the way we find the best routes through a complex network of cities. It's special because it can adjust to changes in traffic and weather as it calculates the best path to take, making sure the route it picks is not only short but also the best one under current conditions.

How It Works

At its core, this algorithm builds on traditional pathfinding methods but adds a key update: it changes the importance (weight) of each path based on what's happening right now with traffic and the weather. Instead of just looking at how long a road is, it also considers current traffic jams and weather conditions. This means it looks at the whole picture to find the best route, understanding that the shortest path might not always be the best choice if, for example, it's heavily raining there or there's a big traffic jam.

Adding Real-time Data

The way it brings in traffic and weather information is by constantly receiving updates about these conditions. This allows the algorithm to change the importance of roads on the fly. For instance, if a road usually quick to travel becomes slow due to heavy traffic or bad weather, the algorithm will know and can choose a different route. This makes sure that the path it picks is really the best one given the current situation.

The Benefits

What makes this algorithm stand out is how flexible and efficient it is. It quickly adapts to changes in traffic and weather, always looking for the best route. This is a big change in how we plan routes through cities, making travel faster, safer, and more reliable.

This new approach builds on what we already know about finding paths but makes it much better by considering the real-world changes that happen every day. It's a modern answer to the old problem of finding the best way to get from one place to another.

Data retrieval in the real-world

In the real world, we have various methods at our disposal to gather information about traffic and weather, crucial factors that can significantly influence travel times and route choices. Here's how we can access and utilize this data:

Accessing Traffic Data

Real-time Traffic Information: We can obtain up-to-the-minute traffic data from multiple sources. This includes GPS tracking systems installed in vehicles, data from traffic sensors spread across road networks, and Application Programming Interfaces (APIs) provided by popular mapping services like Google Maps or Waze. By integrating this data into our system, we gain access to live updates on traffic conditions, such as accidents, road closures, and areas of heavy congestion, enabling us to make informed decisions about route planning.

Gathering Weather Data

Weather Updates: Weather conditions play a significant role in travel, affecting everything from route accessibility to travel safety. We can collect weather data from meteorological services, utilize weather APIs (such as OpenWeatherMap or the National Weather Service), and analyze satellite imagery to understand current weather conditions. Incorporating this data allows us to account for various weather-related travel factors, including precipitation, snow, fog, and even extreme weather events like hurricanes or floods, ensuring that routes are optimized for safety and efficiency.

The Dynamic Pathfinding with Real-time Data algorithm presents an innovative approach to route optimization in dynamic environments. It extends the traditional Dijkstra's algorithm by incorporating real-time traffic and weather conditions into the cost (weight) calculations for each edge in the graph representing a road network.

X Factor of this Algorithm

Caching Mechanism

A pivotal enhancement to this algorithm is the integration of a caching mechanism. This mechanism stores the outcomes of path calculations associated with specific traffic and weather conditions by utilizing a unique hash value to represent each set of conditions. When a request for a path calculation is made, the algorithm first checks the cache using the current conditions' hash value. If a matching hash is found, indicating that a path for these exact conditions has been computed previously, the algorithm immediately retrieves and utilizes the cached path instead of recalculating it.

This approach not only significantly reduces the computational overhead for frequently requested routes under similar conditions but also ensures instantaneous response times for the end-users. By effectively reusing the results of past computations, the caching mechanism greatly enhances the efficiency and performance of the algorithm, making it exceptionally adept at handling the dynamic and often unpredictable variables of real-world travel.

4. Algorithm Implementation

The algorithm I developed, integrates dynamic traffic and weather conditions into the pathfinding process and utilizes a caching mechanism for efficiency, operates through several steps. Here's a detailed breakdown:

Step 1: Define Fixed Weights for Edges

- **Objective**: Establish a baseline for travel distances or times between cities (nodes) in the absence of traffic and weather considerations.

- **Implementation**: Assign fixed arbitrary weights to edges in the graph, representing the distance or base travel time between cities.

Step 2: Graph Construction

- **Objective**: Create a graphical representation of the network of cities and the roads (paths) connecting them.

- **Implementation**:

  - Initialize a graph `G`.

  - Add cities as nodes to the graph.

  - Add roads as edges between nodes with the predefined fixed weights.

Step 3: Real-time Data Integration

- **Objective**: Integrate dynamic traffic and weather data into the algorithm to reflect real-world conditions.

- **Implementation**:

  - Generate random traffic conditions for each edge and weather conditions for each node.

  - Utilize APIs or real-time data sources to obtain current traffic and weather information.

Step 4: Dynamic Weight Calculation

- **Objective**: Adjust the weights of edges dynamically based on current traffic and weather conditions.

- **Implementation**:

- Calculate dynamic weights for each edge, incorporating the effect of traffic levels and weather conditions using formula *f(node) = d(node) + max(w(start node), w(end node))+ t(node)*.

- Update the graph with these dynamic weights for pathfinding.

Step 5: Pathfinding with Caching

- **Objective**: Find the shortest path between two cities, leveraging cached results to enhance efficiency.

- **Implementation**:

  - Generate a unique hash value representing the current traffic and weather conditions.

  - Check if a path for these conditions is already cached:

    - If so, retrieve and use the cached path.

    - If not, proceed to calculate the path using Dijkstra's algorithm with dynamic weights.

  - Cache the new path along with its conditions hash for future use.

Step 6: Caching Mechanism

- **Objective**: Store previously calculated paths and their associated conditions to avoid recalculating for similar future queries.

- **Implementation**:

  - Maintain a cache (`path_cache`) where each entry maps a conditions hash to a corresponding path.

  - Before path calculation, check the cache using the current conditions hash.

  - After calculating a new path, add it to the cache with the associated hash.

Step 7: Display and Use Cached Paths

- **Objective**: Utilize the caching mechanism to efficiently retrieve paths for repeated or similar queries.

- **Implementation**:

- Display existing paths in the cache to illustrate the caching mechanism's effectiveness.

- For a new pathfinding query, first check the cache using the generated hash of current conditions.

- Use the cached path if available or calculate and cache a new path if necessary.

Conclusion:

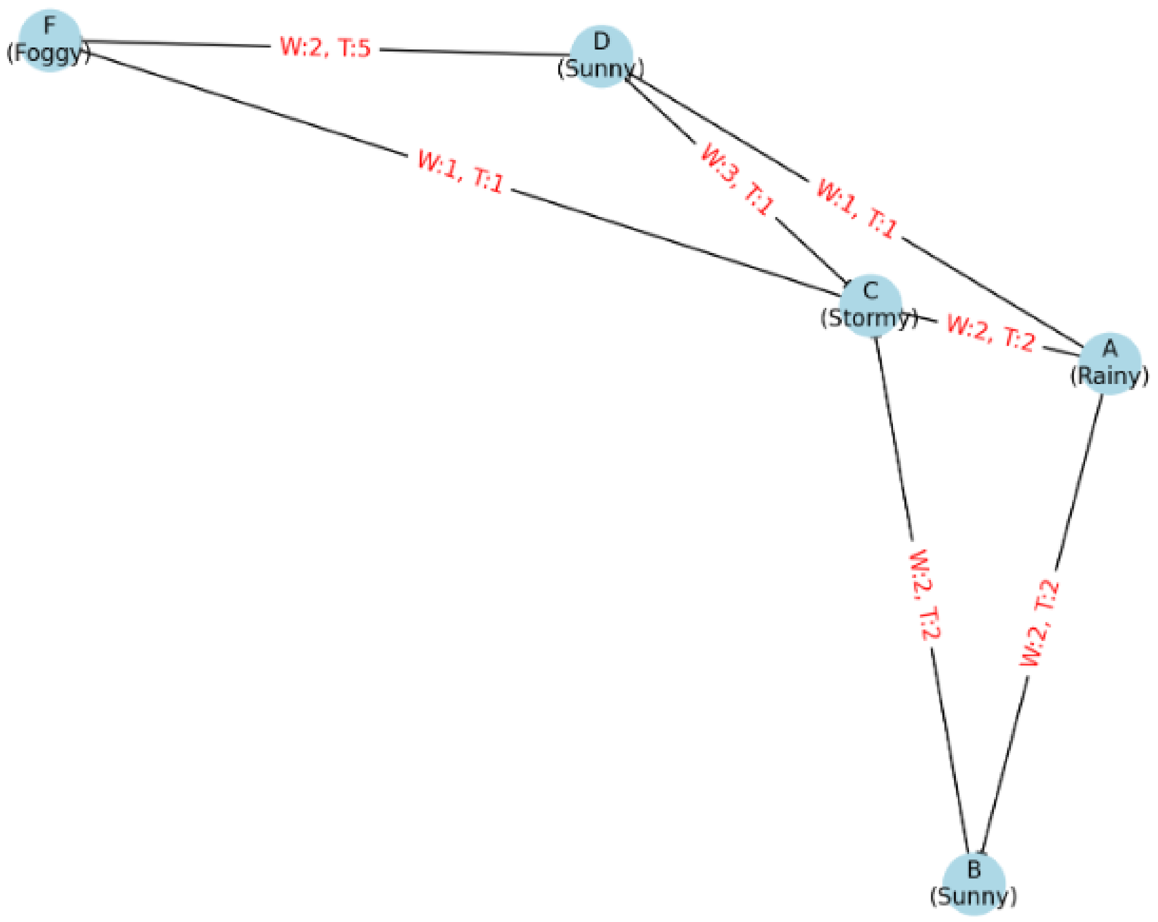
This algorithm, through its innovative integration of dynamic conditions and efficient use of caching, significantly improves the responsiveness and accuracy of pathfinding in a network of cities. By adapting to real-time traffic and weather data, it ensures that the chosen routes are not only short but also practical and optimized for current conditions, showcasing a significant advancement in route optimization technology.

5. Algorithm Dry Run on an example Graph

Let's imagine a Graph, consisting of six cities with pre-defined distance, weather and traffic conditions as follows.

Source city	**Destination city**	**Road Distance**	**Traffic**
A (Rainy)	B (Sunny)	2	2
B (Sunny)	C (Stormy)	2	2
C (Stormy)	F (Foggy)	4	4
A (Rainy)	D (Sunny)	1	1
D (Sunny)	C (Stormy)	3	1
C (Stormy)	F (Foggy)	1	1
A (Rainy)	C (Stormy)	2	2
D (Sunny)	F (Foggy)	2	5

Now Let's find the shortest distance between City A (Source) and the destination city (B)



In the graph modeling the road network between cities, travel costs between any two cities are affected by weather and traffic conditions. Each condition is categorized into levels that uniquely impact travel cost as a percentage increase over the base distance cost. Below is a concise overview of how these conditions modify travel costs.

Here weather conditions are divided in to 5 different categories and in each of weather condition impose different impact on the cost required to travel from one city to an other city. Below are the percentage increase in cost based on weather category.

Weather condition	Percentage Increase (Impact)
Sunny	0%
Cloudy	5%
Rainy	10%
Foggy	15%
Stormy	20%

Similarly traffic conditions are divided in to 5 different levels and each level of traffic imposes different impact on existing cost. Below are the percentage increase in the cost based on level of the traffic at the city.

Traffic Level	Percentage Increase (Impact)
Level 0	0%
Level 1	5%
Level 2	10%
Level 3	20%
Level 4	40%
Level 5	80%

All Paths from City A to City F:

Path 1: $A \rightarrow B \rightarrow C \rightarrow F$

Path 2: $A \rightarrow D \rightarrow C \rightarrow F$

Path 3: $A \rightarrow D \rightarrow F$

Path 4: $A \rightarrow C \rightarrow F$

Each path represents a potential route from City A to City F, navigating through the interconnected network of cities under varying weather and traffic conditions.

In our shortest path calculation for a graph representing a network of cities, the total cost to reach a node, denoted as f(node), combines three key components: distance traveled, weather conditions impact, and traffic conditions impact. Formally, the cost calculation can be expressed as:

f(node) = d(node) + max(w(start node), w(end node))+ t(node) where:

- *f(node)* represents the total cost to reach the node.
- *d(node)* is the edge weight or the distance traveled to reach the node.
- *w(start node)* quantifies the additional cost due to weather conditions of start node in an edge.
- *w(end node)* quantifies the additional cost due to weather conditions of end node in an edge.
- *t(node)* quantifies the additional cost due to traffic conditions.

Let's calculate the shortest path from City A to City F, considering the options to travel from City A to Cities B, C, and D initially.

From A, Lets first go to City B, hence cost incur to reach City B is

$f(B) = d(B) + max(w(A), w(B)) + t(B) + existing\ cost$

$f(B) = 2 + max(w(Rainy), w(Sunny)) + t(Level\ 2) + 0$

$f(B) = 2 + (2 * 0.1) + (2*0.1) + 0$

$f(B) = 2 + 0.4$

$f(B) = 2.4$

Now from A, Lets go to City C, hence cost incur to reach City C is

$f(C) = d(C) + max(w(A), w(C)) + t(C) + existing\ cost$

$f(C) = 2 + max(w(Rainy), w(Stormy)) + t(Level\ 2) + 0$

$f(C) = 2 + (2 * 0.2) + (2*0.1) + 0$

$f(C) = 2 + 0.6 + 0$

$f(C) = 2.6$

Now from A, Lets first go to City D, hence cost incur to reach City D

$f(D) = d(D) + max(w(A), w(D)) + t(D) + existing\ cost$

$f(D) = 1 + max(w(Rainy), w(Sunny)) + t(Level\ 1) + 0$

$f(D) = 1 + (1 * 0.1) + (1*0.05) + 0$

$f(D) = 1 + 0.15$

$f(D) = 1.15$

Here $f(D) < f(B)$ and $f(C)$

Given these calculations, moving from City A to City D incurs the least cost. So, the optimal initial move is to City D.

Now from City D, I can either go to City C or City F.

From city D, Lets go to city C

$f(C) = d(C) + max(w(D), w(C))+ t(C) + existing\ cost$

$f(C) = 3 + max(w(Sunny), w(stormy))+ t(Level\ 1)+1.15$

$f(C) = 3+ (3 * 0.2) + (3*0.05)+1.15$

$f(C) = 3+ 1.90$

$f(C) = 4.90$

Now from D, Lets first go to City F,

$f(F) = d(F) + max(w(D), w(F)) + t(F) + existing\ cost$

$f(F) = 2 + max(w(Sunny), w(Foggy))+ t(Level\ 5)+1.15$

$f(F) = 2+ (2 * 0.15) + (2*0.8)+1.15$

$f(F) = 2+ 3.05$

$f(F) = 5.05$

Here $f(C) < f(F)$

Given these calculations, moving from City D to City C incurs the least cost. So, the optimal initial move is to City C.

Now from city C, I can either go to city F or city D. Here city D is already visited, hence its better to go city F which is our destination

$$f(F) = d(F) + max(w(C), w(F)) + t(F) + \text{existing cost}$$

$$f(F) = 1 + max(w(Stormy), w(Foggy)) + t(Level\ 1) + 4.90$$

$$f(F) = 1 + (1*0.2) + (1*0.05) + 4.90$$

$$f(F) = 1 + 5.15$$

$$f(F) = 6.15$$

Hence the shortest path here is **A -> D -> C -> F** which took the cost of **6.15.**

In future applications of this algorithm under identical conditions, there's no need to recalculate the shortest path. The algorithm optimizes efficiency by storing the current shortest path, generated for specific weather and traffic conditions, in a cache along with a unique hash value. This hash value encapsulates the specifics of these conditions, acting as a key to retrieve the stored path.

Should the same conditions occur again, instead of undergoing the computation process anew, the algorithm will simply retrieve the shortest path from the cache using the hash value. This method significantly enhances the algorithm's efficiency by reducing computational overhead and ensuring rapid access to the optimal route under recurring conditions.

Let us look at the working of shortest path algorithm including the caching mechanism with the code implementation.

6. Prototype of Algorithm Implemented on the Graph Representing Georgia's City Network

I have developed a graph representing the state of Georgia, where the nodes symbolize the cities within the state. Initially, I assigned specific weights to the connections between these cities, representing distance between any two cities These initial weights serve as a foundational aspect of the graph, providing a basic structure for analyzing travel routes across Georgia's cities.

To enhance the practicality and relevance of the graph, I integrated dynamic factors such as real-time traffic delay and weather conditions, which influence the weights of the connections between cities. This integration allows the graph to reflect the current state of travel conditions more accurately, making it an invaluable tool for route optimization and planning. This traffic and weather conditions are randomly generated for the purpose of this implementation.

Furthermore, to streamline the process of determining the most efficient paths under varying conditions, I implemented a caching mechanism. This mechanism stores the results of previous path calculations along with a unique hash value representing the specific traffic and weather conditions at the time of calculation. Should identical conditions recur, the algorithm can quickly retrieve the optimal path from the cache, bypassing the need for recalculating and significantly reducing computational overhead.

In summary, the graph I developed for the state of Georgia, enriched with dynamic weighting and caching capabilities, stands as a sophisticated model for navigation and route planning. It is designed not only to depict the static geographical layout of the state's cities but also to accommodate the fluid and unpredictable nature of travel conditions, thereby offering a robust solution for real-time route optimization.

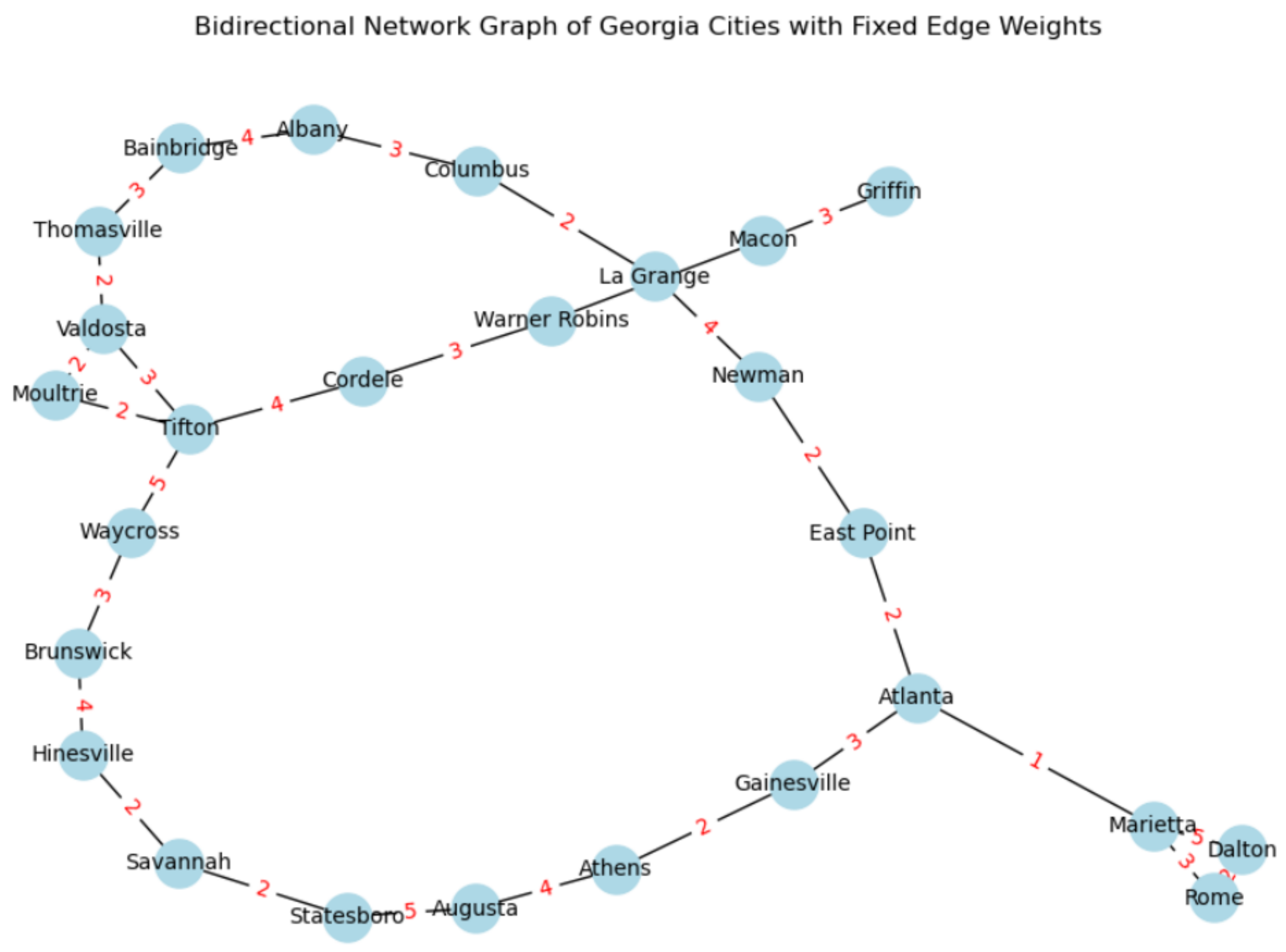
Code Implementation

Initial Graph Generation with Fixed Weights

```
# Since we need fixed weights to simulate distances, we'll assign fixed arbitrary weights for simplicity.

# In a real-world scenario, these would be based on actual distances.

# Define fixed weights for edges

fixed_weights = {

    ('Dalton', 'Rome'): 2,

    ('Dalton', 'Marietta'): 5,

    ('Rome', 'Marietta'): 3,

    ('Marietta', 'Atlanta'): 1,

    ('Atlanta', 'East Point'): 2,

    ('Atlanta', 'Gainesville'): 3,

    ('East Point', 'Newman'): 2,

    ('Newman', 'La Grange'): 4,

    ('La Grange', 'Columbus'): 2,

    ('Columbus', 'Albany'): 3,

    ('Albany', 'Bainbridge'): 4,

    ('Gainesville', 'Athens'): 2,

    ('Athens', 'Augusta'): 4,

    ('Augusta', 'Statesboro'): 5,

    ('Statesboro', 'Savannah'): 2,

    ('Griffin', 'Macon'): 3,

    ('Macon', 'Warner Robins'): 2,

    ('Warner Robins', 'Cordele'): 3,

    ('Cordele', 'Tifton'): 4,
    ('Tifton', 'Valdosta'): 3,
    ('Valdosta', 'Thomasville'): 4,
    ('Thomasville', 'Bainbridge'): 3,
    ('Moultrie', 'Tifton'): 2,
    ('Moultrie', 'Valdosta'): 2,
    ('Waycross', 'Brunswick'): 3,
    ('Brunswick', 'Hinesville'): 4,
    ('Hinesville', 'Savannah'): 2,
    ('Waycross', 'Tifton'): 5,
}

# Create a graph again
G = nx.Graph()

# Add cities as nodes to the graph
for city in cities:
    G.add_node(city)

# Add edges with fixed weights
for (city1, city2), weight in fixed_weights.items():
    G.add_edge(city1, city2, weight=weight)



# Draw the graph with arrows to represent bidirectional edges
plt.figure(figsize=(12, 8))
# Use a spring layout again for better visualization
pos = nx.spring_layout(G)
# We use nx.draw_networkx to have more control over the drawing parameters
```

```
nx.draw_networkx_nodes(G, pos, node_size=500, node_color="lightblue")
nx.draw_networkx_labels(G, pos, font_size=10)
nx.draw_networkx_edges(G, pos, arrows=True)  # Adding arrows to edges
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')

plt.title("Bidirectional Network Graph of Georgia Cities with Fixed Edge Weights")
plt.axis('off')  # Hide the axis for a cleaner look
plt.show()
```

Output



Integrating Weather and Traffic conditions into the Graph

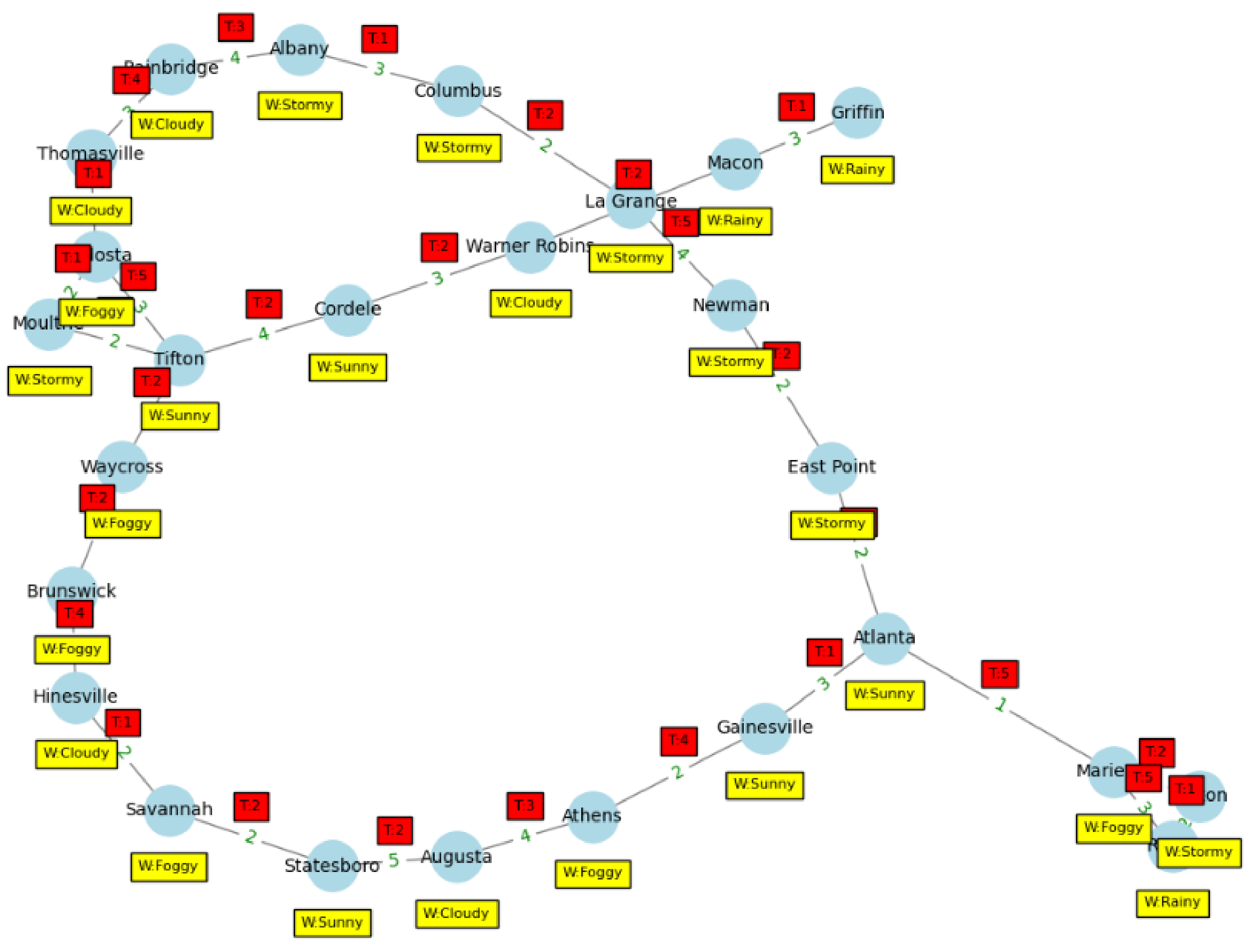
```
def draw_graph_with_random_conditions_and_return_conditions():
    # Generate new random traffic conditions for all edges
    new_traffic_conditions = {edge: random.randint(1, 5) for edge in G.edges()}

    # Generate new random weather conditions for all nodes
    new_weather_conditions = {node: random.choice(['Sunny', 'Rainy', 'Cloudy', 'Stormy', 'Foggy']) for node in G.nodes()}

    # Draw the graph with these new conditions
    plt.figure(figsize=(14, 10))
    nx.draw_networkx_nodes(G, pos, node_size=800, node_color="lightblue")
    nx.draw_networkx_labels(G, pos, font_size=10)
    nx.draw_networkx_edges(G, pos, edge_color='grey')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=fixed_weights, font_color='green')

    # Adjust the placement of new traffic condition annotations
    for edge in G.edges():
        edge_center = [(pos[edge[0]][0] + pos[edge[1]][0]) / 2, (pos[edge[0]][1] + pos[edge[1]][1]) / 2]
        traffic_annotation_pos = (edge_center[0], edge_center[1] + 0.05)
        plt.text(traffic_annotation_pos[0], traffic_annotation_pos[1], f"T:{new_traffic_conditions[edge]}",
                 horizontalalignment='center', verticalalignment='bottom',
                 bbox=dict(facecolor='red', alpha=1), fontsize=8)

    # Adjust the placement of new weather condition annotations
    for node in G.nodes():
        weather_annotation_pos = (pos[node][0], pos[node][1] - 0.1)
        plt.text(weather_annotation_pos[0],                                   weather_annotation_pos[1],
f"W:{new_weather_conditions[node]}",
                 horizontalalignment='center', verticalalignment='top',
                 bbox=dict(facecolor='yellow', alpha=1), fontsize=8)

    plt.title("Georgia Cities Network with Random Traffic and Weather Data")
    plt.axis('off')
    plt.show()

    hash_value = generate_conditions_hash(new_traffic_conditions, new_weather_conditions)

    # Return the generated conditions for use in path finding
    return new_traffic_conditions, new_weather_conditions, hash_value
```

Generating Hash value for the Current Weather and Traffic conditions

```
def generate_conditions_hash(traffic_conditions, weather_data):
    # Create a unique string representation of current conditions
    conditions_str = str(traffic_conditions) + str(weather_data)
    # Generate a hash from this string
    conditions_hash = hashlib.sha256(conditions_str.encode()).hexdigest()
    return conditions_hash
```

Output

Georgia Cities Network with Random Traffic and Weather Data



Dynamically calculating Edge Weight with Weather and Traffic conditions

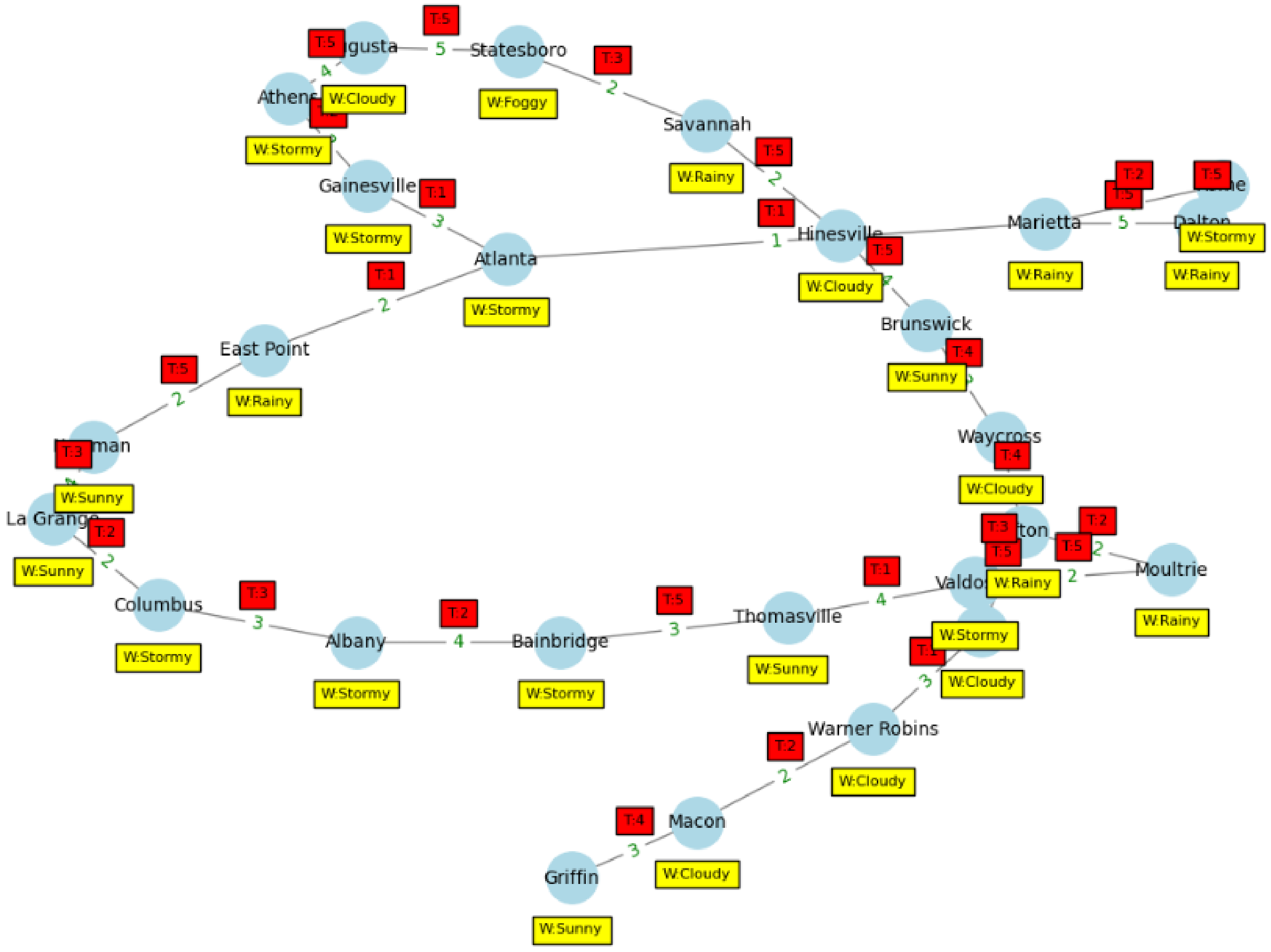
```
def calculate_dynamic_weight(u, v, traffic_conditions, weather_data):
    """
    Calculate the dynamic weight for an edge based on refined traffic and weather conditions.
    """
    base_weight = fixed_weights.get((u, v), fixed_weights.get((v, u)))  # Bidirectional support
    traffic_level = traffic_conditions.get((u, v), traffic_conditions.get((v, u), 1))
    weather_condition_u = weather_data.get(u, "Sunny")
    weather_condition_v = weather_data.get(v, "Sunny")

    # Traffic impact: Define a non-linear impact scale for traffic levels
    traffic_impact_scale = {1: 0.05, 2: 0.1, 3: 0.2, 4: 0.4, 5: 0.8}
    traffic_impact = base_weight * traffic_impact_scale[traffic_level]

    # Weather impact: Define specific impacts for different weather conditions
    weather_impact_values = {'Sunny': 0, 'Cloudy': 0.05, 'Rainy': 0.1, 'Stormy': 0.2, 'Foggy': 0.15}
    weather_impact_u = weather_impact_values[weather_condition_u]
    weather_impact_v = weather_impact_values[weather_condition_v]
    weather_impact = base_weight * max(weather_impact_u, weather_impact_v)  # Take the higher
impact of the two

    # Calculate dynamic weight
    dynamic_weight = base_weight + traffic_impact + weather_impact
    return dynamic_weight
```

Implementing Dynamic Pathfinding with Real-time Data

```
def dijkstra_manual(G, source, target, traffic_conditions, weather_data):
    """
    Dijkstra's algorithm for finding the shortest path, considering dynamic edge weights.
    """
    Q = []  # Priority queue of (cost, node)
    parents = {node: None for node in G.nodes}  # Parent nodes for path reconstruction
    costs = {node: float('inf') for node in G.nodes}  # Start with infinity costs
    costs[source] = 0  # Cost from source to source is 0
    heapq.heappush(Q, (0, source))  # Push source node into priority queue

    while Q:
        current_cost, current_node = heapq.heappop(Q)
        if current_node == target:
            break  # Stop if target is reached

        for neighbor in G.neighbors(current_node):
            weight = calculate_dynamic_weight(current_node, neighbor, traffic_conditions, weather_data)
            new_cost = current_cost + weight

            if new_cost < costs[neighbor]:
                costs[neighbor] = new_cost
                parents[neighbor] = current_node
                heapq.heappush(Q, (new_cost, neighbor))
    # Reconstruct the shortest path
    path = []
    current = target
    while current is not None:
        path.append(current)
        current = parents[current]
    path.reverse()  # Reverse the path to get the correct order from source to target

    return path if path[0] == source else []  # Return the path if it's valid

new_traffic_conditions, new_weather_conditions, hash_value =
draw_graph_with_random_conditions_and_return_conditions()

# Initialize cache for storing paths with their conditions hash
path_cache = {}

def find_or_cache_path(source, destination, conditions_hash, traffic_conditions, weather_data):
    # Check cache for existing path under these conditions
    if conditions_hash in path_cache:
        print("Using cached path for hash :", conditions_hash)
        return path_cache[conditions_hash]
    else:
        # Calculate the path if not found in cache
        path = dijkstra_manual(G, source_city, target_city, traffic_conditions, weather_data)  #
Assuming this function exists
        # Cache this new path with the conditions hash
        path_cache[conditions_hash] = path
        print("Calculated and cached new path for hash :", conditions_hash)
        return path
```

Output Execution

State of Georgia Graph – Version 1



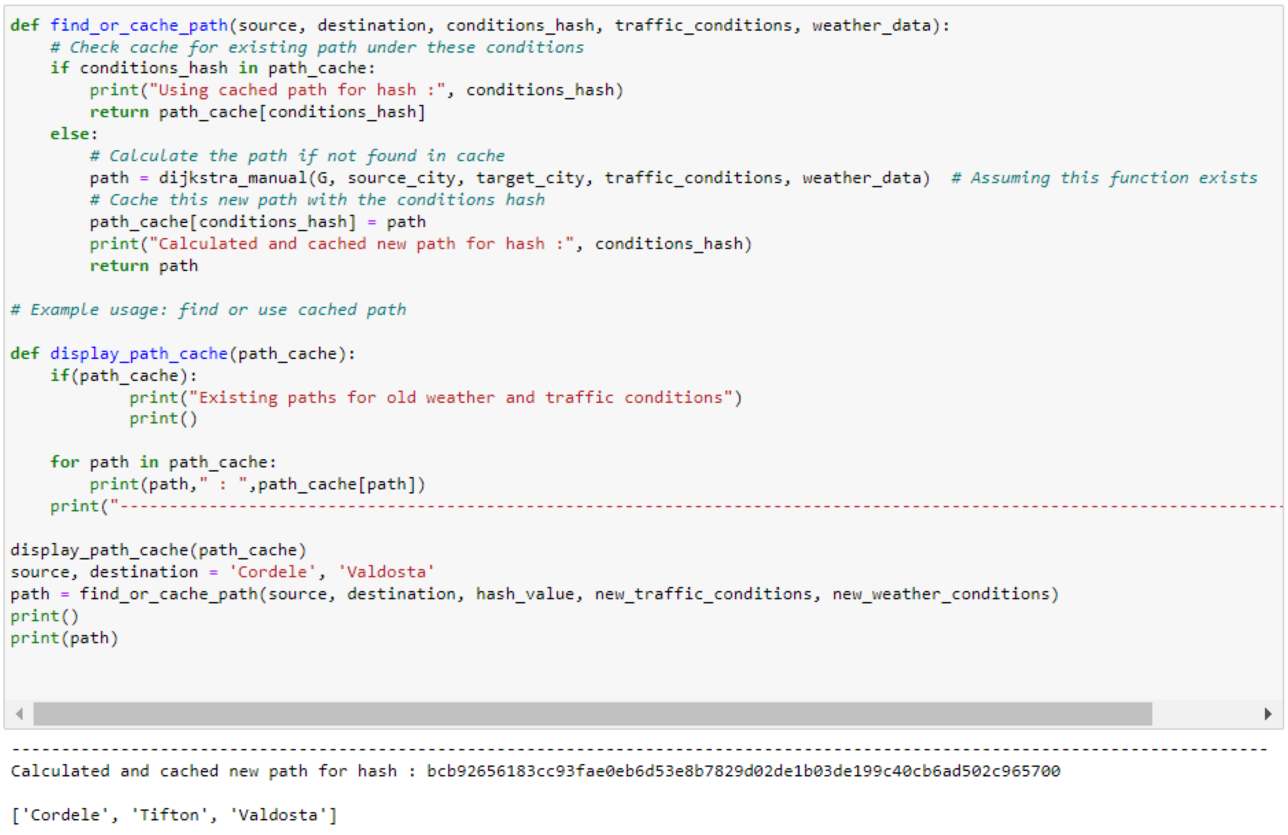
Now, let's determine the shortest path between Cordele and Valdosta. Presently, there are two potential routes connecting these cities:

Path 1: Cordele -> Tifton -> Valdosta.

Path 2: Cordele -> Tifton -> Moultrie -> Valdosta.

First execution call for the shortest path between Cordele and Valdosta

During the initial execution, the algorithm calculated the shortest path between Cordele and Valdosta, identifying the optimal route as ***Cordele -> Tifton -> Valdosta.***

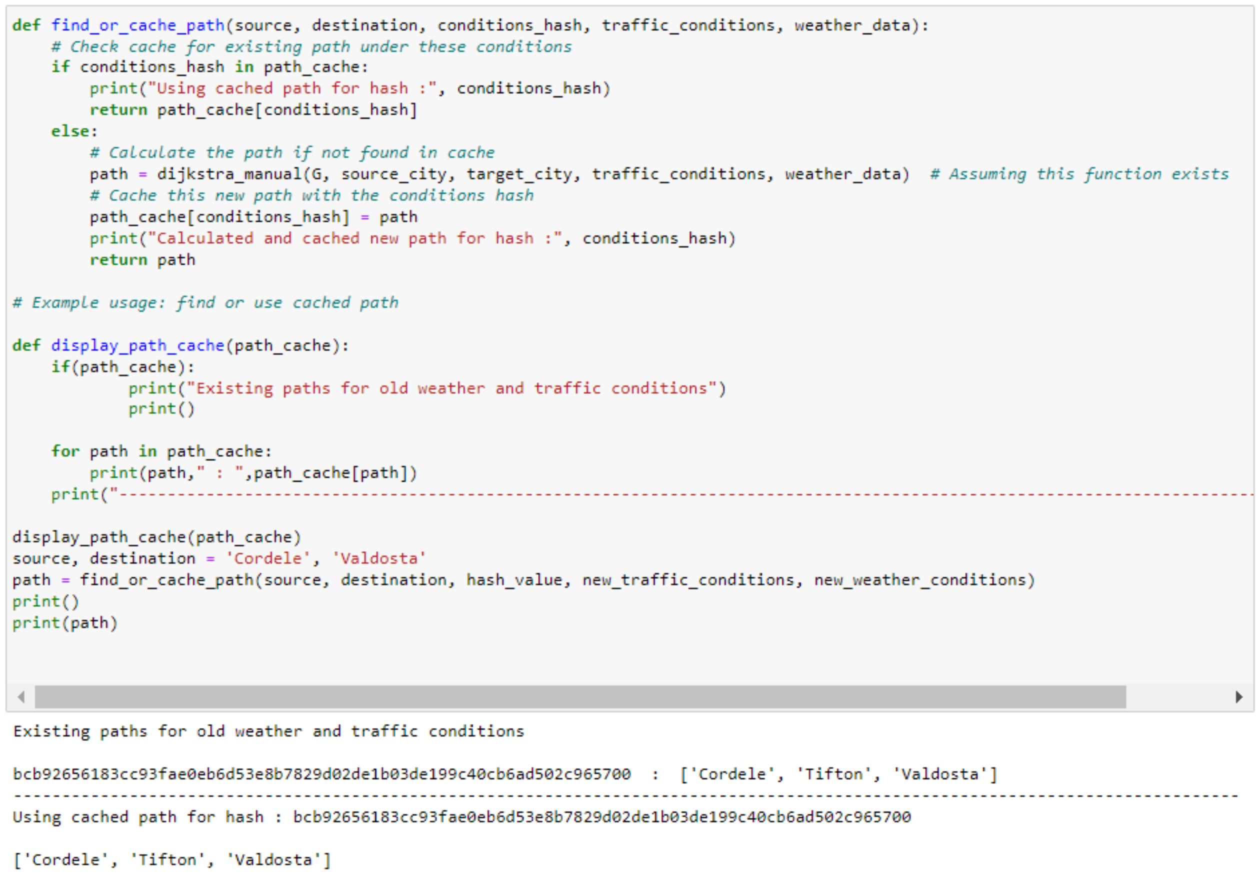


```python
def find_or_cache_path(source, destination, conditions_hash, traffic_conditions, weather_data):
    # Check cache for existing path under these conditions
    if conditions_hash in path_cache:
        print("Using cached path for hash :", conditions_hash)
        return path_cache[conditions_hash]
    else:
        # Calculate the path if not found in cache
        path = dijkstra_manual(G, source_city, target_city, traffic_conditions, weather_data)  # Assuming this function exists
        # Cache this new path with the conditions hash
        path_cache[conditions_hash] = path
        print("Calculated and cached new path for hash :", conditions_hash)
        return path

# Example usage: find or use cached path

def display_path_cache(path_cache):
    if(path_cache):
            print("Existing paths for old weather and traffic conditions")
            print()

    for path in path_cache:
        print(path," : ",path_cache[path])
    print("---------------------------------------------------------------------------------------------------------------

display_path_cache(path_cache)
source, destination = 'Cordele', 'Valdosta'
path = find_or_cache_path(source, destination, hash_value, new_traffic_conditions, new_weather_conditions)
print()
print(path)
```

```
--------------------------------------------------------------------------------------------------------------------
Calculated and cached new path for hash : bcb92656183cc93fae0eb6d53e8b7829d02de1b03de199c40cb6ad502c965700

['Cordele', 'Tifton', 'Valdosta']
```

Second execution call for the shortest path between Cordele and Valdosta with similar weather and traffic conditions.

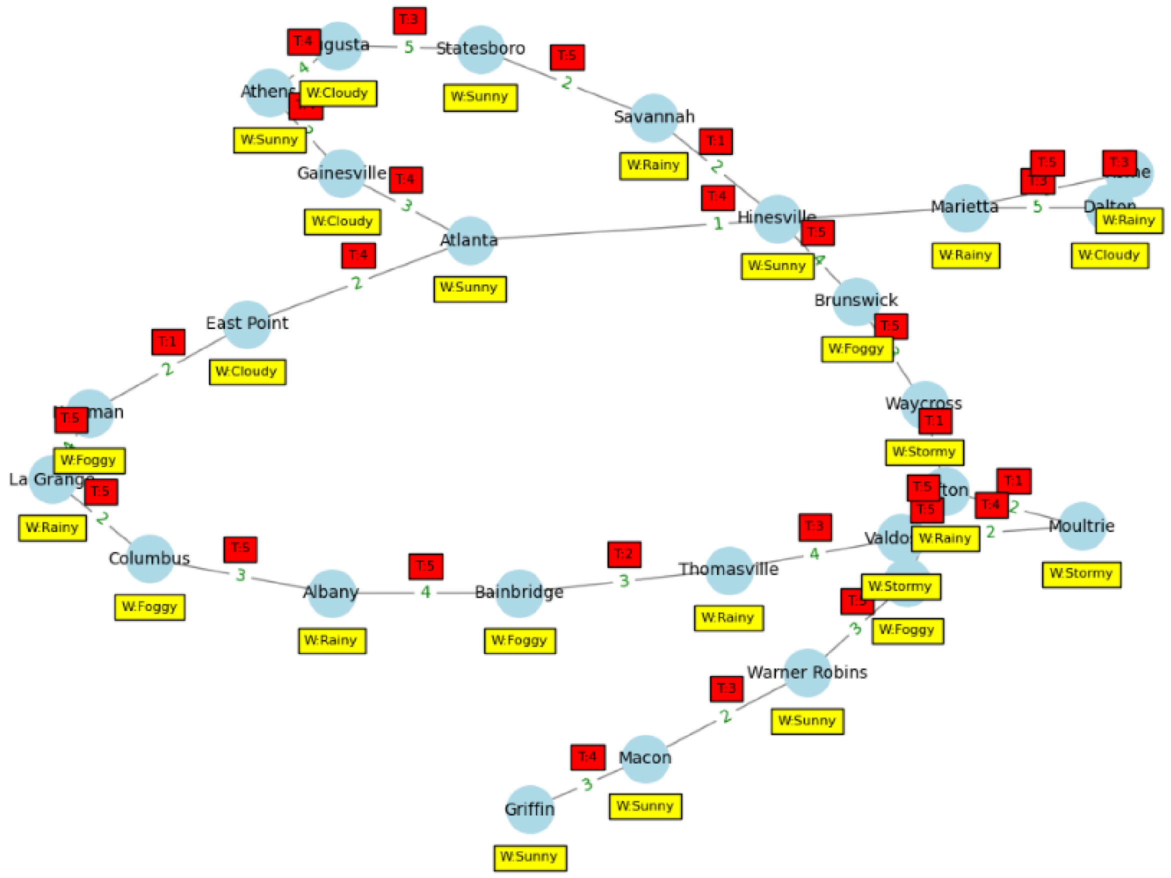
During the second execution, the algorithm efficiently retrieved the shortest path between Cordele and Valdosta from the cache, bypassing the need for recalculating it from scratch. This optimized process is evident in the displayed cache, where the path is associated with a unique hash value used as the key.



```python
def find_or_cache_path(source, destination, conditions_hash, traffic_conditions, weather_data):
    # Check cache for existing path under these conditions
    if conditions_hash in path_cache:
        print("Using cached path for hash :", conditions_hash)
        return path_cache[conditions_hash]
    else:
        # Calculate the path if not found in cache
        path = dijkstra_manual(G, source_city, target_city, traffic_conditions, weather_data)  # Assuming this function exists
        # Cache this new path with the conditions hash
        path_cache[conditions_hash] = path
        print("Calculated and cached new path for hash :", conditions_hash)
        return path

# Example usage: find or use cached path

def display_path_cache(path_cache):
    if(path_cache):
            print("Existing paths for old weather and traffic conditions")
            print()

    for path in path_cache:
        print(path," : ",path_cache[path])
    print("-------------------------------------------------------------------------------------------------------------------

display_path_cache(path_cache)
source, destination = 'Cordele', 'Valdosta'
path = find_or_cache_path(source, destination, hash_value, new_traffic_conditions, new_weather_conditions)
print()
print(path)
```

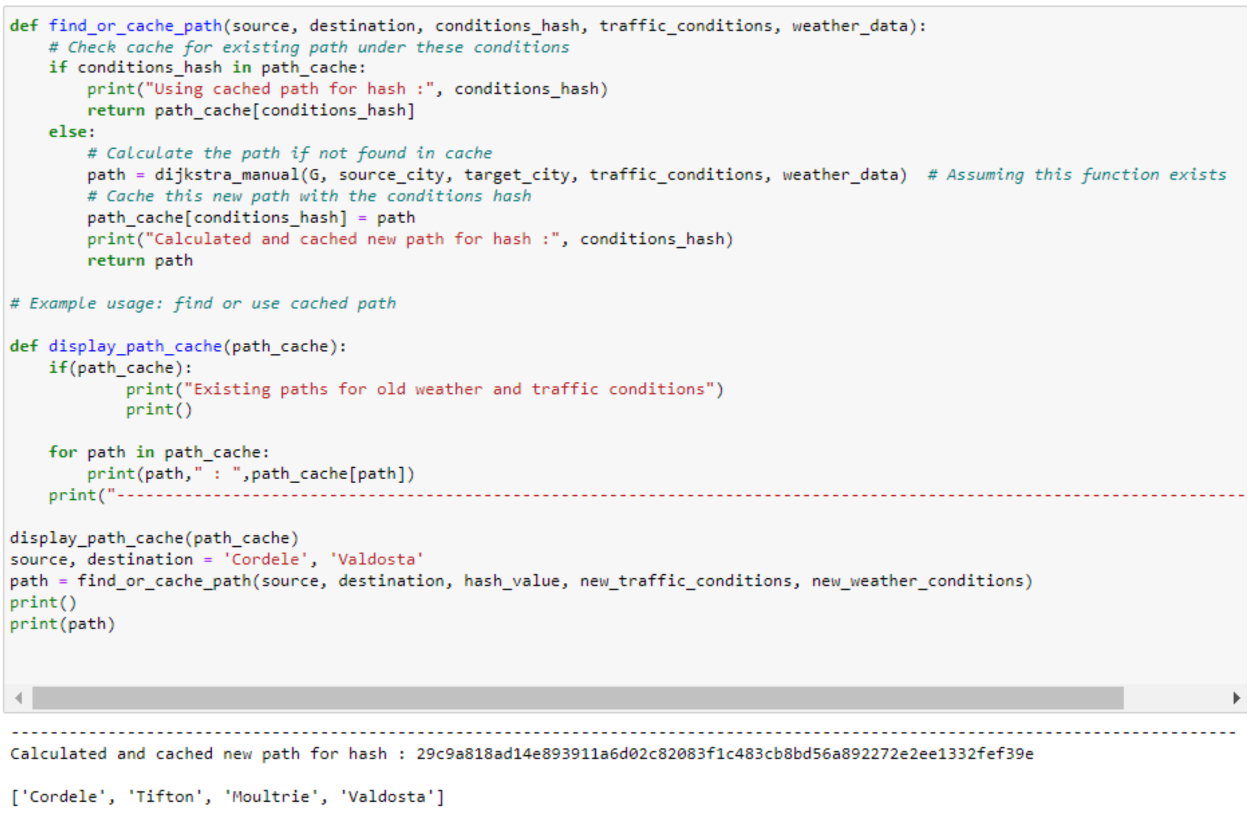
```
Existing paths for old weather and traffic conditions

bcb92656183cc93fae0eb6d53e8b7829d02de1b03de199c40cb6ad502c965700  :  ['Cordele', 'Tifton', 'Valdosta']
-----------------------------------------------------------------------------------------------------------------------------
Using cached path for hash : bcb92656183cc93fae0eb6d53e8b7829d02de1b03de199c40cb6ad502c965700

['Cordele', 'Tifton', 'Valdosta']
```

Graph with new weather and traffic conditions.



Third execution call for the shortest path between Cordele and Valdosta with new weather and traffic conditions.

During this execution, the algorithm calculated the shortest path between Cordele and Valdosta, identifying the optimal route as ***Cordele -> Tifton -> Moultrie -> Valdosta.***



```python
def find_or_cache_path(source, destination, conditions_hash, traffic_conditions, weather_data):
    # Check cache for existing path under these conditions
    if conditions_hash in path_cache:
        print("Using cached path for hash :", conditions_hash)
        return path_cache[conditions_hash]
    else:
        # Calculate the path if not found in cache
        path = dijkstra_manual(G, source_city, target_city, traffic_conditions, weather_data)  # Assuming this function exists
        # Cache this new path with the conditions hash
        path_cache[conditions_hash] = path
        print("Calculated and cached new path for hash :", conditions_hash)
        return path

# Example usage: find or use cached path

def display_path_cache(path_cache):
    if(path_cache):
            print("Existing paths for old weather and traffic conditions")
            print()

    for path in path_cache:
        print(path," : ",path_cache[path])
    print("-----------------------------------------------------------------------------------------------------------------

display_path_cache(path_cache)
source, destination = 'Cordele', 'Valdosta'
path = find_or_cache_path(source, destination, hash_value, new_traffic_conditions, new_weather_conditions)
print()
print(path)
```

```
--------------------------------------------------------------------------------------------------------------------------------
Calculated and cached new path for hash : 29c9a818ad14e893911a6d02c82083f1c483cb8bd56a892272e2ee1332fef39e

['Cordele', 'Tifton', 'Moultrie', 'Valdosta']
```