A red text on a black background

AI-generated content may be incorrect.

**PROJECT AND TEAM INFORMATION**

## Project Title

(Try to choose a catchy title. Max 20 words).

|  |
| --- |
| QuickRoute: Navigating Efficient Delivery Solutions |

## Student/Team Information

|  |  |
| --- | --- |
| Team Name: | Team Path finders |
| Team member 1 (Team Lead)  (Name, Student ID, Email, Picture): | Negi, Ayush - 230112810 ayushnegi11224@gmail.com |
| Team member 2  (Name, Student ID, Email, Picture): | *Pandey, Shreya Mandeep- 23012806 shreyaapandey3@gmail.com* |
| Team member 3  (Name, Student ID, Email, Picture): | *Rawat, Akshay- 230115117 akshayraawat45@gmail.com* |
| Team member 4  (Name, Student ID, Email, Picture): |  |

**PROJECT PROGRESS DESCRIPTION**

## Project Abstract

(Brief restatement of your project’s main goal. Max 300 words).

|  |
| --- |
| The **QuickRoute** project aims to develop an intelligent and interactive web-based delivery route optimization system using classical graph algorithms. In logistics and supply chain operations, determining the most efficient path between multiple delivery points is critical for minimizing time, fuel, and resources. QuickRoute provides a dynamic, real-time solution to this challenge by enabling users to manually build and visualize delivery networks, then compute optimal paths between any two points.  The application allows users to click on a virtual map to define nodes (locations), input edges (connections) along with weights (distance/time), and visualize the resulting graph. Users can delete specific edges without affecting nodes, and select one of three pathfinding algorithms — **Dijkstra**, **A**\*, or **Bellman-Ford** — from a dropdown menu to compute the shortest route.  QuickRoute handles algorithm-specific constraints intelligently. It provides popup warnings if Dijkstra or A\* are used on graphs with negative weights, ensuring only Bellman-Ford executes under those conditions. Once the shortest path is found, the route is highlighted on the graph, the total path cost is displayed, and the path is stored and shown using a **linked list**, enhancing educational understanding of data structures.  Built using **HTML, JavaScript (vis.js), and Python (Flask)**, the application merges frontend interactivity with backend algorithm execution. The result is a seamless, visually engaging, and fully functional tool for both academic learning and real-world delivery optimization. |

## Updated Project Approach and Architecture (Describe your current approach, including system design, communication protocols, libraries used, etc. Max 300 words).

|  |
| --- |
| * **QuickRoute** is built using a modular web-based architecture combining **Flask (Python)** as the backend and **HTML + JavaScript (vis.js)** for the frontend. The project allows users to manually build graphs by clicking on a visual map to add nodes and enter edge connections with weights. Users can delete specific edges, choose algorithms via a dropdown, and visualize the shortest path in real time. * **🔧 System Design:** * Flask serves the main UI (index.html) and handles all algorithm operations through REST API routes (/add-edge, /shortest-path, /delete-edge). * The frontend captures user input and renders the graph live using **vis.js**. * Backend receives the graph data and computes the shortest path using the selected algorithm. * **📚 Algorithms:** * Implemented in Python: **Dijkstra**, **A**\*, and **Bellman-Ford**. * If negative weights are used, the system shows popup alerts and restricts execution to Bellman-Ford only. * **🔁 Data Handling:** * Graph is stored in adjacency list format. * The computed shortest path is stored using a custom **linked list** (linked\_list.py) and displayed on the UI. * **🔧 Tools:** * **vis.js** for dynamic graph rendering * **Fetch API** for frontend-backend communication * All CSS is embedded directly in index.html for simplicity * This setup ensures an interactive, real-time route optimization experience with smooth integration of graph theory and web tech. |

## 

## Tasks Completed (Describe the main tasks that have been assigned and already completed. Max 250 words).

|  |  |
| --- | --- |
| Task Completed | Team Member |
| 1. Designed and implemented Dijkstra, A\*, and Bellman-Ford algorithms 2. Built Flask backend and integrated algorithm routing 3. Developed HTML interface with embedded styling 4. Integrated **Leaflet.js** for map-based node placement 5. Integrated **vis.js** for live graph rendering and visualization 6. Enabled node and edge input via user prompts 7. Implemented edge deletion functionality 8. Developed linked list logic to store and display shortest path 9. Displayed shortest path and total weight visually on map 10. Implemented popup alerts for negative weight constraints in Dijkstra and A\* 11. Tested all algorithms with various edge cases and finalized documentation | 1. Ayush Negi 2. Ayush Negi 3. Shreya Pandey 4. Shreya Pandey 5. Akshay Rawat 6. Shreya Pandey 7. Akshay Rawat 8. Ayush Negi 9. Akshay Rawat 10. Shreya Pandey 11. Ayush Negi, Shreya Pandey, Akshay Rawat |

## Challenges/Roadblocks (Describe the challenges that you have faced or are facing so far and how you plan to solve them. Max 300 words).

|  |
| --- |
| * 1. **Map and Graph Syncing**: Combining **Leaflet.js** for node placement and **vis.js** for graph visualization required tight coordination. Ensuring the UI stayed in sync during real-time updates was initially complex. * 2. **Selective Edge Deletion**: Removing only specific edges without deleting connected nodes was tricky. Custom logic was implemented to target and remove only the selected edge. * 3. **Negative Weight Handling**: Since **Dijkstra** and **A**\* don’t support negative edge weights, validation checks and **popup alerts** were added to restrict execution and guide users to use **Bellman-Ford** when needed. * 4. **Linked List Display**: Displaying the shortest path using a backend-generated **linked list** in the UI required syncing Flask responses with dynamic DOM updates. * 5. **Backend Communication**: Sending and receiving data between the frontend and backend using fetch() with JSON payloads needed careful formatting and handling to avoid lag or crashes. |

## Tasks Pending (Describe the main tasks that you still need to complete. Max 250 words).

|  |  |
| --- | --- |
| Task Pending | Team Member (to complete the task) |
| 1. Final UI styling and visual consistency enhancements  2. Integration of road-based routing for more realistic maps | Shreya Pandey, Akshay Rawat  Ayush Negi |

## 

## Project Outcome/Deliverables

(Describe what are the key outcomes / deliverables of the project. Max 200 words).

|  |
| --- |
| The **QuickRoute** project successfully delivers an interactive, web-based delivery route optimization tool built using HTML, JavaScript, Leaflet.js, vis.js, and Flask. It allows users to manually build delivery networks by placing nodes on a map, entering edges with weights, and selecting algorithms to compute the shortest path.  All three core pathfinding algorithms — **Dijkstra**, **A\***, and **Bellman-Ford** — have been implemented and integrated with UI controls. The system intelligently prevents invalid algorithm usage (e.g., Dijkstra with negative weights) through popup alerts, improving user guidance.  The shortest path is visually highlighted, its total weight is displayed, and the path is also represented in a custom **linked list**, shown in the UI. Edge deletion works independently from node deletion, offering full manual control.  Key deliverables include:   1. Fully functional web interface served via Flask 2. Real-time graph creation and visualization 3. Algorithm switcher and logic validation 4. Linked list output representation 5. Complete documentation and test validation |

# Progress Overview (Summarize how much of the project is done, what's behind schedule, what's ahead of schedule. Max 200 words.)

|  |
| --- |
| The **QuickRoute** project is **100% complete** in terms of core logic, user interface, algorithm integration, and testing. All planned features — including manual node input via Leaflet.js, live graph visualization through vis.js, and shortest path calculations using Dijkstra, A\*, and Bellman-Ford — have been implemented successfully.  Additional enhancements like **edge deletion without affecting nodes**, **popup alerts for invalid algorithm scenarios**, and **linked list representation of computed paths** have also been integrated, pushing the project beyond its original scope.  Currently, the shortest path is visualized as a straight-line connection between nodes and is **not based on real-world road routing**. While this serves the educational and logical purpose of the tool, integrating actual map routing (e.g., using OpenStreetMap or Google Maps APIs) could be explored as a future enhancement.  No tasks are pending or behind schedule. Final testing, polishing, and documentation have all been completed ahead of time. The project is stable, demo-ready, and fully functional. |

# Codebase Information (Repository link, branch, and information about important commits.)

|  |
| --- |
| **Repository Link:** [**https://github.com/AyushNegi11224/QuickRoute**](https://github.com/AyushNegi11224/QuickRoute)  **Branch: main**  **Important Commits:**   * **init: basic Flask app structure and file setup** * **add: Dijkstra, A\*, and Bellman-Ford algorithms in Python** * **feat: Leaflet + vis.js integration for graph/map UI** * **update: edge deletion, path highlighting, and linked list output** * **fix: pop-up validation for negative weights in Dijkstra and A\*** * **final: completed documentation, tested all features** |

## 

## Testing and Validation Status (Provide information about any tests conducted)

|  |  |  |
| --- | --- | --- |
| Test Type | Status (Pass/Fail) | Notes |
| 1. Node and Edge Input 2. All Three Algorithms on Positive Weights 3. Bellman-Ford on Negative Weights 4. Edge Deletion Functionality 5. Linked List Output | 1. Pass 2. Pass 3. Pass 4. Pass 5. Pass | Inputs accepted and reflected on map and graph  Dijkstra, A\*, and Bellman-Ford give correct path and weight  Only Bellman-Ford allowed; others blocked with popups  Edge removed without affecting nodes  Path shown using linked list structure in UI |

# Deliverables Progress (Summarize the current status of all key project deliverables mentioned earlier. Indicate whether each deliverable is completed, in progress, or pending.)

|  |
| --- |
| All key deliverables of the QuickRoute project have been successfully completed. The web interface is fully functional, featuring **Leaflet.js** for interactive map-based node placement and **vis.js** for dynamic graph visualization. Users can add nodes by clicking on the map, enter edge connections with weights, and visually explore the shortest path between locations.  The system supports all three pathfinding algorithms — **Dijkstra**, **A\***, and **Bellman-Ford** — with proper handling of algorithm-specific constraints. Pop-up alerts guide users when invalid inputs (such as negative weights with Dijkstra/A\*) are detected, ensuring accurate usage.  The computed shortest path is highlighted on the graph, with total weight displayed. The path is also stored and displayed as a **linked list**, offering deeper insight into data structure handling.  Edge deletion functionality works independently of nodes, and the **Flask backend** ensures seamless communication between the frontend and the core Python-based algorithms.  It is important to note that the current shortest path visualization is based on **direct node-to-node connections** and not actual **road-based routes**. While this is ideal for algorithmic analysis and educational purposes, real-world routing using road data (e.g., via OpenStreetMap or Google Maps APIs) can be explored as a future enhancement.  All testing, validation, and documentation have been completed. The project is stable, polished, and ready for final evaluation. |