Graph-Based Routing System Simulation

Project Introduction

Modern applications like **Google Maps**, **Ola**, and **Zomato** rely heavily on **graph-based algorithms** to compute optimal routes, handle dynamic traffic updates, and generate personalized suggestions. Behind these systems lies an intricate web of **Data Structures and Algorithms (DSA)** working together to deliver results efficiently, even on massive datasets.

In this project, you will apply your knowledge of DSA to a real-world setting by simulating a **routing system** that models key features of real navigation platforms. You will work in teams to design, implement, and optimize a **multi-phase system** that handles graph-based queries, including shortest paths, k-nearest neighbor searches, and path constraints — all while ensuring modularity, scalability, and performance.

Phased Structure Overview

The project is divided into **three phases**, each designed to progressively deepen your understanding and challenge your algorithmic thinking:

Phase 1 and Phase 2 [Deadline: 10th November]

Focus on implementing robust algorithmic modules and APIs. You will be provided standardized input formats (JSON) representing **user queries** and **graph data**. Your system must parse these inputs, process the queries efficiently, and output results in the specified format.

These phases emphasize **implementation quality**, **correctness**, and **modular design**, and will be **partly autograded**.

Phase 3 [Deadline: 22nd November]

This phase transitions toward an **open-ended, research-oriented exploration**. You will handle a realistic problem similar to those faced by instant delivery companies — a variant of the **Travelling Salesman Problem (TSP)**. You will analyze, benchmark, and experiment with **advanced algorithms and heuristics**, focusing on creativity, efficiency, and benchmarking performance.

System and Evaluation Requirements

You are free to model your **system architecture** as you see fit — no rigid templates are provided. However, **code modularity**, **clarity**, and **software engineering discipline** will be key evaluation criteria. You are expected to:

- Design your own APIs, data abstractions, and testing framework.
- Maintain readability and logical separation of components.
- Create your own test cases and benchmarking scripts.
- Provide a **Makefile** with all necessary build options.

In the real world, software engineers are not provided test cases — hence, **you must generate and validate your own** test data and performance metrics.

Your **report** should include:

- Implementation details
- Assumptions made
- Time and space complexity
- Analysis on real-world test cases you created

Directory Structure and Deliverables

Directory Structure

```
None
|-- Phase-1/
|-- Phase-2/
|-- Phase-3/
|-- MakeFile
|-- SampleDriver.cpp
|-- Report.pdf
```

Each subdirectory corresponding to a phase must contain all .cpp and .hpp files used in that phase. It is recommended to organize code by separating implementation (.cpp) and header (.hpp) files.

You can use (.py) files for generating test cases. Don't provide large JSON files in the submission zip, rather provide just the scripts you used to generate them. We will be discussing them in the Vivas.

Makefile Requirements

Your Makefile must have **three targets (executables):**

- phase1
- phase2
- phase3

Each executable must:

- Be created in the **parent directory** (not inside subfolders)
- Accept **two arguments:**
    1. The JSON file describing the graph
    2. The JSON file specifying the queries

SampleDriver.cpp will be used to handle argument parsing, input/output management, and query execution.

Phase-1

Objective

Implement standard algorithms on a **dynamic graph** that supports **periodic updates and queries**.

Queries

1. **Shortest Path:**
    - Minimizing distance
    - Minimizing time with speed limit as a function of time (changing traffic)
    - Constraints on edges/nodes (e.g., forbidden roads or restricted nodes)
2. **KNN (K Nearest Neighbors):**
    - Based on Euclidean Distance
    - Based on Shortest Path Distance

Phase-2

Objective

Implement **advanced heuristic and approximate algorithms** for routing queries.

Queries

1. **K Shortest Paths (Exact)** — by distance, for k between 2–20.

2. **K Shortest Paths (Heuristic)** — penalize overlapping with the best path and paths that deviate too much from optimal.

In the queries of type-1 you would have observed that the paths are very close to each other, but you know Google Maps do a lot better. So, the final output will be a heuristic of the closeness to shortest distance and the variety of paths. In the algorithm try to have parameters tuning which you can increase or decrease the priority.

3. **Approximate Shortest Paths** — prioritize speed over accuracy (batch queries performance test).

Scoring is based on the **accuracy–speed tradeoff** across a large batch of queries. You will be provided a large number of queries in some small time interval. The MSE from the Shortest Path distances will be computed to score.

You need to handle queries one by one, if the combined query's time crosses the budget time the query will be rejected. Check the query JSON format once.

Phase-3: Delivery Scheduling

Companies like **Zomato, Swiggy, Zepto, and Blinkit** face a complex and fundamental challenge — *how to utilize their delivery fleet most efficiently.* In this project, we present a simplified yet representative version of this real-world optimization problem.

This task is closely related to the well-known **Travelling Salesman Problem (TSP)**, an NP-Hard problem in combinatorial optimization. While solving it optimally for large inputs is computationally infeasible, your objective is to **explore and implement practical heuristics and state-of-the-art algorithms** that perform well in realistic scenarios.

You are expected to design algorithms that balance **accuracy, computational speed, and scalability**, taking advantage of the fact that the underlying graphs are **Euclidean** rather than arbitrary. The evaluation will be **relative**, based on the trade-off between execution time and solution quality.

A detailed report should accompany your implementation, explaining your algorithmic choices, heuristics used, experimental results, and insights gained — particularly emphasizing the reasoning behind your design decisions and their real-world relevance.

Problem Statement

At time **t = 0**, there are **n delivery boys** all located at a **central depot**. There are **m delivery orders**, each having a pickup and delivery node.

Each delivery boy can carry multiple orders and deliver them in any order — as long as **pickup precedes delivery**.

Goal

Minimize the **total delivery time**, defined as the sum of delivery completion times for all orders. Compare this with minimizing for the maximum delivery time, i.e. the max of the individual delivery completion times.

Appendix — Formalized Specifications

1. General Notes

- JSON files must be valid UTF-8.
- Node/edge/query IDs are unique integers.
- Distances: meters | Time: seconds | Speed: m/s.
- Latitude/Longitude treated as Euclidean for small areas.

---

2. Directory and Makefile

```
None
ProjectRoot/
 Phase-1/
 Phase-2/
 Phase-3/
 Makefile
  SampleDriver.cpp  (#  copy  this  file  in  all  phases  to  use
accordingly)
 Report.pdf
```

Executables must be named exactly:

```
None
./phase1 <graph.json> <queries.json> <output.json>
./phase2 <graph.json> <queries.json> <output.json>
./phase3 <graph.json> <queries.json> <output.json>
```

You can have other options in the Makefile. Just ensure that these options are present.

---

3. Graph File Schema — graph.json

```JSON
{
      "meta": {
            "id": "sample_testcase_1",
            "nodes": 5,
            "description":  "Instructor-provided  map  for  course
project"
      },
      "nodes": [
            {
                  "id": 0,
                  "lat": 19.070000,
                  "lon": 72.870000,
                  "pois": ["Restaurant", "Hospital"]
            }
      ],
      "edges": [
            {
                  "id": 1001,
                  "u": 1,
                  "v": 2,
                  "length": 100.0,
                  "average_time": 5.5,
                  "speed_profile": [40, 42, 39, 45, ...(96 slots)],
#only in phase1
                  "oneway": false,
       "road_type": "primary"
    }
  ]
}
```

- Node IDs are 0-based.
- Respect direction for one-way edges.

- Fallback to average_time if speed_profile missing.
- Time-dependent profiles use 96 × 15-minute slots (the average speeds throughout the day in 15 min intervals).
- **Constraints**: **Nodes <=1e5, Edges <= 1e5**

Important Note: The driver code ('SampleDriver.cpp') adds the processing_time to all the query outputs. You don't need to worry over that.

4. Queries — Phase 1

Input Format

```JSON
{
      "meta": { "id": "qset1" },
      "events": [ <query_or_update>, ... ]
}
```

Example Queries

1. **Dynamic Updates:**

```JSON
 #input format
{
      "type": "remove_edge",
      "edge_id": 1001
}

#output format
{
      "done": true
}
```

```JSON
 #input format
{
      "type": "modify_edge",
      "edge_id": 1002,
      "patch": { "length": 120.0 }
}

#output format
{
      "done": true
}
```

2. Shortest Path:

```JSON
#input format
{
      "type": "shortest_path",
      "id": 1,
      "source": 10,
      "target": 250,
      "mode": "time",
      "constraints": {
       "forbidden_nodes": [5],
       "forbidden_road_types": ["primary"]
      }
}

#output format
{
      "id": 1,
      "possible": true,
      "minimum_time/minimum_distance":123,
      "path":[list of node ids from source to target]
```

```
}
```

- Mode will be either of “time” / ”distance”.
- forbidden_nodes = list of node ids forbidden.
- forbidden_road_types = list of road types (“primary”,”secondary”,”expressway”,etc).
- If no possible path output “possible” as false and no need of other options.

2. **KNN Query:**

```JSON
#input format
{
      "type": "knn",
      "id": 2,
      "type" : "resturant",
      "query_point": { "lat": 19.07, "lon": 72.87 },
      "k": 5,
      "metric": "shortest_path"
}

#output format
{
      "id" :2,
      "nodes": [list of required node ids]
}
```

- The query returns the k nearest neighbor of given type. If not more than k available then return the available ones.

---

5. Phase 2 Queries

K Shortest Paths (Exact)

JSON

```
#input format
{
      "type": "k_shortest_paths",
      "id": 10,
      "source": 10,
      "target": 250,
      "k": 5,
      "mode": "distance"
}

#output format
{
      "id" : 10,
      "paths":[
            {
                  "path": [List of node ids]
                  "length": 123
            }
      ]
}
```

K Shortest Paths (Heuristic)

JSON

```
{
  "type": "k_shortest_paths_heuristic",
  "id": 12,
  "source": 10,
  "target": 250,
  "k": 5,
   "heuristic": { "overlap_penalty": 0.7, "distance_penalty": 0.3
}
}

#output format
```

```
{
      "id" : 12,
      "paths":[
            {
                  "path": [List of nodes]
                  "length": 123
                  "penalty": 12
            }
      ]
}
```

Approximate Shortest Path

JSON

```
#input format
{
      "type": "approx_shortest_path",
      "id": 13,
      "queries": [
            {"source": 10,
             "target": 250},
            {"source": 12,
             "target": 253}
      ]
      "time_budget_ms": 10,
      "acceptable_error_pct": 5.0
}

#output format
{
      "id":13
      "distances":["approx shortest distances in order"]
}
```

- All query outputs within the acceptable error percentage from the exact will be counted.
- If the total time taken by the query exceeds the budget then the score is 0. So, please take care to output well before time safely.

---

6. Phase 3 — Delivery Scheduling

Input queries

```JSON
{
      "orders": [
            {
                  "order_id": 1,
                  "pickup": 100,
                  "dropoff": 200
            },
            {
                  "order_id": 2,
                  "pickup": 150,
                  "dropoff": 300
            }
        ],
      "fleet": {
                  "num_delievery_guys": 1,
                  "depot_node": 0
            }
}
```

Output Example

```JSON
{
      "assignments": [
            {
                  "driver_id": 0,
                  "route": [0, 100, 110,  200, 150, 300],
                  "order_ids":[1, 2]
            }
      ],
```

```
    "metrics": {
        "total_delivery_time_s": 12345.0
    }
}
```

---

7. Validation Checklist

- Maintain event order in output for grading compatibility.
- phase1, phase2, phase3 binaries present and runnable.
- Each takes two arguments and outputs valid JSON.
- No private test cases included in submission zip — only generation scripts.

---

8. Grading Summary (Subject to Change)

- **Phase 1 & 2 (5+7 marks):** Correctness on hidden tests, API conformance, modularity, and benchmarks for approximate methods.
- **Phase 3 (8 marks):** Working scheduling solution(s) with reproducible benchmarks, depth of analysis, and code quality.

---

9. Creating Testcases

- You can use Open Street Maps to extract the real world map nodes, edges and Normal Distributions to model time, average speed.
- You can also directly generate sample test cases using fair assumptions. While generating outputs try to use algorithms without any assumptions, not caring about speed, just correctness.
- You are free to share test cases with each other but not the generating scripts.

---

10. AI policy

You are free to choose AI tools. However, you need to keep a record of the AI interactions. You need to include your chat logs in your report.

---

11. Plagiarism policy

We will check against plagiarism. If we find any instances of code copying, we will forward the cases to DADAC.
