**Project summary in 500 words**

This project is a powerful "Delhi Metro Assistant," a command-line application (CLI) designed to help users navigate the Delhi Metro network. It's built in Python and uses the delhi\_metro\_edges.csv file as its database. The script combines data parsing, advanced graph theory, and a polished user interface to provide a range of useful features.

**Core Concept: Multiple Graph Models**

At its heart, the project represents the metro network as a **graph**, where stations are **nodes (vertices)** and the tracks between them are **edges (links)** with a "weight" equal to their distance.

The script's most elegant feature is that it doesn't just build one graph; it builds **three** different ones to answer different questions:

1. **A Simple Dictionary Graph (self.graph):** This is a basic Python dictionary used for internal lookups, storing the *full* station names (e.g., kashmere gate\_(interchange red yellow violet)) to understand line details.
2. **A NetworkX Graph for Distance (self.nx\_graph):** This graph is built for speed and simplicity. It uses *simple* station names (e.g., "kashmere gate") as nodes. The edge weight is the **distance in kilometers**. This graph is used to find the shortest physical path.
3. **A Specialized Graph for Interchanges (self.g\_interchange):** This is the most clever part of the project. To find the path with the *fewest line changes*, it builds a complex graph where a "node" is a **(station, line)** tuple.
   * **Travel:** Traveling between two stations on the *same line* (e.g., ("saket", "yellow") to ("malviya nagar", "yellow")) has a low weight of **1**.
   * **Interchange:** Switching lines *at the same station* (e.g., from ("hauz khas", "yellow") to ("hauz khas", "magenta")) has a high penalty weight of **100**.

When Dijkstra's algorithm finds the "shortest" path on this graph, it will avoid the 100-point penalty at all costs, preferring to travel many more stops (each costing 1) if it means avoiding an interchange.

**Key Features**

1. **Shortest Path (by Distance):** Using the distance graph (nx\_graph), it finds the route with the minimum total kilometers.
2. **Minimum Interchange Path:** Using the specialized penalty graph (g\_interchange), it finds the route with the fewest line changes, which is often more convenient for passengers.
3. **Fare Calculation:** For any path, it calculates the total distance and uses a predefined list of fare slabs to estimate the journey cost.
4. **Turn-by-Turn Directions:** It generates human-readable directions, telling the user which line to start on, where to travel, and explicitly when to "Change at..." to a new line.
5. **Full Map Visualization:** It uses matplotlib and networkx to generate and save a high-resolution PNG image of the entire metro network, with lines color-coded and interchange stations highlighted.
6. **Journey Visualization:** After finding a path, the user can visualize *that specific journey*. It opens a pop-up window showing the full map faded in the background, with the user's path drawn brightly on top.
7. **User-Friendly Interface:** The CLI is enhanced with colorama (for colored text) and rich (for beautiful tables), and includes a helpful feature to list all stations with numbers, allowing users to input either a station's name or its number.

**Project Overview**

**Main Purpose**

The Delhi Metro Assistant is a **command-line interface (CLI) application** built in Python. Its primary goal is to help users navigate the Delhi Metro network by:

1. Finding the **shortest path** between two stations based on distance.
2. Finding the **easiest path** by minimizing the number of line changes (interchanges).
3. Providing detailed, human-readable **directions**, **fare estimations**, and **distance** for the calculated route.
4. **Visualizing** the entire metro map and specific user journeys.

**Core Technologies**

* **Programming Language:** Python
* **Core Libraries:**
  + **networkx:** The primary library for creating, managing, and analyzing the graph data structures.
  + **matplotlib:** Used to generate and save visualizations of the metro network and user journeys.
  + **csv:** Used for reading and parsing the initial data from the .csv file.
  + **rich & colorama:** Used to create a polished, modern, and colorful user interface in the command line (e.g., for menus and tables).

**The Data & Graph Model**

**Role of delhi\_metro\_edges.csv**

This file is the **database** for the entire project. It functions as an **edge list**, which is a simple way to represent a graph.

* **Structure:** Each row contains station\_u, station\_v, and distance\_km.
* **Meaning:** It defines a single **edge** (a track) between two **adjacent** stations (u and v) and specifies its **weight** (the distance).

**Graph Concepts**

* **Graph:** A data structure representing a network. It consists of nodes and edges.
* **Nodes (Vertices):** In this project, the **metro stations** are the nodes.
* **Edges (Links):** The **metro tracks** connecting two *adjacent* stations.
* **Edge Weight:** A value assigned to an edge. Here, it represents the **distance in kilometers** between the two stations.
* **Graph Type:** The project uses an **undirected graph**. A track from Station A to Station B can also be used to travel from B to A. A directed graph (which has one-way paths) would not accurately model a metro system.

**The Core Logic: A Three-Graph Strategy**

The project's central design decision is to use **three different graph representations** simultaneously. This is done because we need to optimize for two different goals (distance vs. interchanges) that cannot be solved by a single, simple graph.

**1. self.nx\_graph (For Shortest Distance)**

* **Purpose:** To find the path with the minimum total kilometers.
* **Nodes:** Simple, normalized station names (e.g., "hauz khas").
* **Edges:** Weighted by **distance (km)** (e.g., 1.8).
* **Algorithm:** When a user asks for the shortest path, **Dijkstra's algorithm** is run on this graph using distance as the weight.

**2. self.g\_interchange (For Minimum Interchanges)**

This is the most complex and clever part of the project.

* **Purpose:** To find the path with the fewest line changes.
* **Nodes:** A node is a **tuple** of (station\_name, line\_color) (e.g., ("kashmere gate", "red"), ("kashmere gate", "yellow"), etc.). A single interchange station is thus represented by multiple nodes, one for each line it serves.
* **Edges (Two Types):**
  1. **Travel Edges:** Connect adjacent stations *on the same line* (e.g., ("hauz khas", "yellow") to ("green park", "yellow")). These have a very low, uniform weight of **1**.
  2. **Interchange Edges:** Connect different lines *at the same station* (e.g., ("kashmere gate", "red") to ("kashmere gate", "yellow")). These have a very high **penalty weight of 100**.
* **How it Works:**
  1. When Dijkstra's algorithm runs on this graph, it seeks the lowest total "cost".
  2. The algorithm will always prefer to add 1 + 1 + 1 (traveling three more stops) rather than pay the 100 (making one interchange).
  3. It will only "pay" the 100-point penalty if it's absolutely necessary to change lines to reach the destination.
  4. A final path cost of **207** clearly indicates **2 interchanges** (2 \* 100) and **7 station hops** (7 \* 1).

**3. self.graph (For Utility & Lookups)**

* **Purpose:** A simple Python dictionary used as a utility for data lookups.
* **Structure:** Maps the *full* station name (e.g., "kashmere gate\_(interchange red yellow violet)") to its neighbors.
* **Use Case:** Used by \_parse\_station\_lines to get the line information for a station.

**Pathfinding Algorithms**

**Dijkstra's Algorithm**

* **What it is:** A classic and highly efficient algorithm for finding the **shortest path** between nodes in a **weighted graph**.
* **Why Dijkstra?**
  + **vs. BFS (Breadth-First Search):** BFS finds the path with the *fewest edges* (fewest stops), but it **cannot** process weights. It would treat a 10km track as "1 stop" and a 0.5km track as "1 stop," failing to find the shortest *distance*.
  + **vs. DFS (Depth-First Search):** DFS is a search algorithm that finds *a* path, but it provides no guarantee that the path is the shortest or most optimal.
* **Implementation:** The project *does not* manually implement Dijkstra. It leverages the highly optimized version built into the **networkx library** (nx.dijkstra\_path and nx.dijkstra\_path\_length). This is faster, less error-prone, and more robust.
* **Time Complexity:** The networkx implementation uses a priority queue, giving an efficient time complexity of $O(E + V \log V)$ or $O(E \log V)$, where $V$ is the number of stations and $E$ is the number of tracks.

**A\* (A-star) Algorithm**

* **How it's different:** A\* is an *informed* search algorithm (an "upgrade" to Dijkstra). It uses a **heuristic** (an educated guess) to explore paths that are already heading in the correct physical direction.
* **Advantage:** If we had the latitude/longitude for each station, we could use the "as-the-crow-flies" distance as a heuristic. This would make A\* *faster* than Dijkstra's because it wouldn't waste time exploring paths that are clearly moving away from the destination.

**Implementation Details & Features**

**Data Loading and Handling**

* **Reading CSV:** The csv library is used to read the file row by row. It's configured to skip any lines that start with # (comments).
* **Station Normalization:**
  + User input is normalized using .strip().lower() to handle case sensitivity and extra spaces.
  + To allow input by name or number, two mappings are created:
    1. self.station\_index\_map: e.g., "hauz khas" -> 53
    2. self.index\_station\_map: e.g., 53 -> "hauz khas"
* **Error Handling:** The get\_station\_from\_input function checks if the user's input (name or number) exists in its mapping. If not, it prints an error and returns None, preventing the program from crashing.

**Feature Breakdown**

* **Fare Calculation:**
  + The self.fare\_slabs list stores a **static** (hard-coded) set of distance-to-fare rules.
  + After finding a path, the system gets the total distance (e.g., 14.5 km).
  + The \_calculate\_fare function iterates through the slabs until it finds the correct bracket (e.g., "Up to 12-21 km is Rs. 43") and returns that fare.
* **CLI User Experience:**
  + colorama and rich are used purely for aesthetics.
  + They make the menu clear, print results in color (e.g., line names in magenta, interchanges in yellow), and display the fare slabs in a beautifully formatted table.
* **Map Visualization (visualize\_map)**
  1. A matplotlib.pyplot figure is created.
  2. nx.kamada\_kawai\_layout is used to position the station nodes in a clean, non-overlapping way.
  3. A hard-coded dictionary line\_colors\_map assigns a specific hex color to each line name (e.g., 'red': '#e51d25').
  4. nx.draw\_networkx\_edges is called once for *each line*, drawing only that line's edges in its specific color.
  5. nx.draw\_networkx\_nodes is called to draw the stations. Interchange nodes are made larger and black to stand out.
  6. The final plot is saved as a high-resolution PNG.
* **Journey Visualization (visualize\_journey)**
  1. It performs all the steps of visualize\_map, but with one change: all nodes and edges are drawn with a high transparency (alpha=0.2), making them look "faded."
  2. It retrieves the self.last\_path (which was stored from the user's last search).
  3. It re-draws *only* the nodes and edges from that path on top, with alpha=1.0 (fully opaque), bright colors, and thick lines.
  4. Instead of saving, it uses plt.show() to open an interactive pop-up window.

**Scalability & Future Improvements**

**Project Bottlenecks**

* **Startup Time:** The main bottleneck is the one-time graph creation process at the start, especially \_build\_interchange\_graph. For a massive network, this could take a few seconds.
  + **Optimization:** This could be optimized by **caching** the built networkx graphs. We could use pickle to save the g\_interchange and nx\_graph objects to a file. On next startup, the script would check if the .csv has changed. If not, it would load the pre-built graphs from the pickle file, making the startup almost instantaneous.

**Adapting and Scaling**

* **Other Metro Systems:** The system is highly adaptable. To make it work for the Mumbai Metro, you would only need to:
  1. Create a mumbai\_metro\_edges.csv in the same format.
  2. Update the line\_colors\_map with Mumbai's line names and colors.
* **Scalability:** For a system with thousands of stations (e.g., all of India's metros), a CSV is not ideal.
  1. **Database:** A **graph database** like **Neo4j** would be the perfect solution. Neo4j is *designed* to store and query graph relationships and would be extremely fast, even at a massive scale.
  2. **Concurrency:** The current script is a single-user CLI app, so concurrency is not an issue. If this were a web server, the graphs would be loaded into memory once, and multiple user requests could query them (read-only) in parallel without conflict.

**Potential New Features**

* **Real-Time Data:** To handle delays, you would need a live API. The graph *structure* wouldn't change, but the **edge weights** would become dynamic. The weight could be (base\_time + current\_delay). Dijkstra's would then find the *fastest* path, not just the shortest.
* **Alternative Routes:** After finding the shortest path, you could "disable" one of its critical edges (e.g., by temporarily increasing its weight to infinity) and run Dijkstra's again. This would find the "second-best" route.
* **GUI (Graphical User Interface):** The core DelhiMetro class logic is separate from the CLI. You could import this class into a **Tkinter**, **PyQt**, or **Kivy** application to build a full desktop GUI with clickable maps and dropdown menus.