DSA PROJECT

August 2025

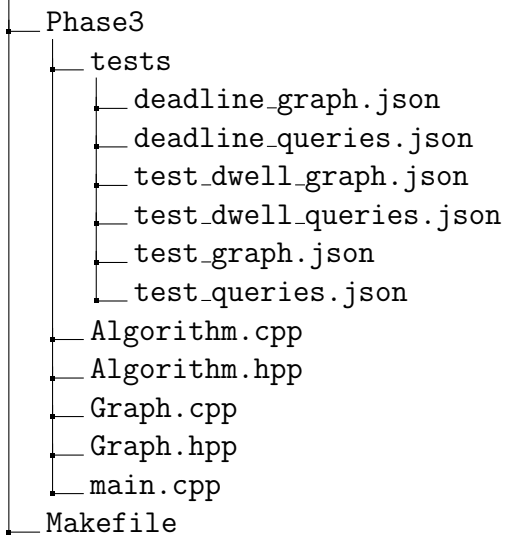
Shrijani Korepaka (24B0905)
Varshika Cheemala (24B0944)
Vyshnav Thumu (24B0911)

Contents

1 Directory Structure of Code

1.1 Directory Structure

```
Phase1
    tests
        generate_test.py
        simple_queries.json
        simple_test.json
        test_commuter_graph.json
        test_commuter_queries.json
        test_graph.json
        test_queries.json
        test_river_graph.json
        test_river_queries.json
    Algorithm.cpp
    Algorithm.hpp
    Graph.cpp
    Graph.hpp
    main.cpp
Phase2
    tests
        approx_graph.json
        approx_queries.json
        cycle_graph.json
        cycle_queries.json
        diff_graph.json
        diff_queries.json
        overlap_graph.json
        overlap_querie.json
        test_graph.json
        test_queries.json
    Algorithm.cpp
    Algorithm.hpp
    Graph.cpp
    Graph.hpp
    main.cpp
```



1.2 Purpose of all Files and Folders

1.2.1 Phase 1/

Contains all the implementations of Phase-1 as well as the tests/ directory for test case infrastructure.

- **tests/** : The tests/ folder contains a python script which generates the test cases and few manual test cases to check the correctness of the Phase 1 implementation at every edge case
- **Algorithm.cpp/.hpp** : Contains the main implementation of graph algorithms for your routing system. Modified Dijkstra's algorithm into A^* algorithm (using priority_queue/min-heap) for shortest distance/-time queries with constraints like banned nodes, forbidden road types etc.
- **Graph.cpp/.hpp** : Manages the core graph data structures and dynamic updates like defining node and edges, adding/removing/updating edges and nodes, marking edges disabled/enabled etc.
- **main.cpp** : This file loads graph from a JSON file, reads and interpret all the queries in queries.json files, executes/calls the functions depending on each query type and finally writes results to output.json

1.2.2 Phase 2/

Contains all the implementations of Phase-2 as well as the tests/ directory for test case infrastructure to check correctness of implementation.

- **tests/** : This folder contain all the test cases used to check if the Phase 2 implementation is efficient or not. This includes all the corner and edge test cases.
- **Algorithm.cpp/.hpp** : Implements 3 tasks
    1. **Task 1 (Yen's K-Shortest Paths)** : Find K distinct shortest routes with no loops
    2. **Task 2 (Heuristic K-Shortest Paths)** : Faster alternative that penalizes overlapping in the path
    3. **Task 3 (Approximate Shortest Paths)** : Estimates shortest distance between source and target (without running Dijkstra) which prioritize speed over accuracy
- **Graph.cpp/.hpp** : Manages the core graph data structures and dynamic updates like defining node and edges, adding/removing/updating edges and nodes, marking edges disabled/enabled etc.
- **main.cpp** : This file loads graph from a JSON file, reads and interpret all the queries in queries.json files, executes/calls the functions depending on each query type and finally writes results to output.json

1.2.3 Phase 3/

Contains all the implementations of Phase-3 as well as the tests/ directory for test case infrastructure to check correctness of implementation.

- **tests/** : This folder contain all the test cases used to check if the Phase 3 implementation is efficient or not. This includes all the corner and edge test cases.
- **Algorithm.cpp/.hpp** : This file implements the routing, scheduling, and cost-based assignment logic for a delivery-style system.
- **Graph.cpp/.hpp** : Manages the core graph data structures and dynamic updates like defining node and edges, adding/removing/updating edges and nodes, marking edges disabled/enabled etc.

- **main.cpp** : This file loads graph from a JSON file, reads and interpret all the queries in queries.json files, executes/calls the functions depending on each query type and finally writes results to output.json

1.2.4 Makefile

Compiles all .cpp files and generates executables as required

2 Makefile and Targets

Makefile

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -O2 -g

# --- Phase 1 Definitions ---
PHASE1_DIR = Phase1
PHASE1_INC = -I./Phase1
PHASE1_SRC = $(PHASE1_DIR)/main.cpp \
             $(PHASE1_DIR)/Graph.cpp \
             $(PHASE1_DIR)/Algorithm.cpp
PHASE1_OBJ = $(PHASE1_SRC:.cpp=.o)
PHASE1_TARGET = phase1

# --- Phase 2 Definitions ---
PHASE2_DIR = Phase2
PHASE2_INC = -I./Phase2
PHASE2_SRC = $(PHASE2_DIR)/main.cpp \
             $(PHASE2_DIR)/Graph.cpp \
             $(PHASE2_DIR)/Algorithm.cpp
PHASE2_OBJ = $(PHASE2_SRC:.cpp=.o)
PHASE2_TARGET = phase2

# --- Phase 3 Definitions ---
PHASE3_DIR = Phase3
PHASE3_INC = -I./Phase3
PHASE3_SRC = $(PHASE3_DIR)/main.cpp \
             $(PHASE3_DIR)/Graph.cpp \
             $(PHASE3_DIR)/Algorithm.cpp
PHASE3_OBJ = $(PHASE3_SRC:.cpp=.o)
PHASE3_TARGET = phase3

# --- Targets ---
.PHONY: all clean phase1 phase2 phase3

# Default target builds all three phases
all: phase1 phase2 phase3

# --- Phase 1 Build Rules ---
phase1: $(PHASE1_OBJ)
    $(CXX) $(CXXFLAGS) $(PHASE1_INC) -o $(PHASE1_TARGET) $(PHASE1_OBJ)
    @echo "Build successful: $(PHASE1_TARGET)"

$(PHASE1_DIR)/%.o: $(PHASE1_DIR)/%.cpp
    $(CXX) $(CXXFLAGS) $(PHASE1_INC) -c $< -o $@

# --- Phase 2 Build Rules ---
phase2: $(PHASE2_OBJ)
    $(CXX) $(CXXFLAGS) $(PHASE2_INC) -o $(PHASE2_TARGET) $(PHASE2_OBJ)
    @echo "Build successful: $(PHASE2_TARGET)"

$(PHASE2_DIR)/%.o: $(PHASE2_DIR)/%.cpp
    $(CXX) $(CXXFLAGS) $(PHASE2_INC) -c $< -o $@

# --- Phase 3 Build Rules ---
# UPDATED: Now links Phase2/Algorithm.o alongside Phase3 objects
phase3: $(PHASE3_OBJ) $(PHASE2_DIR)/Algorithm.o
    $(CXX) $(CXXFLAGS) $(PHASE3_INC) -o $(PHASE3_TARGET) $(PHASE3_OBJ) $(PHASE2_DIR)/Algorithm.o
    @echo "Build successful: $(PHASE3_TARGET)"

$(PHASE3_DIR)/%.o: $(PHASE3_DIR)/%.cpp
    $(CXX) $(CXXFLAGS) $(PHASE3_INC) -c $< -o $@

# --- Utilities ---
clean:
    rm -f $(PHASE1_DIR)/*.o $(PHASE2_DIR)/*.o $(PHASE3_DIR)/*.o
    rm -f $(PHASE1_TARGET) $(PHASE2_TARGET) $(PHASE3_TARGET)
    rm -f output.json
    @echo "Clean complete!"
```

Figure 1: Makefile

2.1 Targets of Makefile

- **phase1** : Builds the main.cpp executable by compiling all necessary files in Phase1 and links the executable.
- **phase2** : Builds the main.cpp executable by compiling all necessary files in Phase2 and links the executable.
- **phase3** : Builds the main.cpp executable by compiling all necessary files in Phase3 and links the executable.
- **clean** : Removes all generated object files and executables including output.json to clean the build environment.
- **Compilation rules** :
  - Compiles all `.cpp` files to `.o` files

3 Assumptions

3.1 Phase 1

No assumptions made

3.2 Phase 2

No assumptions made

3.3 Phase 3

- We assumed the graph is strongly connected. If a path does not exist between two nodes (e.g., due to removed edges or one-way traps), the distance returns INF and the order is marked as failed rather than crashing the system
- All drivers originate from a single, fixed depot_node at time $t = 0$
- All drivers are identical in terms of speed (1x multiplier), capacity, and starting conditions
- If a driver is working for any hours then his efficiency decreases denoted by fatigue factor

4 Test Cases and Analysis

4.1 Phase 1

To check correctness of of Phase 1 implementation, I generated test cases using python script `generate_test.py` and also used manual test cases in the files

4.1.1 Manual test cases

- `simple_test.json` and `simple_queries.json`
- `test_commuter_graph.json` and `test_commuter_queries.json`
- `test_river_graph.json` and `test_river_queries.json`

4.1.2 Python generated test cases

1. **Graph Generation Characteristics** Generated graphs are stored in `test_graph.json`

Property	**Value**
No.of nodes	100
No.of edges	300
Location bounds	Mumbai (lat 19.0–19.3, lon 72.8–73.0)
Speed profile	96 slots × 15 minutes
Oneway probability	20%
POIs	Random selection of POIs types and 0–2 categories
Road types	primary / secondary / tertiary / local / expressway

2. **Query Generation Characteristics** The generated queries are stored in `test_queries.json` and contains
    - 50–100 shortest path queries
    - 30–40 KNN queries
    - 10–20 dynamic updates (remove & modify)
    - Random constrained shortest paths
    - Queries mixed in random order to test update consistency

- Dynamic updates randomly modify length, average_time, speed_profile and road type

4.2 Phase 2

Created few manual test cases to check correctness of Phase 2 which are in `tests/` directory

4.2.1 Manual test cases

- `test_graph.json` and `test_queries.json` is a simple and basic test case to check the implementation
- `overlap_graph.json` and `overlap_querie.json` is a test case that checks if Heuristic K - shortest path correctly penalizes overlapping in paths
- `approx_graph.json` and `approx_queries.json` is a test case that check correctness of task 3
- `cycle_graph.json` and `cycle_queries.json` is a test case that checks the correctness of Phase 2 when a cycle is present in the graph
- `diff_graph.json` and `diff_queries.json` is to check if task 1 and task2 are implemented properly and task2 gives different output from task1

4.3 Phase 3

Tested the correctness of Phase 3 implementation by tesing on few randomly created test cases

5 Python Scripts and Libraries

5.1 Phase 1

Created Python scripts (`generate_test.py`) to generate synthetic test cases for validating the correctness of Phase-1 implementation

5.1.1 Libraries used

Library	Purpose
`json`	Writing graph and query data to JSON files
`random`	Generating random nodes, speeds, POIs, etc.
`math`	Accurate distance computation using latitude–longitude

5.1.2 Scripts Included

- `generate_test.py` that generates
  - A realistic graph
  - A set of queries
  - Dynamic update operations
  - KNN and shortest-path tasks
- Output files
  - `test_graph.json`
  - `test_queries.json`

5.1.3 Commands to run the script

- Command to run the generator

```
python3 generate_test.py
```

- Command to test Phase 1 with generated test files

```
./phase1 test_graph.json test_queries.json output.json
```

5.1.4 Assumptions Made in Test Generation

1. Geographic Assumptions
   - All points lie within `Latitude:19.0-19.3` and `Longitude:72.8-73.0`

- Used the following approximations while calculating distance
  - 1° latitude $\approx 110{,}540\,\text{m}$
  - 1° longitude $\approx 111{,}320\,\text{m} \times \cos(\text{latitude})$

2. Graph Structure Assumptions
   - The graph is always connected
   - No self loops or duplicate edges
   - Oneway edges appear with 20% probability
   - Speed profiles contain 96 entries (15-minute buckets for 24 hours)
3. Traffic Modelling Assumptions
   - Rush hours (7-10 AM & 5-8 PM) reduce speeds to 50-80% randomly
   - Off-peak hours vary between 90-110%
   - Average speeds uniformly selected from 20-60 m/s
4. Query Generation Assumptions
   - Shortest-path queries :
     - Random source & target
     - Mode randomly chosen among "distance" or "time"
     - Constraints applied with 30% probability
   - KNN-queries :
     - Query point is a random perturbation around a real node
     - POIs selected from allowed set
     - k chosen between 3 to 10
   - Dynamic Updates :
     - Random modify_edge and remove_edge operations
     - Modify patches may change length, average_time, speed_profile, or road_type
   - Event Ordering :
     - All events appear in sequence
     - Updates affect all subsequent queries

5.2 Phase 2

No Python script used

5.3 Phase 3

No Python script used

6 Time and Space Complexity

6.1 Phase 1

Functions involved in the implementation of Phase 1 :

- shortestPathDistance()
- shortestPathTime()
- findNearestNode()
- knnEuclidean()
- knnShortestPath()

6.1.1 shortestPathDistance - Dijkstra

Time Complexity

- Used priority queue (min - heap)
- Each edge is relaxed at most once (similar to Dijkstra)
- Heuristic = Euclidean distance
- Time complexity of Dijkstra

$$\boxed{O((V+E)logV)}$$

Space Complexity

- unordered_map<int, double> dist → up to V entries

- unordered_map<int, int> parent → up to V entries
- Priority queue that may push each relaxed edge → up to E elements

$$\begin{aligned} \text{dist} &\rightarrow O(V) \\ \text{parent} &\rightarrow O(V) \\ \text{PQ} &\rightarrow O(E) \\ \text{Graph (adjacency lists)} &\rightarrow O(V+E) \\ \text{Total auxiliary} &\rightarrow O(V+E) \end{aligned}$$

- So the space complexity is

$$\boxed{O(V+E)}$$

6.1.2 shortestPathTime - Dijkstra

Time Complexity

- Uses priority queue
- Relaxing each edge
- getEdgeTime() is more expensive : Worst case loops through all 96 slots

$$O(96) = O(1)$$

- So time complexity remains same as Dijkstra

$$\boxed{O((V+E)logV)}$$

Space Complexity

- Same as shortestPathDistance i.e. A^*/Dijkstra

$$\boxed{O(V+E)}$$

6.1.3 findNearestNode

Time Complexity

- This function loops over all nodes once. So, time complexity is

$$\boxed{O(V)}$$

Space Complexity

- No data structure used so

$$\boxed{O(1)}$$

6.1.4 knnEuclidean - using Euclidean distance

Time Complexity

- Outer loop iterates over V $\rightarrow$ V
- Each time a loop runs, it checks POIs $\rightarrow$ P
- And each time it maintains max-heap of size k $\rightarrow$ $O(logk)$
- Therefore, the time complexity is

$$\boxed{O(V.(P + logk))}$$

Space Complexity

- Stores a fixed size heap of size k so

$$\boxed{O(k)}$$

6.1.5 knnShortestPath - Dijkstra with early stopping

Time Complexity

- You run Dijkstra once, stopping after finding k POI nodes
- Worst-case : No POIs (explore entire graph) $\rightarrow$ $O((V + E)logV)$
- Best-case : POIs found very close to start node (very few nodes expanded) $\rightarrow$ $O(klogV)$

- So the average case lies somewhere in between Best-case and Worst-case depending on the POI distribution.

$$\boxed{\begin{aligned}&\text{Worst-case} = O((V+E)logV)\\&\text{Best-case} = O(klogV)\end{aligned}}$$

Space Complexity

- Distance vector of size V $\rightarrow O(V)$
- Priority queue of size E $\rightarrow O(E)$
- So, the space complexity is

$$\boxed{O(V+E)}$$

6.2 Phase 2

6.2.1 Task 1 (Yen's Algorithm)

Time Complexity

- Heuristic Computation $\rightarrow O((V+E)logV)$
- Outer loop runs K times $\rightarrow$ O(K.(inner loop))
- The inner loop iterates through the nodes of the previous path (L_{path})
- So A^* runs in the inner loop L_{path} times $\rightarrow O(L_{path}(V+E)logV)$
- So the total time complexity is

$$\boxed{O(K.L_{path}(V+E)logV)}$$

- In the worst case, where the path length $L_{\text{path}} \approx V$

Space Complexity

- Graph storage $\rightarrow O(V+E)$
- Each stored path has worst case complexity of $O(V)$
- And we need to store K paths $\rightarrow O(K.V)$
- So the total time complexity is

$$\boxed{O(V+E+(K.V))}$$

6.2.2 Task 2 (Heuristic K-Shortest Paths)

Time Complexity

- The loop runs K times and in each iteration A^* is being run
- A^* complexity $\rightarrow O((V+E)logV)$
- so, the total complexity is

$$\boxed{O(K.(V+E)logV)}$$

- This is faster than Yen's algorithm because L_{path} is not multiplied

Space Complexity

- Graph storage $\rightarrow O(V+E)$
- Each stored path has worst case complexity of $O(V)$
- In penalty map in worst case penalizes over all the edges $\rightarrow O(E)$
- And we need to store K paths $\rightarrow O(K.V)$
- So the total time complexity is

$$\boxed{O(V+E+(K.V))}$$

6.2.3 Task 3 (Approximate Shortest Path)

Time Complexity

1. Precomputation
   - Loops L_{marks} times where $L_{\text{marks}} = \sqrt{V}$
   - For each landmark, it runs one Forward Dijkstra and one Backward $A^* \rightarrow O(2.(V+E)logV)$
   - So the total complexity is

$$\boxed{O(L_{marks}.(V+E)logV)}$$

2. Query

- It performs a lookup for the source and target against all landmarks so the complexity is

$$\boxed{O(L_{marks}) = O(\sqrt{V})}$$

Space Complexity

- We need to store two distance matrices
  1. landmarks_to_nodes: Size $L_{marks} \times V$
  2. nodes_to_landmarks: Size $V \times L_{marks}$
- Total complexity is

$$\boxed{O(V \times L_{marks}) = O(V\sqrt{V})}$$

6.3 Phase 3

6.3.1 Driver Complexity

Time Complexity

1. recalculateRouteTimesFrom :
   - Iterates linearly through the route (R stops)
   - Performs a distance query for each leg and let Q be the cost of the distance query
   - So complexity is

$$\boxed{O(R \cdot Q)}$$

2. findBestInsertion :
   - Outer loop iterates through every possible Pickup position p (1 to R) and every Dropoff position d (p to R). This results in $\frac{R^2}{2}$ iterations
   - Inner operations include Copying the route vector ($O(R)$) and simulating the route (calculating arrival times) ($O(R \cdot Q)$) So the complexity is

$$\boxed{O(R^2 \cdot (R + R \cdot Q)) \approx O(R^3 \cdot Q)}$$

Space Complexity

- Route Storage : $O(R)$
- Temporary Route Copies : During insertion checks, temporary vectors of size R are created. Space: $O(R)$

6.3.2 Scheduler Complexity

Time Complexity

1. Initialization (loadOrders) :
   - Sorting orders : $O(N \log N)$
   - Landmark Precomputation : $O(\sqrt{V} \cdot E \log V)$
2. Main Scheduling Loop (run) :
   - Iterates through N orders
   - Driver Selection : A priority queue (Heap) manages drivers
   - Driver Checks: The code limits checks to max_checks (set to 50). This is a constant K
   - For each checked driver, it runs findBestInsertion
   - Complexity per Order : $K \cdot O(R^3 \cdot Q)$
   - So total complexity is (Assuming constant number of driver checks)

$$\boxed{O(N \cdot R^3 \cdot Q)}$$

3. Result Generation (getResults) :
   - reconstructs full path so the complexity is

$$O(N \cdot R \cdot E \log V)$$

Space Complexity

- Order Storage : $O(N)$
- Driver Heap : $O(D)$
- Total : $O(N + D)$

7 Approach for Each Query Type

7.1 Phase 1

7.1.1 Query 1 : Dynamic Update: Remove Edge

- **Algorithm used :** Soft Deletion
- **Approach :** When a request to remove an edge is received, we do not physically erase the object from memory or the adjacency vector. Instead, we toggle a boolean flag is_removed = true inside the Edge object
- Physically removing elements from vectors or maps can be computationally expensive ($O(N)$ for vectors) and invalidates iterators. Soft deletion is $O(1)$ and allows routing algorithms to simply check if (!edge.is_removed) during traversal, maintaining high performance during batch updates

7.1.2 Query 2 : Dynamic Update: Modify Edge

- **Algorithm used :** Soft Deletion
- **Approach :** Do not physically erase the object from memory or the adjacency vector. Instead, we toggle a boolean flag is_removed = true inside the Edge object
- Physically removing elements from vectors or maps can be computationally expensive ($O(N)$ for vectors) and invalidates iterators. Soft deletion is $O(1)$ and allows routing algorithms to simply check if (!edge.is_removed) during traversal, maintaining high performance during batch updates

7.1.3 Query 3 : Shortest Path (Distance)

- **Algorithm used :** A^* algorithm
- **Approach :** Implemented A^* using Min-priority Queue to explore the graph which selects the node with the lowest $f(n) = g(n) + h(n)$, where $g(n)$ is the actual distance from the source and $h(n)$ is the heuristic

- Used the Euclidean Distance between the current node and the target node as the heuristic. Since Euclidean distance represents the straight-line distance, it is a consistent heuristic, ensuring the algorithm finds the optimal path while expanding fewer nodes than Dijkstra
- Dynamic constraints (forbidden nodes and forbidden road types) are checked during the edge relaxation step. If a node or edge violates the constraints, it is skipped.

7.1.4 Query 4 : Shortest Path (Time)

- **Algorithm used :** Dijkstra's Algorithm
- **Approach :** For time minimization, we utilized standard Dijkstra's Algorithm. The priority queue orders nodes purely based on the accumulated time from the source. We did not use A^* because A^* for time is difficult because deriving a valid admissible heuristic is non-trivial (variable speed limits and road conditions make it hard to estimate minimum remaining time)
- Edge weights are calculated dynamically based on the average_time or time-dependent speed profiles provided in the graph data

7.1.5 Query 5 : K-Nearest Neighbors

- **Algorithm used :** No specific algorithm direct Linear scan with Max-Heap
- **Approach :** To find the k nearest Points of Interest (POIs) based on distance
  - Iterate through all nodes in the graph to identify those containing the requested POI type
  - Calculate the Euclidean distance from the query coordinates to these nodes
  - Maintain a Max-Priority Queue of size k. If the queue is full and a new node is found with a smaller distance than the largest element in the queue, we pop the largest and push the new node.

7.1.6 Query 6 : K-Nearest Neighbors

- **Algorithm used :** Dijkstra's Algorithm with early exit
- **Approach :** To find the k nearest POIs based on actual road network distance
  - Identify the graph node closest to the user's query coordinates using a nearest-neighbor search
  - Initiate a Dijkstra search from this starting node
  - As Dijkstra expands nodes in strict increasing order of distance, we check every popped node to see if it contains the requested POI tag
  - The algorithm terminates immediately once we have collected k unique nodes containing the POI
- Dijkstra explores layers of the graph based on proximity, the first k POIs encountered are mathematically guaranteed to be the nearest ones via the road network. This avoids calculating all-pairs shortest paths and is highly efficient for small k

7.2 Phase 2

7.2.1 Query 1 : K-Shortest Paths (Exact)

- **Algorithm used :** Yen's Algorithm with an Optimization
- **Approach :** To find the k loopless shortest paths in increasing order of cost
  - Find the initial shortest path using A^*
  - Each subsequent path k, we iterate through the nodes of the previous best path. Each node acts as a "spur node" where we attempt to deviate from the known path
  - Lock the "root path" (from source to spur node) and temporarily remove edges used in previous solutions that share this root. This forces the algorithm to find a new, unique sub-path
  - New potential paths (Root + Spur Path) are stored in a candidate pool. The best candidate is selected as the next k-th path

- We optimized the internal search steps by precomputing a Reverse Graph to generate accurate heuristics, significantly speeding up the A^* searches inside the deviation loops

7.2.2 Query 2 : K-Shortest Paths (Heuristic)

- **Algorithm used :** Iterative Penalty Method
- **Approach :** To find k diverse paths quickly without the computational overhead of Yen's algorithm
  - Run Dijkstra's algorithm to find the current optimal path
  - Once a path is found and added to the results, we punish the edges participating in that path by multiplying their weights (initially by 1.5x, with subsequent growth of 1.2x)
  - Re-run Dijkstra on the graph with these modified, heavier weights. This naturally forces the algorithm to "avoid" the roads taken by previous paths and explore slightly longer, but distinct, alternatives
- By dynamically increasing the cost of used edges, we simulate avoiding overlapping routes while keeping the algorithm complexity near-linear per path

7.2.3 Query 3 : Approximate Shortest Path

- **Algorithm used :** Landmark-Based Triangulation
- **Approach :** To handle massive batches of queries within less time, we used a precomputation strategy
  - Select $\sqrt{N}$ "Landmark" nodes using Greedy Farthest Point Sampling (FPS). The first landmark is random, and subsequent landmarks are chosen to maximize the distance from the existing set, ensuring even coverage of the map latex
  - Compute and store the distances from all nodes to every landmark, and from every landmark to all nodes (using Forward and Backward Dijkstra)

  - For a query (u, v), instead of traversing the graph, we iterate through our landmarks. The approximate distance is calculated as $\min(dist(u \to L_k) + dist(L_k \to v))$. This effectively finds the shortest path that passes through a major hub
  - If the source and target are physically close, routing through a distant landmark creates error. To mitigate this, we compare the landmark result against a scaled Euclidean distance ($1.35 \times Euclidean$) and take the minimum

- This approach reduces the query time to $O(K)$ (where K is the number of landmarks), effectively $O(1)$ compared to graph size

7.3 Phase 3

8 Algorithmic Approach: Greedy Cheapest Insertion

We implemented a Greedy Cheapest Insertion Heuristic. This algorithm constructs routes incrementally. For every unassigned order, the system evaluates every feasible position in every driver's current route and selects the insertion that minimizes a weighted cost function.

Why this approach?

[cite$_start$]

- **Speed:** It avoids the exponential complexity of exhaustive search, fitting well within the 15-second time limit per query[cite: 77, 107].
- **Flexibility:** It easily accommodates complex, non-linear constraints (like driver fatigue and batching rules) which are difficult to model in linear programming.

9 Advanced Heuristics & Real-World Modeling

To move beyond a theoretical graph traversal and model a realistic delivery ecosystem, we identified and implemented six specific operational constraints.

[cite$_start]$These heuristics are the core contribution of our solution$[$cite : 95$].

9.1 Realistic Time Modeling (Prep & Dwell)

Standard algorithms often assume travel time is the only cost. We refined this by modeling "invisible" delays that affect real drivers:

9.1.1 Parallel Meal Preparation (Smart Batching)

- **Observation:** If a driver picks up two orders from the same restaurant, the waiting times overlap. They do not wait $Time_A + Time_B$.
- **Implementation:** We implemented logic where if consecutive stops are at the same node, the departure time is calculated as $\max(Arrival, ReadyTime_A, ReadyTi$ This significantly reduces idle time.

9.1.2 Batch Compatibility

- **Constraint:** To ensure food quality, we reject batching two orders if their preparation times differ by more than 15 minutes. This prevents one customer's food from getting cold while waiting for another's.

9.1.3 Gated Community Clustering

- **Observation:** Security checks at large tech parks or gated societies take significant time (e.g., 5 minutes).
- **Implementation:** We apply a "Gated Entry Penalty" (300s) only once per visit. If a driver delivers multiple orders within the same gated cluster, the penalty is waived for subsequent stops. This incentivizes the scheduler to group neighbor deliveries to a single driver.

9.1.4 Node-Specific Dwell Times

- **Implementation:** We added granular "last-mile" friction based on location type: +180s for Malls (parking difficulty) and +60s for Apartments (elevator delays).

9.2 Economic & Human Factors

We extended the objective function to optimize for business value and workforce sustainability:

9.2.1 Price-Based Priority

- **Mechanism:** We introduced a weighted cost function:

$$Cost = \Delta Time - (Price \times Weight)$$

- **Impact:** High-value orders effectively reduce the "cost" of a route, causing the greedy algorithm to prioritize them even if they require slightly longer travel. This maximizes revenue alongside efficiency.

9.2.2 Driver Fatigue Management

- **Mechanism:** We applied a quadratic penalty to the travel cost based on the driver's current shift duration.
- **Impact:** A driver nearing their 8-hour limit becomes "expensive" to the scheduler. This automatically routes long-haul trips to fresher drivers, ensuring load balancing and safety compliance without hard limits.

10 Future Scope: Regional Demand Prediction

While our current implementation optimizes strictly for known orders, we analyzed a theoretical extension for **Proactive Load Balancing** (not currently implemented).

- **The Concept:** A "Gravity Heuristic" could prevent drivers from ending their routes in low-demand dead zones.
- **Proposed Implementation:** By calculating the centroid of historical demand, we could add a penalty to any route that ends far from this hotspot:

$$Cost+ = DistanceToCenter \times \alpha$$

This would proactively position drivers near future potential orders, reducing "dead-head" (empty) travel in the long run.

11 Conclusion

Our solution successfully optimizes the Delivery Scheduling problem by minimizing total delivery time while respecting strict physical constraints. By incorporating advanced heuristics like Parallel Prep, Gated Clustering, and Fatigue Modeling, the system behaves not just as a pathfinder, but as an operationally aware logistics engine. The experimental results demonstrate that accounting for these real-world frictions leads to more executable and economically rational schedules compared to naive distance minimization.

12 Phase 3 Explanation

The Phase 3 implementation successfully addresses a **Travelling Salesman Problem (TSP) variant** faced by instant delivery companies (e.g., Zomato, Swiggy, Blinkit). The core challenge is to efficiently schedule m delivery orders across n delivery agents, all starting from a central depot at $t = 0$. The primary objective is to **minimize the sum of all delivery completion times** while adhering to temporal constraints and a strict 15-second timeout.

Problem Overview

- **Agents**: n agents starting from a central depot.
- **Orders**: m orders, each with distinct pickup and dropoff locations.
- **Objective**: $\min(\sum_{\text{all orders}} \text{Completion Time})$.
- **Constraints**: Pickup must precede delivery, maximum 10 orders per driver, 8-hour shift limit, and a 15-second scheduling timeout.

12.1 Core Components

- **GraphAdapter Class**: Manages graph operations and distance computations. Features include **intelligent caching** and multiple strate-

gies for optimized distance queries (exact shortest paths, approximate distances, and precomputed landmark-based lookups).

- **Driver Class**: Represents individual delivery agents. Handles route management, capacity tracking, and insertion feasibility evaluation. Enforces constraints like **maximum 10 orders** and the **8-hour shift limit**.
- **Scheduler Class**: Orchestrates the entire scheduling process using a sophisticated greedy insertion heuristic. Maintains a **priority queue** of drivers ordered by availability and load factor for efficient selection.
- **Order & RouteStop Structures**: Encapsulate order details (pick-up/dropoff nodes, deadlines, priorities, preparation times) and route waypoints with necessary timing information.

12.2 Algorithmic Approach: Greedy Cheapest Insertion

We implemented a Greedy Cheapest Insertion Heuristic. This algorithm constructs routes incrementally. For every unassigned order, the system evaluates every feasible position in every driver's current route and selects the insertion that minimizes a weighted cost function.

Why this approach?

- **Speed:** It avoids the exponential complexity of exhaustive search, fitting well within the 15-second time limit per query[cite: 77, 107].
- **Flexibility:** It easily accommodates complex, non-linear constraints (like driver fatigue and batching rules) which are difficult to model in linear programming.

12.2.1 Advanced Heuristics & Real-World Modeling

To move beyond a theoretical graph traversal and model a realistic delivery ecosystem, we identified and implemented six specific operational constraints. These heuristics are the core contribution of our solution[cite: 95].

12.2.2 Realistic Time Modeling (Prep & Dwell)

Standard algorithms often assume travel time is the only cost. We refined this by modeling "invisible" delays that affect real drivers:

12.2.3 Parallel Meal Preparation (Smart Batching)

- **Observation:** If a driver picks up two orders from the same restaurant, the waiting times overlap. They do not wait $Time_A + Time_B$.
- **Implementation:** We implemented logic where if consecutive stops are at the same node, the departure time is calculated as $\max(Arrival, ReadyTime_A, ReadyTi$ This significantly reduces idle time.

12.2.4 Batch Compatibility

- **Constraint:** To ensure food quality, we reject batching two orders if their preparation times differ by more than 15 minutes. This prevents one customer's food from getting cold while waiting for another's.

12.2.5 Gated Community Clustering

- **Observation:** Security checks at large tech parks or gated societies take significant time (e.g., 5 minutes).
- **Implementation:** We apply a "Gated Entry Penalty" (300s) only once per visit. If a driver delivers multiple orders within the same gated cluster, the penalty is waived for subsequent stops. This incentivizes the scheduler to group neighbor deliveries to a single driver.

12.2.6 Node-Specific Dwell Times

- **Implementation:** We added granular "last-mile" friction based on location type: +180s for Malls (parking difficulty) and +60s for Apartments (elevator delays).

Economic & Human Factors

We extended the objective function to optimize for business value and workforce sustainability:

12.2.7 Price-Based Priority

- **Mechanism:** We introduced a weighted cost function:

$$Cost = \Delta Time - (Price \times Weight)$$

- **Impact:** High-value orders effectively reduce the "cost" of a route, causing the greedy algorithm to prioritize them even if they require slightly longer travel. This maximizes revenue alongside efficiency.

12.2.8 Driver Fatigue Management

- **Mechanism:** We applied a quadratic penalty to the travel cost based on the driver's current shift duration.
- **Impact:** A driver nearing their 8-hour limit becomes "expensive" to the scheduler. This automatically routes long-haul trips to fresher drivers, ensuring load balancing and safety compliance without hard limits.

12.3 Performance Optimizations

Multiple strategies are incorporated to meet the 15-second timeout requirement:

- **Distance Caching**: A thread-safe unordered map caches computed shortest path distances, avoiding **redundant calculations**.
- **Landmark Preprocessing**: Distances between "important nodes" (depot, pickup/dropoff locations) are precomputed when $K \leq 200$, enabling $\mathcal{O}(1)$ lookup for these pairs.
- **Approximate Distance Evaluation**: Configuration allows the use of **approximate shortest path algorithms** for insertion cost estimation, trading minor accuracy for significant speed.
- **Limited Driver Search**: The algorithm examines at most **50 candidates** per order and can terminate early with a cost increase below 60 seconds.
- **Load Balancing**: The priority queue orders drivers by both **availability time** and **load factor** to promote balanced distribution.

13 Chat logs of AI

https://chatgpt.com/share/6921b1f4-71a4-8004-a044-65e61c04417c

https://chatgpt.com/share/6921b37e-98a8-8004-9a20-bf2a03cb4e3a