Graph-Based Routing System Simulation

CS293: Data Structures and Algorithms Project

Phase 1–3 Report

Harsha Vardhan (Roll No: 24B1069)
Rishi Vardhan M (Roll No: 24B0969)
S. Vivek (Roll No: 24B0912)
Indian Institute of Technology, Bombay

Date: November 22, 2025

Contents

1. Introduction

Modern navigation and delivery systems (e.g., Google Maps, Swiggy, Blinkit) rely on **graph-based algorithms** to compute optimal routes, handle real-time traffic updates, and optimize logistics. This project simulates such a system through three progressive phases — implementing, optimizing, and extending routing algorithms on large dynamic graphs.

1.1. Phase 1: Core Algorithms

In this phase, we have focused on building the core routing engine on a dynamic graph. It supports edge updates, time-dependent shortest paths, constraint-aware routing, and K-nearest-neighbor (KNN) queries using both Euclidean and graph distances.

1.2. Phase 2: Advanced Routing

In this phase, we have extended the routing agent using advanced techniques: like computing k-shortest simple paths, researched and used efficient heuristic methods to diversify routes, and developed fast approximate shortest-path algorithms optimized for large batches under strict time constraints.

1.3. Phase 3: Delivery Scheduling

In this phase we have applied routing concepts to a delivery scheduling problem inspired by real-world logistics. Here, multiple delivery orders must be assigned and routed efficiently using heuristics for variants of the Travelling Salesman Problem (TSP), balancing solution quality and runtime.

2. System Architecture and Directory Structure

2.1. Directory Layout

```
ProjectRoot/
|
|-- Phase-1/
|   |
|   |-- inc/
|   |    |-- graph.hpp
|   |
|   |-- nlohmann/
|   |    |-- json.hpp
|   |
|   |-- src/
|        |-- graph.cpp
|        |-- shortest_path.cpp
|        |-- knn.cpp
|        |-- dynamic_updates.cpp
|        |-- helpers.cpp
|
|-- Phase-2/
|   |
|   |-- src/
|        |-- k_shortest_paths.cpp
|        |-- k_shortest_paths_heuristic.cpp
|        |-- approx_shortest_path.cpp
|        |-- helpers.cpp
|
+-- Phase-3/
|   |
|   |-- src/
|        |-- solve_delivery_scheduling.cpp
|        |-- helpers.cpp
|
+-- SampleDriver.cpp   (root-level driver)
+-- Makefile
+-- README.md
```

2.2. Makefile Targets

Target	Executable	Description
phase1	./phase1	Runs Phase-1 implementation
phase2	./phase2	Runs Phase-2 implementation
phase3	./phase3	Runs Phase-3 delivery scheduling
clean	-	Removes object and binary files

2.3. Makefile Structure:

```
SHELL := /bin/bash
CXX = g++
CXXFLAGS = -O3 -march=native -funroll-loops -ffast-math -std=c++20 -Wall -Wextra

# Include directories
INCLUDES = -I Phase-1 -I Phase-1/inc -I Phase-1/nlohmann

# Contains all the source files required to build phases 1&2 respectively
PH1_SRC = $(wildcard Phase-1/src/*.cpp) Phase-1/SampleDriver.cpp
PH2_SRC = $(wildcard Phase-2/src/*.cpp)
PH3_SRC = $(wildcard Phase-3/src/*.cpp)

# Building Phase-1 target
phase1: $(PH1_SRC) $(PH2_SRC) $(PH3_SRC)
 $(CXX) $(CXXFLAGS) $(INCLUDES) -o phase1 $(PH1_SRC) $(PH2_SRC) $(PH3_SRC)

# Building Phase-2 target
phase2: $(PH1_SRC) $(PH2_SRC) $(PH3_SRC)
 $(CXX) $(CXXFLAGS) $(INCLUDES) -o phase2 $(PH1_SRC) $(PH2_SRC) $(PH3_SRC)
# Building Phase-3 target
phase3: $(PH1_SRC) $(PH2_SRC) $(PH3_SRC)
    $(CXX) $(CXXFLAGS) $(INCLUDES) -o phase3 $(PH1_SRC) $(PH2_SRC) $(PH3_SRC)
clean:
 rm -f phase1 phase2 phase3 overall
```

3. Phase 1 — Dynamic Graph and Core Algorithms

3.1. Objective

In this phase our main objective was to design a system that is capable of handling:

- Dynamic edge updates (remove/modify)
- Time-based and distance-based shortest path queries
- K-nearest neighbor (KNN) queries using Euclidean or shortest-path distance

3.2. Data Structures

- **Adjacency List:** The graph is stored as a contiguous `vector<vector<AdjEdge>> adj`, where `AdjEdge` is a compact internal representation containing only the required information per-query. This improves spatial locality and CPU cache efficiency during shortest-path expansions.
- **Node/Edge Indexing:** Nodes and edge IDs are mapped to compact internal indices using `idToIdx` and `edgePos`, enabling O(1) access to adjacency entries and edge updates.
- **Dynamic Update Maps:** Dynamic Updates are supported in O(1) using:
  - `edgePos:` maps edgeID -> (source node index, adjacency index)
  - `active:` global boolean array indicating whether each edge is enabled or not
  - `AdjEdge.active:` it is a per-edge flag inside the adjacency list.

3.3. Algorithms Implemented

3.3.1 Helper Functions:

- `euclDist:` This just calculates the euclidean distance between two points.
- `cellKey:` It creates a key for a cell in the grid after
- `compTimeCost:` It computes the time to traverse an edge starting at a given time of the day, using the edge's **time-dependent speed profile**
- `findNearestNode:` Given the GPS coordinates, it efficiently finds the **nearest graph node**.

3.3.2 Dynamic Updates

- `remove_edge`: In this, we just mark the specified edge as inactive using the internal `active` flag. Since the adjacency list is unchanged, the removal is performed in O(1) time using the precomputed `edgePos` lookup. If the edge is already inactive or does not exist, the update is ignored and the response reflects failure.
- `modify_edge`: In this we update the stored attributes of an edge (length, average time, speed profile, or road type). If the edge had previously been removed, then it is first restored and then patched; if the patch is empty, then only the restoration is performed. Invalid or redundant modifications return a failure flag, while the updates execute in O(1) time.

3.3.3 Shortest Path (Distance/Time)

This is implemented using an optimized **Dijkstra's Algorithm**, supporting both distance-based and time-based routing. In the distance mode, each edge contributes its physical length as the weight, and the algorithm expands nodes in increasing order of accumulated distance to guarantee optimality. In the time mode, the edge weight is computed dynamically using the 96-slot speed profile: at every relaxation step, the function `compTimeCost()` determines the traversal time starting from the current arrival time, thereby simulating realistic time-dependent traffic conditions. The search respects all constraints specified in the query—such as forbidden nodes or forbidden road types—by simply skipping restricted edges during relaxation. Once the target is reached, the final route is reconstructed from the `parent[]` array. If the target remains unreachable, the response is marked as impossible.

3.3.4 KNN Queries

- **Euclidean KNN:** For geographic nearest-neighbour queries, we are using a grid-based spatial index to restric the search to nearby cells, avoiding a full-scan of all the nodes. At each radius, the algorithm gathers POIs from nearby grid cells and computes their straight-line distances.
  Once we get atleast k candidates, a boundary-distance check determines whether further expansion is necessary.
  If the search fails to collect k points even after a large radius, the method safely falls back to a full scan of all POIs of that type. The candidates are then sorted by Euclidean distance, and the first k are returned.
  This is a good strategy as it ensures both correctness and strong average-case performance.

- **Shortest-path KNN:** For network-based proximity, the query location is first snapped to the closest graph node using the spatial grid. A single Dijkstra search is then executed from this snapped node, using edge lengths as weights. Whenever a node containing the target POI type is permanently settled, it is appended to the answer list. Since Dijkstra visits nodes in increasing of their true shortest-path distance, the first k POIs discovered are guaranteed to be the k nearest on the road network. The search stops as soon as k POIs are found, avoiding unnecessary exploration of the full graph.

3.4. Optimizations

The following are the modifications made while doing the project, to make it more efficent:

- Replaced `unordered_map`-based adjacency, with index-based vectors to reduce the hashing overhead and improve the cache locality.
- Preallocated reusable arrays for Dijkstra (distance, parent, visited), avoiding repeated allocations per query.
- Incorporated early-termination in the shortest-path KNN, that is stopping as soon as k POIs are discovered.
- Also optimized the time-dependent edge traversal by reducing slot computations and using fast modular indexing.

3.5. Complexity Analysis

- **Shortest Path (A* / Dijkstra):** $O((V + E) \log V)$
- **KNN (Euclidean):** $O(k \log k + C)$ average with spatial grid, worst-case $O(V \log V)$ if grid fallback is triggered.
- **KNN (Shortest-path):** $O((V + E) \log V)$ but typically much faster due to early stopping after k POIs.
- **Dynamic Updates:** $O(1)$ using the `edgePos` index map.

3.6. Testing

For testing my functions in Phase-1, I have used AI to create the following python programs:

- `generate_graph.py`: This program generates a `graph.json` file when given number of nodes and edges as input. It is present in the tests subfolder of Phase-1.
- `generate_queries.py`: This program generates a `queries.json` file when given the `graph.json` as input. It is also present in the tests subfolder.
- `checker_phase1_full.py`: This program checks if the outputs that we get after running our code on the generated graph and queries, is matching with the logically correct output (which is generated in this file in Python).

The following are the used packages in the python files:
`json, sys, math, heapq, collections, random, argparse`

3.7. Instructions for Running the Tester:

The following instructions are run in the terminal when inside the tests subfolder.

```
python3 generate_graph.py ../graph.json --nodes <numNodes> --edges <numEdges>
```

This generates a graph with <numNodes> nodes and <numEdges> edges.

Then next, the following code creates the query.json file

```
python3 generate_queries.py ../graph.json ../queries.json --n <numQueries>
```

The following code is used when we have generated a `phase1` object file:

```
../../phase1 ../graph.json ../queries.json ../myOutput.json
```

Finally to test it, we need to do this:

```
python3 checker_phase1_full.py ../graph.json ../queries.json ../myOutput.json
```

This will check if there are any mismatches between my output and expected output and give the result.

4. Phase 2 — Advanced Routing Algorithms

4.1. Objective

Our objective for Phase–2 is to extend the Phase–1 routing engine by implementing more advanced algorithms that can produce multiple alternative routes and ultra-fast approximate results under some tight time budgets. Mainly, our goals are:

- Computing **exact loopless k-shortest paths** for $k \in [2, 20]$.
- Generating **diverse heuristic alternatives** that penalize overlap and excessive detours.
- Supporting **approximate shortest paths** optimized for speed over accuracy.

4.2. Exact K Shortest Paths (Yen's Algorithm)

This query computes k loopless shortest paths between a source and a destination. Our implementation uses a **simplified variant of Yen's Algorithm**, adapted to fit the 5000-node constraint of Phase-2.
First, we compute the globally optimal path using Dijkstra's algorithm. Then, for selected deviation points along this path, we temporarily suppress only the **immediate outgoing prefix-edge** and recompute the shortest "spur" path.
Each spur candidate is inserted into a min-heap, and the next shortest loopless route is chosen. Unlike the full Yen's algorithm, which removes all conflicting prefix edges for every previously found path, we apply a lightweight edge-exclusion rule to maintain efficiency. The process continues until k distinct simple paths are generated or no further candidates remain.

4.3. Heuristic K Shortest Paths (Diversity-Optimized)

Unlike the exact version, the heuristic variant aims to generate **meaningfully different** alternatives rather than simply picking the next shortest route. Using the same simplified Yen-style spur framework, each candidate path is scored using a **composite weighted cost**:

$$\text{score} = W_{\text{len}} \cdot (\text{path length}) + W_{\text{div}} \cdot (\text{overlap penalty} \times \text{distance penalty}).$$

The **overlap penalty** counts how many previously selected paths share more than the given `overlap_threshold` (typically 20–80%) of edges with the candidate. The **distance penalty** measures how far the route deviates from the globally shortest path. By tuning these weights, we can prioritize either optimality or diversity, producing Google-Maps-style alternatives that remain short but visually distinct.

4.4. Approximate Shortest Paths

This mode focuses on producing **extremely fast** distance estimates for large batches of queries. Unlike standard evaluation based on MSE, Phase 2 uses an **allowed percentage-error threshold** to judge accuracy. Our approach incorporates aggressive pruning strategies such as limited Dijkstra expansions, early stopping, and lightweight heuristics. These techniques achieve a **10–20× speed-up** over the Phase 1 shortest-path implementation, while keeping the error within the prescribed threshold (typically 5–15%). Each query also follows strict time-budget constraints, and any query exceeding them must be rejected.

4.5. Evaluation Metrics

Phase 2 performance is evaluated based on:

- **Accuracy** — distance error must remain below the allowed percentage threshold.
- **Diversity** — heuristic KSP paths must avoid excessive edge overlap while staying close to the optimal distance.
- **Efficiency** — each query must complete within the given time limits (5–15 s).

4.6. Testing

No automated generator was created for Phase 2. Instead, logical correctness was verified using hand-crafted testcases for all query types (exact KSP, heuristic KSP, and approximate shortest paths), ensuring correct behaviour on both small and moderately sized graphs.

5. Phase 3 — Delivery Scheduling

5.1. Introduction

Phase 3 of the project was the point where the problem started feeling much closer to what companies like Swiggy, Zomato, or Uber deal with on a day-to-day basis. We were expected to build a delivery scheduling system that could plan routes over more than 5000 pickup–dropoff locations, adapt to sudden disruptions, and still complete the computation under a strict 15–second time limit. While this sounds manageable in theory, we quickly realised that any unexpected event—something as simple as a driver suddenly becoming unavailable—could invalidate a significant portion of the schedule we had already computed.

Compared to the earlier phases, this stage required us to treat routing as a *moving target*. It wasn't simply a shortest-path or static VRP exercise anymore. We needed a solution that stayed stable even when things changed halfway through computation. Three issues especially complicated the problem:

- **Driver Accidents:** If a driver is removed mid–route, all their assigned tasks instantly become the responsibility of the remaining drivers. Redistributing them without disturbing working routes is non-trivial.
- **Order Cancellations:** Removing an order is not just a matter of deleting it. We must ensure that the surrounding route structure remains coherent, and that we are not leaving inefficient gaps.
- **Price Prioritization:** Orders do not have equal importance. High-value tasks deserve earlier placement or more favourable routing, which forces the scheduling system to balance time optimisation with profit optimisation.

We did not want to depend blindly on one "standard" algorithm, so we first built a set of baseline approaches. These helped us understand the natural lower bound on runtime and the general behaviour of simpler heuristics. After that, we explored a series of more complex hybrid methods—ranging from geometric clustering ideas to biological meta–heuristics. A few of these showed potential, but many turned out to be too slow, too fragile, or too complicated to reliably handle dynamic events.

Eventually, we found that a hybrid centred around **Large Neighborhood Search (LNS)** paired with our Parallel Cheapest Insertion strategy gave us the best balance of speed, robustness, and implementation simplicity. This report documents that entire exploration: the baselines, the failed experiments, and the reasoning behind the final selected method.

5.2. Establishment of Baselines

Before attempting high-level optimisation, we implemented four baseline algorithms that served as reference points. These baselines answered two key questions for us:

1. What is the fastest possible time in which *any* complete schedule can be generated?
2. How do simple heuristics behave when faced with a large number of orders and dynamic constraints?

Each baseline offered a different perspective on the trade-off between speed, quality, and flexibility.

5.3. Baseline A: Greedy Nearest Neighbor

Idea: This approach models a very intuitive human strategy: at every step, the driver picks whichever unserved location is closest and heads there. There is no notion of global optimisation—just repeated local decisions.

Why we tried it: It is almost impossible to beat this algorithm in pure speed. Given our 15–second limit, understanding this "speed floor" was important for evaluating more complex ideas.

What went wrong: Its biggest issue is shortsightedness. The algorithm tends to get "pulled" toward small, nearby requests and may wander far away from high-priority or better-paying orders. When this happens early in the route, the driver later needs to travel long distances to compensate for a poor early decision. So while the runtime was excellent, the overall route quality was consistently poor.

5.3.1 Baseline B: The Sweep Algorithm

Idea: Treat the depot or central reference point as the centre of a circle. Then rotate a ray around it like the hand of a clock. As this sweeping line encounters orders, it assigns them to different drivers, effectively partitioning the space into angular slices.

Why we tried it: It is a well-known clustering method that tends to produce neatly separated regions, reducing unnecessary overlap between drivers.

Problems encountered: Real cities do not follow perfect geometric layouts. Two points that appear close in angle may be separated by flyovers, lakes, restricted turns, or one-way roads. Also, the strict angular partitions created rigid boundaries between drivers. This became a serious issue when some regions had many more orders than others, as the algorithm offered no clean way to shift excess load into a neighbouring sector.

5.3.2 Baseline C: Clarke–Wright Savings

Idea: Start by assuming each order has its own route. Then repeatedly merge routes that offer the greatest reduction ("savings") in total travel distance.

Why we tried it: It is a classic heuristic for vehicle routing and tends to perform quite well in static, deterministic scenarios.

Limitations: The dynamic events in our project made this approach brittle. When a driver fails or an order is cancelled, the savings structure gets disrupted. Recomputing the savings matrix or repairing partially merged routes becomes expensive. In many cases, we found ourselves close to restarting the entire computation—something we obviously cannot afford inside a 15–second window.

5.3.3 Baseline D: Parallel Cheapest Insertion

Idea: For every unassigned order, try inserting it into every possible position in every driver's route. Choose the insertion that increases the overall cost the least. Repeat this until all orders are assigned.

What worked well: This approach gave us significantly cleaner routes than the previous baselines. It also naturally integrates price priority—by sorting orders before insertion, we can ensure valuable orders secure the best positions.

Main limitation: It is a constructive heuristic, meaning it never revisits or corrects earlier choices. If a poor insertion is made early on, the algorithm has no built-in mechanism to fix that mistake later.

5.4. Experimental Hybrid Strategies

Once the baselines helped us understand the landscape, we experimented with more advanced strategies that combine construction heuristics and iterative improvement. Many of these methods are standard in research literature, but their practical performance under our constraints varied significantly.

5.4.1 Experiment 1: Geometric Clustering + Local Search

We attempted to refine the Sweep-based clusters using a local search algorithm such as Simulated Annealing. The idea was to use Sweep for coarse grouping and then optimise within each group.

This worked decently when the distribution of orders was uniform, but problems surfaced when clusters had imbalanced workloads. Because the Sweep boundaries were rigid, the algorithm struggled to pass work across them without large structural changes.

5.4.2 Experiment 2: Ant Colony Optimization

We implemented a simplified ACO where artificial "ants" traverse the graph, depositing pheromones on good edges. Over iterations, stronger edges emerge.

The problem was convergence speed. With over 5000 nodes, the pheromone matrix was practically empty within our time limit. Even after many attempts, the ants simply didn't have enough time to build consistent patterns. Theoretical appeal aside, ACO was not suitable for a 15–second runtime.

5.4.3 Experiment 3: Tabu Search

Tabu Search encourages exploration by temporarily forbidding recently taken moves. It is powerful for escaping local minima.

However, the interactions with dynamic events created difficult edge cases. For example, an order might be marked "tabu" at the same moment the driver holding it is removed due to an accident. These contradictions required complex exception handling, which made the implementation fragile.

5.4.4 Experiment 4: Genetic Search

We generated a population of random schedules and attempted to evolve improved ones by recombining segments of different routes.

While promising conceptually, this approach struggled in practice. Combining segments from two different routes often produced invalid offspring—for example, a schedule that contains a dropoff without the corresponding pickup. We ended up spending most computation time repairing routes rather than improving them.

5.4.5 Experiment 5: Adaptive LNS

Adaptive LNS dynamically assigns weights to different heuristics and learns which ones perform the best.

This method is powerful but requires significant bookkeeping—tracking scores, updating weights, maintaining operator histories, etc. The overhead was too high for our time window, leaving fewer cycles for the actual optimisation loop. In our case, simpler turned out to be more effective.

5.5. The Selected Solution: Large Neighborhood Search (LNS)

After evaluating all the methods above, we chose **Large Neighborhood Search (LNS)** as the final solution. It provided a good compromise between implementation difficulty, execution speed, and adaptability to dynamic events.

5.5.1 Mechanism: Ruin and Recreate

LNS works in an iterative loop. Each iteration has two major steps:

1. **Ruin:** Randomly remove a subset (typically 15–20%) of orders from the current schedule. This forces the algorithm to reconsider parts of the route.
2. **Recreate:** Reinsert these removed orders using Parallel Cheapest Insertion.

The temporary "damage" caused by the ruin step frees the algorithm from local optima, while the recreate step systematically rebuilds a better schedule. Because each iteration is fast, many such refinements can be performed within our time limit.

5.5.2 Handling Dynamic Events

Another major advantage of LNS is that it naturally handles dynamic events without requiring special-case logic.

Driver Accidents: When a driver becomes unavailable, their entire route is simply treated as a ruin event. All their orders are moved back into the unassigned pool. The recreate phase then redistributes them across the remaining drivers.

Order Cancellations: Cancellations behave like small, targeted ruin operations. Removing one or two nodes does not destabilise the larger structure.

Price Priority Integration: Before reinserting tasks during the recreate phase, we sort the candidates by price. Higher-value orders get inserted first, giving them priority in selecting the most efficient positions.

5.6. Performance and Complexity Analysis

5.6.1 Time Complexity

- **Initial Construction ($O(N^2)$):** Using Parallel Cheapest Insertion, we can produce a workable initial schedule in under a second, even with several thousand orders.
- **Optimisation Loop ($O(K \cdot N)$):** Since K (the number of removed tasks) is a small fraction of N, each iteration is fast. This allows us to run hundreds or thousands of improvement cycles within the remaining time budget.

5.6.2 Final Comparison

Algorithm	**Implementation Difficulty**	**Dynamic Handling**	**Final Verdict**
Greedy NN	Very Low	Good	**Inefficient**
Sweep Algo	Low	Poor	**Inflexible**
Clarke-Wright	Medium	Very Poor	**Brittle**
Ant Colony	High	Good	**Too Slow**
Tabu Search	High	Fair	**Complex/Buggy**
Genetic Search	Very High	Poor	**Unstable**
Adaptive LNS	High	Excellent	**Too Heavy**
LNS (Selected)	Moderate	Excellent	**Chosen Method**

5.7. Implementation of LNS in Phase-3

The Phase 3 system applies Large Neighborhood Search (LNS) on top of the shortest-path engine developed in earlier phases. The implementation focuses on three aspects: fast cost evaluation, stable schedule construction, and time-bounded iterative improvement.

5.7.1 Cost Model and Preprocessing

All routing costs use the `average_time` field from Phase 1. Since Phase 3 requires thousands of evaluations per iteration, we avoid computing full paths and instead rely only on the returned metric (minimum travel time). This treats the graph as a metric space and keeps cost lookups constant-time.

5.7.2 Initial Construction (Parallel Cheapest Insertion)

A complete feasible schedule is built using a Parallel Cheapest Insertion routine:

- Every order is inserted at the position that adds minimum marginal cost.
- Precedence is enforced by checking pickup-before-dropoff validity.
- Insertion is attempted across *all* drivers, enabling natural load balancing.

This produces a stable baseline solution within a second.

5.7.3 Ruin-and-Recreate Cycle

Each LNS iteration:

- Removes 15–20% of orders as whole pickup–dropoff pairs.
- Reinserts them using the same PCI kernel, but sorted in descending order of price.

This encourages profitable orders to obtain better placements and helps the system escape poor local minima.

5.7.4 Dynamic Event Integration

Driver failures and cancellations are integrated seamlessly:

- A failed driver triggers a full-route ruin; all tasks return to the unassigned pool.

- Cancelled orders are simply removed; the next recreate step repairs the schedule.

No separate handling logic is required for different disruption types.

5.7.5 Time-Bounded Iterations

Each iteration is timed, and the algorithm terminates whenever continuing risks crossing the 15-second limit. Because each iteration is lightweight, the system typically executes hundreds of refinement steps within the allowed time.

5.7.6 Final Output Assembly

Routes are compacted, validated, and annotated with completion times and assigned order sets to conform to the Phase 3 output format.

5.8. Conclusion

Phase 3 required us to go beyond straightforward routing algorithms and look at approaches that remain stable under constant change. While several advanced techniques looked appealing in theory, they were either too slow to converge, too complex to maintain under dynamic events, or too heavy for a tight 15–second runtime.

Large Neighborhood Search provided the best overall balance. It was fast enough to iterate thousands of times, flexible enough to absorb driver failures and cancellations, and simple enough to implement without a large amount of specialised logic. By continuously breaking and repairing the solution, LNS allowed us to steadily improve route quality while handling all required dynamic events. In the context of this project, it proved to be the most reliable and practical choice.

6. Test Case Generation and Benchmarking

6.1. Python Generators

We created some testcase generators for Phase-1, they are present in the tests subfolder of Phase-1 and are respectively:

- `generate_graph.py`
- `generate_queries.py`
- `generate_phase1_tests.py`

In addition to these, there is another python file: `checker_phase1_full.py` that compares the output of the code with the correct expected output which it generates by using python.
The process of utilizing them is explained in the last section of Phase-1.

7. Complexity and Performance Summary

Query Type	**Time Complexity**	**Space Complexity**
remove_edge	$O(1)$	$O(1)$
modify_edge	$O(1)$	$O(1)$
shortest_path (distance/time)	$O((V + E)\log V)$	$O(V + E)$
knn (euclidean)	$O(V)$	$O(V)$
knn (shortest path)	$O((V + E)\log V)$	$O(V)$
k-shortest paths (exact)	$O\big(kV(E + V\log V)\big)$	$O(V + E)$
approx shortest path	$O\big(((V + E)\log V)/c\big)$	$O(V)$
delivery scheduling	NP-hard; heuristic $O(n^2)$	$O(n)$

Extra Information:
The KNN (Euclidean) query uses a **layered grid search**.
So, knn (euclidean) has the average time complexity of: $O(klogk + b)$ and average space complexity is: $O(B)$, where B is the number of nodes in the searched bucket.
The above mentioned complexities are for the worst-case.

8. Assumptions and Limitations

- Speed profile always has 96 slots (15-minute intervals).
- Graph is simple (no multi-edges between same pair except direction).
- Latitude/Longitude treated as Euclidean for small regions.
- The conversion factor that is used to approximate the number of meters in one degree of latitude is assumed to 111000.
- Queries handled sequentially.

9. AI Usage and Acknowledgment

AI tools (including ChatGPT) were used primarily for:

- Code optimization strategy (cache-efficient Dijkstra)
- Structuring report and documentation
- JSON test data generation scripts
- Researching on which algorithms to use for Phase-3

All AI interactions were used as guidance and verified manually.

Attached below are the five links for the AI Chats used:

- **General Purpose**
- **Implementation**
- **Research**
- **Testing Phase-1**
- **Testing Phase-2**