Project Report: Bangalore FlowFast

A Dynamic Traffic Optimization & Routing System

Date: October 28, 2025 **Project Files:** `app.py` , `index.html` , `script.js` , `style.css` , `requirements.txt`

1. The Problem: Gridlock in Bengaluru

Bengaluru's traffic congestion is a well-documented economic and social issue. While existing navigation tools provide routes, they are often reactive, relying on historical data or GPS pings from other users. They don't effectively model the *cause* of congestion, such as queue lengths at signals, or how a delay on one road will cascade to another in real-time. This results in routes that may not be truly optimal by the time the driver reaches the congested area.

2. Our Solution: "Bangalore FlowFast"

We have developed **Bangalore FlowFast**, a web-based application that provides the fastest travel routes based on a **live, self-correcting traffic simulation**.

Our solution consists of two main parts:

1. **Frontend (** `index.html` , `script.js` , `style.css` **):** A clean, interactive user interface built with HTML and Leaflet.js. Users can select a start and end point from key Bengaluru junctions. The frontend visualizes the calculated optimal route and, crucially, displays a live-updated map of traffic conditions (Green/Yellow/Red) across the entire network.
2. **Backend (** `app.py` **):** A powerful Flask server that acts as the "brain" of the operation. It runs a continuous, real-time simulation of the city's traffic network, updating its internal "live graph" every few seconds.

3. The Working Prototype (Suggestion #1)

Our prototype is fully functional and demonstrates the core concept. Its operation is based on a continuous feedback loop:

1. **Base Network:** The system starts with a `city_graph` (`app.py`), a static map of key junctions (Koramangala, Silk Board, etc.) and their standard, no-traffic travel times.
2. **Live Simulation:** A background scheduler (`APScheduler`) in `app.py` constantly runs two processes:
   - `simulate_traffic_flow()` : This function simulates new cars arriving at various "smart intersections" (`SmartIntersection` class), adding them to virtual queues.
   - `run_all_signals()` : This function simulates traffic lights clearing a set number of cars from the longest queues at each intersection.

3. **Dynamic Feedback Loop (`update_traffic_weights()`):** This is the system's most innovative feature. Every 10 seconds, the backend:
    - **Measures** the simulated congestion (i.e., the length of car queues) at every intersection.
    - **Calculates** a `delay` penalty for each road based on this congestion (e.g., +1 minute for every 5 cars in the queue).
    - **Generates** a new `live_graph` by adding this real-time delay to the base travel times.
4. **Smart Routing:**
    - When the user selects a route on the frontend, `script.js` calls the `/api/get-route` endpoint.
    - The backend's `find_fastest_route` function runs Dijkstra's algorithm on the `live_graph` , not the static one.
    - This ensures the route provided is the fastest path *at that exact moment*, dynamically accounting for simulated traffic jams and signal queues.

4. Core Data Structures & Algorithms (DSA) Used

This project's effectiveness is built on a foundation of classic and efficient DSA, all visible in `app.py` :

- **Graphs (Adjacency List):** The entire `city_graph` and `live_graph` are represented as **weighted graphs** using Python's dictionaries. This adjacency list format is ideal for representing road networks, mapping each junction (node) to a list of its neighbors and the travel time (weight) to reach them.
- **Dijkstra's Algorithm:** This is the **core algorithm** used in the `find_fastest_route` function. It is the perfect choice for finding the shortest (fastest) path from a single source (start point) to all other nodes in a weighted graph, which is exactly what a route-finding service needs.
- **Priority Queue (Min-Heap):** To implement Dijkstra's algorithm efficiently, we use a **min-heap** via Python's `heapq` module. The priority queue ensures that we are always exploring the most promising path (the one with the lowest current travel time) first, which is critical for the algorithm's performance.
- **Queues (FIFO):** The `SmartIntersection` class uses a `deque` (double-ended queue) from the `collections` module. This is a perfect implementation of a **First-In, First-Out (FIFO) queue** to simulate a line of cars at a traffic light, ensuring cars are processed in the order they arrive.
- **Hash Maps (Dictionaries):** Beyond the graph itself, hash maps are used for all critical O(1) average-time lookups. This includes the `intersections` dictionary (mapping junction names to their `SmartIntersection` objects) and the `incoming_lanes` within each intersection (mapping road names to their traffic queues).

5. Innovation & Uniqueness (Suggestions #3 & #4)

Our solution stands out from competitors and proves itself as an innovative, practical, and superior approach.

- **Why It's Unique:** Most navigation apps are *data aggregators*. Our system is a *simulation engine*. It doesn't just show you a traffic jam; it **models the queue** causing the jam and **predicts the delay** it will cause. This "smart intersection" model provides a much deeper, more accurate understanding of traffic flow.
- **Why It's Innovative:** The core innovation is the **dynamic feedback loop**. The `live_graph` is a "living" entity. Its edge weights (travel times) are not static or based on old data; they are recalculated every 10 seconds based on the simulated traffic flow. This makes our routing system proactively adaptive.
- **Why It's Practical & The Best Choice:**
  - **Practical:** The prototype is built, functional, and demonstrates the concept end-to-end. It proves the logic is sound.
  - **Effective:** By routing users based on a predictive simulation, our system can distribute traffic more intelligently, preventing gridlock before it becomes critical. It routes users around the *cause* of the delay (the queue), not just the *symptom* (the slow-moving traffic).

6. Conclusion & Next Steps

The "Bangalore FlowFast" prototype successfully demonstrates a novel and highly effective method for traffic routing. It moves beyond simple data aggregation into the realm of real-time simulation and predictive modeling.

Future enhancements would include:

- Integrating real-world data (e.g., from traffic cameras or IoT sensors) to "seed" the simulation, replacing the random car generator with live data.
- Expanding the `city_graph` to include a more granular map of Bengaluru.
- Developing a mobile application for on-the-go use.