CSCI 377 City Bike Planner Final Report

| Andrew Skoblov, Hanita Darfour, Freire Jonathan |

Abstract

This project presents a web-based City Bike Planner designed to optimize the placement of bike-sharing stations and the routing of users in urban areas. By leveraging graph data structures, shortest path algorithms, and live APIs, the system calculates the most efficient bike routes between stations while considering distance, estimated time, and real-world traffic. The planner also includes real-time weather updates and bike availability at each station.

Introduction

Urban mobility is a growing concern in cities like New York. The City Bike Planner helps users navigate an interconnected network of bike stations. The main goal is to improve the usability and efficiency of bike-sharing systems using intelligent algorithms.

We created a system that calculates the shortest path between a user's current location and their destination, using Dijkstra's algorithm. Additionally, we used BFS to analyze the connectivity of our bike station network. Our planner supports station lookup, bike renting, estimated ride time, and weather updates.

System Architecture & Design

The system is a browser-based application built with:

- **HTML/CSS**: Front-end layout and styling
- **JavaScript**: Core logic, algorithms, and interactivity
- **Leaflet.js**: Map rendering and route plotting
- **OpenRouteService API**: Real-world bike routing
- **OpenWeatherMap API**: Current weather integration

Key modules:

- **Graph Module**: Custom implementation of graph using adjacency lists
- **Routing Module**: Dijkstra's algorithm and BFS
- **Weather Module**: API-based weather data retrieval
- **Bike Renting System**: Dynamic dropdown and inventory tracker

---

Data Structures

- Graph: Stores stations (nodes) and distances (weighted edges)
- Map () objects: Used for quick access to station distances and connections
- Arrays and queues: Used in BFS for level-wise traversal

Pseudocode

Dijkstra's Algorithm:

```
function dijkstra(startNode, endNode):
    set distance of all nodes to Infinity
    distance[startNode] = 0
    while unvisited nodes exist:
        node = unvisited node with smallest distance
        for each neighbor of node:
            newDist = distance[node] + weight(node, neighbor)
            if newDist < distance[neighbor]:
                distance[neighbor] = newDist
                previous[neighbor] = node
    reconstruct path from endNode
```

BFS:

```
function bfs(startNode, maxDepth):
    initialize queue with startNode
    mark startNode as visited
    while queue is not empty:
        node, depth = queue.pop()
        if depth > maxDepth: continue
        for each neighbor:
            if not visited:
                mark visited
                enqueue neighbor with depth+1
```

Algorithm Analysis

- **Dijkstra's Runtime**: O(V^2 + E)
- **BFS Runtime**: O(V + E)

Both algorithms perform efficiently due to the small graph size (under 20 nodes). The app executes route calculations in under 100ms on average.

Experiments & Interpretation

We tested routing between all station pairs in New York. For example:

- **Central Park → Wall Street**: shortest path is 7.8 km, est. time = 31 min
- BFS from Central Park with depth 2 found all connected stations
- Changing available bikes correctly updates dropdowns and availability
- Weather API correctly updates with current temp and description

Interpretation:

- Dijkstra reliably finds optimal paths
- BFS correctly enumerates paths
- Estimated time is realistic (~4 min/km)
- Weather feature adds user context (e.g., rain = reconsider bike option)

Conclusion

We successfully designed and implemented a city bike planner with core routing logic, station availability, and real-time data. The planner supports basic operations (route finding, renting, weather display) and is built for further extension (live station data, user login, historic usage tracking).

Some ideas for the implementation of algorithms, weather integration, and user interface were inspired by brainstorming with ChatGPT. All final code, structure, and logic were developed, tested, and validated by our team to ensure accuracy and alignment with the project objectives.

Appendix

- **Code Base**: index.html, style.css, script.js
- **APIs Used**: OpenRouteService, OpenWeatherMap
- **Tested Cities**: New York (default), with placeholders for others
- **Key Functions Documented**: calculateRoute(), dijkstra(), bfs(), fetchWeatherForStation()