# 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 **three arguments:**
    1. The JSON file describing the graph
    2. The JSON file specifying the queries
    3. The JSON file containing the output

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 (not time)

---

# 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.

The paths need to be simple paths, i.e., loopless paths, as in the real world, a user getting a looped path will be really annoyed.

For just this part, the constraint on both edges and vertices of the graph is 5000.

2. **K Shortest Paths (Heuristic)** — penalize overlapping in the paths and paths that deviate too much from optimal, for k between 2-7

In the queries of type-1, you would have observed that the paths are very close to each other, but you know Google Maps does a lot better. So, the final output will be a heuristic of the closeness to the shortest distance and the variety of paths. In the algorithm, try to have parameter tuning so that 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 a small time interval. ~~The MSE from the Shortest Path distances will be computed to score.~~

The threshold error percentage will be used. It was always there in the appendix and the JSON format. There was just a mismatch between here and in the JSON.

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.

**For the phase-3, you can ignore the speed profiles and always use the average time.**

---

# 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.

**About `SampleDriver.cpp`**

The file is given just to help you understand how we want the query input and output to be handled. You can format the file as per your needs. You can copy it and use it. Just keep note of the following things:

- The final output binaries should take the three file paths as specified above.
- You should first handle reading the **`graph.json`**
- Once you have read the graph structure, you can start your pre-processing. Pre-processing time won't be counted in grading, but keep it reasonable, not more than 5 mins.
- Post this, you can read the **`queries.json`**.
- **Please handle the queries one by one; processing queries in batches can result in loss of marks.**
- Before starting the process, the query function starts the chrono timer, and in the query output JSON file, appends the processing time in ms as follows:

```
JSON
{
      "id": 123,
      # other required things in the query
      "processing_time": 12
}
```

---

# 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": [40.0, 42.0, 39.0, 45.0, ...(96
slots)], #only in phase1
                  "oneway": false,
                  "road_type": "primary"
            }
      ]
}
```

- Node IDs are 0-based (0, 1, 2, …, n-1).
- Respect direction for one-way edges.
- Fallback to average_time if speed_profile is missing.
- Time-dependent profiles use 96 × 15-minute slots (the average speeds throughout the day in 15-minute intervals).
- For the speed profile problems, always assume you are at the starting node on t=0.
- For a road taking more than 15 minutes, you should use the next speed profile after 15 minutes.

- Road types can only be: “primary”, “secondary”, “tertiary”, “local”, and “expressway”.
- Node pois can only be: “restaurant”, “petrol station”, “hospital”, “pharmacy”, “hotel”, “atm”.
- **Constraints**: **Nodes <=1e5, Edges <= 1e5**
- To keep things simple, we are sticking to simple graphs (between two nodes A,B, either one or two one-way edges will be there, or a two-way edge will be there in between A and B)

---

**Important Note:** The driver code (‘SampleDriver.cpp’) adds the processing_time to all the query outputs. You don’t need to worry over that. Check the new description of the SampleDriver file given in Appendix Section 3.

# 4. Queries — Phase 1

## Input Format

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

## Output Format

```JSON
{
      "meta": { "id": "qset1" }
      "results": [ <query_or_update_outputs>, ... ]
}
```

- “events” in the input contains an array of queries/updates in order, as mentioned in the examples below.
- “results” in the output contains an array of output of queries/updates in order, in the formats mentioned in the examples below.
- “meta” in the output file should be copied from the input file meta.

## Example Queries

1. **Dynamic Updates:**

JSON

```
 #input format
{
      "id": 121,
      "type": "remove_edge",
      "edge_id": 1001
}

#output format
{
      "id": 121,
      "done": true
}
```

- Removing an edge only disables the edge (blocking a road). The edge may be used again (for example: reopening a road).
- If you are removing an edge that is already “removed” then output done as false.

JSON

```
 #input format
{
      "id": 37,
      "type": "modify_edge",
      "edge_id": 1002,
      "patch": { "length": 120.0 }
}

#output format
{
      "id": 37,
      "done": true
}
```

- If the edge_id given corresponds to a deleted edge, then restore the edge (reopen the road) with the values in the patch. If the patch is empty, then use the previously stored values.
- In other cases, if the edge is not present or if the patch is empty, then ignore the query and output “done” as false.
- “patch” can only change the fields: “length”, “average_time”,”speed_profile”,”road_type”.

- The update will count for all the queries coming after it.

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":[10, 13, 17, 100, 104, 250] #list of node ids from
source to target in order
}
```

- Mode will be either “time” / ”distance”.
- forbidden_nodes = a list of node IDs forbidden.
- forbidden_road_types = list of road types (“primary”,” secondary”, etc).
- If no possible path, output “possible” as false and no need for other options.
- The constraints field or fields inside it may be missing. You need to handle these things in your code.
- The field is “minimum_time/minimum_distance”

2. **KNN Query:**

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

#output format
{
      "id" :2,
      "nodes": [4, 7, 9, 11, 17] #list of node ids in any order
}
```

- The query returns the k nearest neighbors of a given type. If not more than k available, then return the available ones.
- “metric” is either “shortest_path” or “euclidean”.
- For the shortest path, you can consider the paths from the nearest node point to the given latitude, longitude. (Assume no ties)
- Here, the shortest path is with respect to distance, not time.

---

# 5. Phase 2 Queries

### K Shortest Paths (Exact)

```JSON
#input format
{
      "type": "k_shortest_paths",
      "id": 10,
      "source": 10,
      "target": 250,
      "k": 2,
```

```
      "mode": "distance"
}

#output format
{
      "id" : 10,
      "paths":[
            {
                  "path": [10,11,13,17,20,250],
                  "length": 123
            },
            {
                  "path": [10,11,12,18,20,250],
                  "length": 128
            }
      ]
}
```

- Output the k-shortest paths in order of distance.

### K Shortest Paths (Heuristic)

JSON

```
{
  "type": "k_shortest_paths_heuristic",
  "id": 12,
  "source": 10,
  "target": 250,
  "k": 2,
  "overlap_threshold": 60
}

#output format
{
      "id" : 12,
      "paths":[
```

```
            {
                  "path": [10,11,13,17,20,250],
                  "length": 123
            },
            {
                  "path": [10,8,12,18,13,250],
                  "length": 135
            }
      ]

}
```

- Again, output the k-shortest paths in order of distance.
- The first path should be the real shortest path
- For this problem, you can assume k to be 2-7.
- For scoring on how good the output is:
    - Penalty for all roads will be calculated.
    - Overlap Penalty for i th Path = number of other paths (among the k paths) with more than the “overlap_threshold” percentage of edges common
    - Distance Penalty for i th Path = percentage difference from the shortest path length / 100 + 0.1
    - Penalty for i th Path = Overlap Penalty * Distance Penalty
    - Total Penalty = Sum of penalties of all k paths
    - You need to minimize this total penalty.

### 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":[
                  {"source": 10,
                   "target": 250,
                   "approx_shortest_distance":123},
                  {"source": 12,
                   "target": 253,
                   "approx_shortest_distance":115}
                  ]
}
```

- 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 will be penalised. So, please take care to output well before time safely.
- Acceptable error percentage will be between 5%, 10% or 15%

---

# 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 are present and runnable.
- Each takes two arguments and outputs valid JSON.
- No private large test cases included in submission zip — only generation scripts.

- While processing queries, **ensure each one is enclosed within a try–catch (try–except) block**. This prevents the entire program from crashing (e.g., due to a segmentation fault) and allows it to continue handling subsequent queries gracefully.

---

# 8. 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, but not the generating scripts.

---

# 9. Report

- **Directory Structure of Code:** Provide a clear description of the directory layout of your project. Mention the purpose of each major folder and file so that the evaluator can easily understand where different components of your code reside.
- **Makefile and Targets:** Include your Makefile in the report and briefly explain all the targets you have defined (e.g., compilation, running, cleaning, testing). Mention how each target should be used.
- **Assumptions:** If you have made any assumptions during implementation or testing, list them clearly in the report. These can include assumptions about input format, constraints, data limits, or any design choices.
- **Test Cases and Analysis:** For any phase where you created your own test cases, include them and describe the analysis you performed—such as average execution time, accuracy of results, and other relevant observations.
- **Python Scripts and Libraries:** If you used Python for generating or analyzing test cases, mention any external libraries you used. Also include instructions on how to run these scripts and note any assumptions made while creating the test cases.
- **Time and Space Complexity:** For every algorithm or major code component you implemented, include the time complexity and space complexity. This should cover all query types and phases.
- **Approach for Each Query Type:** For every type of query you handle, describe the approach taken. You may simply mention standard algorithms that were covered in the slides or lectures. For anything not covered in class, provide a short explanation of the method and reasoning.
- **Phase 3 Explanation:** For Phase 3, give a detailed explanation of your algorithms, ideas, techniques used, and the analysis performed. This section should reflect your effort in exploring different directions and developing your final solution..

---

# 10. 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.

### Important Points Regarding Grading

**Phase 1:**

- The autograded component will be evaluated based on both accuracy and speed.
- A possible criterion could be 70% weightage for accuracy under reasonable execution time and 30% for execution time relatively.
- Execution time will be relatively graded; an example criterion will be scores linearly scaled between the maximum and minimum recorded times.
- Aim to design algorithms that are not only correct but also highly efficient. You are encouraged to explore and implement various optimizations.

**Phase 2:**

- The autograded portion will follow a similar grading scheme as Phase 1, balancing accuracy and efficiency.
- For heuristic evaluation, relative grading and threshold-based assessments will be applied.
- Focus on developing algorithms that perform well both quantitatively (in terms of results) and qualitatively (in terms of heuristics and adaptability).

**Phase 3:**

- The final evaluation will include some relatively graded components, but a significant portion of the marks will depend on:
    - The depth of exploration and experimentation,
    - The originality of ideas, and
    - The quality and comprehensiveness of your final report.

---

# 11. 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.

---

# 12. Plagiarism policy

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