DSA Project Report

24B1012, 24B1010, 24B0929

November 2025

1 Directory Structure

```
.
|-- Phase-1
|   |-- include
|   |   `-- nlohmann
|   |       `-- json.hpp
|   |-- Graph.cpp
|   |-- Graph.hpp
|   |-- makefile
|   |-- phase1
|   |-- SampleDriver.cpp
|   `-- script.sh
|-- Phase-2
|-- Phase-3
|-- report
|-- results
|   |-- testcase0
|   |-- testcase1
|   |-- testcase2
|   |-- testcase3
|   |-- testcase4
|   |-- testcase5
|   |-- testcase6
|   |-- testcase7
|   |-- testcase8
|   `-- testcase9
|-- tests
|   |-- testcase0
|   |-- testcase1
|   |-- testcase2
|   |-- testcase3
|   |-- testcase4
|   |-- testcase5
|   |-- testcase6
|   |-- testcase7
|   |-- testcase8
|   `-- testcase9
|-- README.md
|-- SampleDriver.cpp
40 directories, 80 files
```

1.1 Phase 1

This directory contains the implementation of Phase 1 queries. The include directory contains the header files we have used in the project. This includes the nlohmann folder that houses the single include json.hpp file to help us parse json files in C++. Next we have the Graph.cpp and the Graph.hpp files which contain the implementation and declaration of the Graph class respectively. The makefile is used to compile the code. phase1 is the executable file that is generated after compilation. SampleDriver.cpp is the driver code that is given to parse the json queries and call the appropriate C++ functions. script.sh is the bash script that we use to run our code on multiple test cases.

1.2 Phase 2 and Phase 3

These directories follow the same structure as Phase 1, with the implementation of Phase 2 and Phase 3 queries respectively. The executables are named phase2 and phase3 respectively.

1.3 report

This directory contains the LaTeX source code of this report.

1.4 results

This directory contains the output files generated after running our code on the test cases in the tests directory. Each testcase$_i$ directiory contains 2 files - output1.json and output2.json respectively for Phase 1 and Phase 2 query outputs.

1.5 tests

Each testcase$_i$ directory contains the testcases we run our input on. Each directory contains the graph.json file which contains the graph data and the queries1.json and queries2.json files which contain the queries for Phase 1 and Phase 2 queries respectively.

1.6 README.md

This file contains the instructions to compile and run the code.

2 Makefile and Targets

3 Assumptions

4 Test Case and Analysis

5 Python Scripts and Libraries

6 Time and Space Complexity

7 Approach for Each Query

7.1 Phase 1 Queries

7.1.1 Remove Edge

This is pretty straightforward. We just make the disabled flag of that edge true in case the edge exists and is not already disabled. We return true in this case. If the edge does not exist or is already disabled, we return false.

7.1.2 Modify Edge

We first check if the query has a patch. If it does, we check which attributes need to be modified. If no modification is needed, we return false. Otherwise we modify the required attributes and return true. Before modification we check if the edge exists and is not disabled. If it does not exist we return false directly. If it exists and is disabled we enable it and automatically return true since modification is guaranteed. Modification of attributes is done as per the patch provided in the query.

7.1.3 Shortest path

For shortest path query, we use Dijkstra's algorithm. We first parse through the query to get required parameters like source, target, forbidden nodes and road types. We then run modified Dijkstra's algorithm on the graph considering the forbidden nodes and road types. If we reach the target node, we return the shortest distance. If we exhaust all possibilities without reaching the target, we return false in the "possible" attribute of answer indicating no path exists. The notion of minimum is either distance of time based on the query parameter. If it is distance, we consider edge weights as distances. If it is time, time cost is calculated using the speed profile given in the edge attributes which is done by finding the speed corresponding to the given time of day and calculating time cost as distance/speed.

7.1.4 KNN

There are two types of KNN queries: one based on euclidean distance and the other based on shortest path distance. For euclidean distance based KNN query, we simply calculate the euclidean distance from the given coordinates to all other nodes and maintain a max-heap which we pop until size is K to get the K nearest nodes which are then returned. For shortest path distance based KNN query, first we find the nearest node from the given coordinates using euclidean distance. We then run modified Dijkstra's algorithm from that node to find shortest path distances to all other nodes considering forbidden nodes and road types. We maintain a max-heap of size K to get the K nearest nodes based on shortest path distance which are then returned.

7.2 Phase 2 Queries

7.2.1 K shortest paths (heuristic)

Unlike before where we used Yen's algorithm to find K shortest paths, we now use a heuristic approach to find candidate paths and then select the best one based on certain criteria. For finding the candidate paths, we first find the shortest path using Dijkstra's algorithm. We then multiply the edge weights of the edges in the shortest path by a penalty factor which needs to be tuned appropriately and run Dijkstra's algorithm again to find another path. We repeat this process until we have adequate number of candidate paths. Penalty is also implemented for paths that are unique so that running Dijkstra again does not return same path which will lead to infinite loop. The candidate paths is valid if we do not get a previously found path again and if the length of the path is within a certain stretch factor of the original shortest path length after the size of candidate set is k in order to ensure k outputs.

Once we have the candidate paths, we need to choose the K paths among them. To do this, we calculate 2 types of penalties for each candidate path: distance penalty and overlap penalty. We have a dynamic set of paths whose sized is capped at K. For each path we it to our dynamic set temporarily and calculate the distance penalty and overlap penalty with respect to the paths already in the dynamic set as given in the problem statement.

If the total penalty is minimized by adding this path, we permanently add this path to the dynamic set. We repeat this process until we have K paths in our dynamic set or we exhaust all candidate paths. Finally we return the paths in the dynamic set as our answer.

8 Phase 3 Explanation