Delhi Metro Navigation System

Project Overview - This project simulates a fully functional Delhi Metro Navigation System using C++. It combines multiple Data Structures and Algorithms (DSA) concepts, Object-Oriented Programming (OOP), and real-world logic to allow users to interact with a virtual metro system. The system handles station navigation, route-finding, real-time delay simulation, and even user-friendly features like station name auto-completion.

Core Features & Functionalities-

1. Object-Oriented Graph Implementation

- The entire metro system is modeled using a Graph class.
- Each station is represented as a vertex, and connections between them (with weights) as edges.
- The Vertex struct within the class stores neighboring stations along with their respective distances or times.
- OOP principles ensure modularity, encapsulation, and code reuse.

2. Shortest Distance & Time Calculation

- **Dijkstra's Algorithm** is used to calculate the shortest path between two stations.
- The same algorithm is adapted for both:
    - **Shortest Distance (KM)** – Edge weights are based on physical distances.
    - **Shortest Time (minutes)** – Edge weights include fixed inter-station timings and transfer delays.
- A priority_queue with a custom comparator ensures efficient retrieval of the minimum-cost path.

3. Real-Time Delay Propagation

- Implements **Breadth-First Search (BFS)** using a queue to simulate delays spreading through the network.

- When a station has a delay, the delay propagates to its neighbors with an additional delay (e.g., +3 minutes per level).

- Delay is capped to affect only nearby stations (within a limited number of hops) to mimic real-world containment.

4. Station Auto-Completion

(i). Prefix Matching

First, the input is matched with the beginning (prefix) of each station name.

- For example, typing "Raj" can instantly match to "Rajiv Chowk~BY" because it starts with "Raj".

- This is **fast and accurate** when the user knows the starting letters of the station.

(ii). Fuzzy Matching (Levenshtein Distance)

If no prefix match is found, or the match is ambiguous, the system uses **Levenshtein Distance**, which measures how many single-character edits (insert, delete, substitute) are needed to convert one string into another.

- This helps with **typos or partial name mismatches**.

- For example, input "neta subh" can still match "Netaji Subhash Place~PR" even though the spelling is incomplete or wrong.

- The algorithm calculates the distance between the input and each station name and suggests the **closest** matches.

5. Fare Calculation Logic

- The fare is computed based on distance slabs:
    - ₹10 for <=2 KM, ₹20 for <=5 KM, … up to ₹60.

- This replicates real-world Delhi Metro fare structure, making the system feel intuitive and useful.

6. Interactive CLI Interface

- The system is menu-driven with user prompts for various operations:
    - Listing stations
    - Viewing the full metro map
    - Calculating shortest distance or time
    - Simulating delays
    - Exiting the application
- Each input is processed with validation and appropriate user feedback.

Dijkstra's Algorithm

Dijkstra's algorithm is a fundamental graph traversal technique used in this project to determine the shortest path between two metro stations. It operates by assigning a tentative distance value to every station in the network — initially set to infinity — except for the source station, which is assigned a distance of zero. The algorithm maintains a priority queue to always process the station with the currently known shortest distance. From this station, it explores all its directly connected neighbors, updating their tentative distances if a shorter path is found through the current station. In the context of the metro system, these distances can either represent physical distances in kilometers or travel times in minutes, depending on whether the user opts for a shortest-distance or shortest-time calculation. Each time a better (shorter or faster) route to a neighboring station is discovered, that route is updated in the queue. This process continues until the destination station is reached or all reachable stations are processed. The algorithm ensures that the optimal path is always selected based on the weights (distance or time) associated with each metro connection. In the system, the shortest path is also stored as a readable string to display the full route, making it user-friendly while remaining computationally efficient.

Breadth-First Search (BFS) for Delay Propagation

Breadth-First Search (BFS) is used in this project to simulate **real-time delay propagation** from a delayed station to its nearby stations. The BFS algorithm begins from the affected (delayed) station and explores all its neighbors level-by-level, ensuring that the delay spreads logically, mimicking real-world behavior. Each neighbor experiences a slight increase in delay (e.g., +3 minutes per hop), and this continues for a limited number of hops (e.g., 3 levels deep) to prevent the delay from unrealistically affecting distant stations. A queue is used to track the current station, accumulated delay, and level of propagation. By

using a combination of unordered_map for visited nodes and delay times, and a queue for managing traversal, BFS efficiently simulates how disruptions in one part of the network ripple out to nearby nodes, providing a dynamic and realistic model of metro operations.

Object-Oriented Programming (OOP) Design

The project is built around the principles of **Object-Oriented Programming (OOP)** to ensure modularity, reusability, and scalability. The central component is the Graph class, which encapsulates all metro station data and operations. Each metro station is treated as a Vertex, and connections between them are modeled as edges using adjacency maps (unordered_map). By using classes, the system separates responsibilities cleanly—for example, adding stations, linking routes, calculating shortest paths, and simulating delays are all implemented as **member functions** of the graph. This structure promotes clean code organization, where internal state like station connections or delay propagation data is safely encapsulated. The use of **constructors**, **encapsulation**, and **member functions** highlights classic OOP features. This design choice not only enhances readability and maintainability but also makes it easier to extend the project—such as by adding new metro lines, integrating live data, or switching to more advanced data structures like heaps or tries—without disturbing the existing functionality.

Concept	**Usage**
unordered_map	Station adjacency and delay mapping
vector	Storing graph edges, station list
priority_queue	Dijkstra’s shortest path
queue	BFS for delay propagation
string & sort	Auto-completion and path printing

tuple	Used for BFS node metadata (station, delay, level)

🔧 Design Decisions

- **Modularity**: The Graph class abstracts all graph-related operations like adding vertices, adding edges, Dijkstra's traversal, and simulating delay.
- **Scalability**: Though designed for Delhi Metro, the structure allows easy scaling to other cities or expanded maps.
- **Error Handling**: The system gracefully handles invalid station inputs, station not found, or empty results from autocomplete.

📝 How to Use

1. Compile the C++ code using any standard compiler (e.g., g++).
2. Run the executable. You'll see a welcome screen with a list of options.
3. Choose operations from the menu by typing numbers (1–6).
4. For route-based features, you can enter full or partial station names.
5. View route, delay impact, and fare accordingly.

🔍 Sample Metro Stations Used

- Blue Line: Noida Sector 62, Botanical Garden, Rajiv Chowk
- Yellow Line: Huda City Center, AIIMS, Vishwavidyalaya
- Orange Line: IGI Airport, DDS Campus
- Pink & Red Line interchanges (Punjabi Bagh, Rajouri Garden)

Now the c++ code

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <climits>
#include <string>
#include <algorithm>
using namespace std;
class Graph {
public:
   struct Vertex {
        unordered_map<string, int> nbrs;
    };
    unordered_map<string, Vertex> vtces;
    void addVertex(const string& name) {
        vtces[name] = Vertex();
    }
    void addEdge(const string& v1, const string& v2, int weight) {
        if (!vtces.count(v1) || !vtces.count(v2) || vtces[v1].nbrs.count(v2)) return;
        vtces[v1].nbrs[v2] = weight;
        vtces[v2].nbrs[v1] = weight;
    }
    struct DijkstraPair {
        string vname, psf;
        int cost;
        bool operator>(const DijkstraPair& o) const {
            return cost > o.cost;
        }
    };
    pair<int, string> dijkstra(const string& src, const string& dest, bool timeMode = false) {
        unordered_map<string, int> dist;
        priority_queue<DijkstraPair, vector<DijkstraPair>, greater<>> pq;
        for (const auto& p : vtces) dist[p.first] = INT_MAX;
        dist[src] = 0;
        pq.push({src, src, 0});
        while (!pq.empty()) {
            auto cur = pq.top(); pq.pop();
            if (cur.vname == dest)
                return {cur.cost, cur.psf};
            for (const auto& nbr : vtces[cur.vname].nbrs) {
                int wt = timeMode ? 120 + 40 * nbr.second : nbr.second;
                int newCost = cur.cost + wt;
                if (newCost < dist[nbr.first]) {
                    dist[nbr.first] = newCost;
                    pq.push({nbr.first, cur.psf + " -> " + nbr.first, newCost});
                }
            }
        }
        return {-1, ""};
    }
    void displayStations() {
        int i = 1;
        cout << "\nAvailable Metro Stations:\n";
        for (auto& p : vtces) cout << i++ << ". " << p.first << '\n';
        cout << endl;
    }
```

```
    void displayMap() {
        cout << "\nDelhi Metro Map:\n";
        for (auto& p : vtces) {
            cout << p.first << " -> ";
            for (auto& nbr : p.second.nbrs) cout << nbr.first << "(" << nbr.second << ") ";
            cout << endl;
        }
    }
    void simulateDelay(const string& station, int delayMinutes) {
        if (!vtces.count(station)) {
            cout << "\U0001F6AB Station not found!\n";
            return;
        }
        unordered_map<string, int> delayTime;
        unordered_map<string, bool> visited;
        queue<tuple<string, int, int>> q;
        int maxHops = 3;
        q.push({station, delayMinutes, 0});
        delayTime[station] = delayMinutes;
        visited[station] = true;
        cout << "\n\U0001F534 Real-Time Delay Propagation from: " << station << " (+" << delayMinutes << "
mins)" << endl;
        while (!q.empty()) {
            auto [curStation, curDelay, level] = q.front(); q.pop();
            if (level >= maxHops) continue;
            for (const auto& nbr : vtces[curStation].nbrs) {
                string neighbor = nbr.first;
                int newDelay = curDelay + 3;
                if (!delayTime.count(neighbor) || newDelay > delayTime[neighbor]) {
                    delayTime[neighbor] = newDelay;
                    q.push({neighbor, newDelay, level + 1});
                }
            }
        }
        cout << "\n\U0001F4CD Affected Stations:\n";
        for (const auto& [st, d] : delayTime) {
            if (st != station)
                cout << " → " << st << " : +" << (d - delayMinutes) << " min (Total: " << d << " mins)\n";
        }
        cout << endl;
    }
};
vector<string> stationList;
void buildStationList(Graph& g) {
    stationList.clear();
    for (const auto& p : g.vtces)
        stationList.push_back(p.first);
    sort(stationList.begin(), stationList.end());
}
int levenshtein(const string& a, const string& b) {
    int n = (int)a.size(), m = (int)b.size();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1));
    for (int i = 0; i <= n; ++i) dp[i][0] = i;
    for (int j = 0; j <= m; ++j) dp[0][j] = j;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (tolower(a[i - 1]) == tolower(b[j - 1]))
                dp[i][j] = dp[i - 1][j - 1];
```

```
            else
                dp[i][j] = 1 + min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]});
        }
    }
    return dp[n][m];
}
string autoCompleteStation(const string& input) {
    string partial = input;
    transform(partial.begin(), partial.end(), partial.begin(), ::tolower);
    vector<pair<int, string>> candidates;
    for (const auto& name : stationList) {
        string lowerName = name;
        transform(lowerName.begin(), lowerName.end(), lowerName.begin(), ::tolower);
        if (lowerName.find(partial) == 0)
            candidates.emplace_back(0, name);
        else
            candidates.emplace_back(levenshtein(partial, lowerName), name);
    }
    sort(candidates.begin(), candidates.end());
    vector<string> matches;
    for (int i = 0; i < min(5, (int)candidates.size()); ++i) {
        matches.push_back(candidates[i].second);
    }
    if (matches.empty()) {
        cout << "No station matches your input. Try again.\n";
        return "";
    } else if (matches.size() == 1) {
        cout << "Auto-completed to: " << matches[0] << endl;
        return matches[0];
    } else {
        cout << "Did you mean:\n";
        for (int i = 0; i < matches.size(); ++i)
            cout << i + 1 << ". " << matches[i] << endl;
        int choice;
        cout << "Enter choice (1-" << matches.size() << "): ";
        cin >> choice; cin.ignore();
        if (choice >= 1 && choice <= matches.size()) return matches[choice - 1];
        else {
            cout << "Invalid choice.\n";
            return "";
        }
    }
}
int calculateFare(int distance) {
    if (distance <= 2) return 10;
    else if (distance <= 5) return 20;
    else if (distance <= 12) return 30;
    else if (distance <= 21) return 40;
    else if (distance <= 32) return 50;
    else return 60;
}
void createMetroMap(Graph& g) {
    g.addVertex("Noida Sector 62~B");
    g.addVertex("Botanical Garden~B");
    g.addVertex("Yamuna Bank~B");
    g.addVertex("Rajiv Chowk~BY");
    g.addVertex("Vaishali~B");
    g.addVertex("Moti Nagar~B");
```

```
    g.addVertex("Janak Puri West~BO");
    g.addVertex("Dwarka Sector 21~B");
    g.addVertex("Huda City Center~Y");
    g.addVertex("Saket~Y");
    g.addVertex("Vishwavidyalaya~Y");
    g.addVertex("Chandni Chowk~Y");
    g.addVertex("New Delhi~YO");
    g.addVertex("AIIMS~Y");
    g.addVertex("Shivaji Stadium~O");
    g.addVertex("DDS Campus~O");
    g.addVertex("IGI Airport~O");
    g.addVertex("Rajouri Garden~BP");
    g.addVertex("Netaji Subhash Place~PR");
    g.addVertex("Punjabi Bagh West~P");
    g.addEdge("Noida Sector 62~B", "Botanical Garden~B", 8);
    g.addEdge("Botanical Garden~B", "Yamuna Bank~B", 10);
    g.addEdge("Yamuna Bank~B", "Vaishali~B", 8);
    g.addEdge("Yamuna Bank~B", "Rajiv Chowk~BY", 6);
    g.addEdge("Rajiv Chowk~BY", "Moti Nagar~B", 9);
    g.addEdge("Moti Nagar~B", "Janak Puri West~BO", 7);
    g.addEdge("Janak Puri West~BO", "Dwarka Sector 21~B", 6);
    g.addEdge("Huda City Center~Y", "Saket~Y", 15);
    g.addEdge("Saket~Y", "AIIMS~Y", 6);
    g.addEdge("AIIMS~Y", "Rajiv Chowk~BY", 7);
    g.addEdge("Rajiv Chowk~BY", "New Delhi~YO", 1);
    g.addEdge("New Delhi~YO", "Chandni Chowk~Y", 2);
    g.addEdge("Chandni Chowk~Y", "Vishwavidyalaya~Y", 5);
    g.addEdge("New Delhi~YO", "Shivaji Stadium~O", 2);
    g.addEdge("Shivaji Stadium~O", "DDS Campus~O", 7);
    g.addEdge("DDS Campus~O", "IGI Airport~O", 8);
    g.addEdge("Moti Nagar~B", "Rajouri Garden~BP", 2);
    g.addEdge("Punjabi Bagh West~P", "Rajouri Garden~BP", 2);
    g.addEdge("Punjabi Bagh West~P", "Netaji Subhash Place~PR", 3);
}
int main() {
    Graph g;
    createMetroMap(g);
    buildStationList(g);
    int choice;
    string src, dest, input;
    cout << "\n**** WELCOME TO THE DELHI METRO APP ****\n";
    do {
        cout << "\n1. List All Stations\n2. Show Metro Map\n3. Get Shortest Distance"
             << "\n4. Get Shortest Time\n5. Simulate Delay Propagation\n6. Exit\nEnter your choice: ";
        cin >> choice;
        cin.ignore();
        switch (choice) {
            case 1:
                g.displayStations();
                break;
            case 2:
                g.displayMap();
                break;
            case 3:
                cout << "Enter source (partial/full): "; getline(cin, input);
                src = autoCompleteStation(input);
                if (src == "") break;
                cout << "Enter destination (partial/full): "; getline(cin, input);
```

```
            dest = autoCompleteStation(input);
            if (dest == "") break;
            {
                auto res = g.dijkstra(src, dest, false);
                if (res.first == -1) cout << "No path found.\n";
                else {
                    cout << "Shortest Distance: " << res.first << " KM\n";
                    cout << "Estimated Fare: Rs " << calculateFare(res.first) << endl;
                    cout << "Path: " << res.second << endl;
                }
            }
            break;
        case 4:
            cout << "Enter source (partial/full): "; getline(cin, input);
            src = autoCompleteStation(input);
            if (src == "") break;
            cout << "Enter destination (partial/full): "; getline(cin, input);
            dest = autoCompleteStation(input);
            if (dest == "") break;
            {
                auto res = g.dijkstra(src, dest, true);
                if (res.first == -1) cout << "No path found.\n";
                else {
                    cout << "Shortest Time: " << res.first / 60 << " minutes\n";
                    cout << "Path: " << res.second << endl;
                }
            }
            break;
        case 5:
            g.displayStations();
            cout << "Enter station with delay (partial/full): ";
            getline(cin, input);
            src = autoCompleteStation(input);
            if (src == "") break;
            int delay;
            cout << "Enter delay in minutes: ";
            cin >> delay; cin.ignore();
            g.simulateDelay(src, delay);
            break;
        case 6:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid option. Try again.\n";
            break;
    }
  } while (choice != 6);
  return 0;
}
```