Graph-Based Routing System Simulation

Directory Structure of Code

```
24b1098_24b1038_24b1091/
 Phase-1/
    algorithms.cpp
    algorithms.hpp
    graph.cpp
    graph.hpp
    SampleDriver.cpp
    nlohmann/
        json.hpp
 Phase-2/
    algorithms.cpp
    algorithms.hpp
    approx.cpp
    approx.hpp
    graph.cpp
    graph.hpp
    kshortest.cpp
    kshortest.hpp
    SampleDriver.cpp
    nlohmann/
        json.hpp
 Phase-3/
    delivery.cpp
    delivery.hpp
    graph.cpp
    graph.hpp
    precompute.cpp
    main.cpp
    nlohmann/
        json.hpp
 testcases/
     graph_generator.py
     Phase1_query_generator.py
     Phase2_query_generator.py
     Phase3_query_generator.py
 Makefile
 Report.pdf
```

Testcases folder is used to generate the .json files of the graph, queries required for testing phase 1, 2 and 3

Makefile and Targets

The following Makefile is used to compile and run all three phases of the project.

Makefile Listing

```
CXX = g++
CXXFLAGS = -std=c++17 -O3 -Wall

.PHONY: all generate_json run clean

# Folders
PH1 = Phase-1
PH2 = Phase-2
PH3 = Phase-3

# Executables to be created in parent folder
all: phase1 phase2 phase3 precompute generate_json

phase1: $(PH1)/*.cpp
 $(CXX) $(CXXFLAGS) $(PH1)/*.cpp  -o phase1

phase2: $(PH2)/*.cpp
 $(CXX) $(CXXFLAGS) $(PH2)/*.cpp  -o phase2

phase3: $(PH3)/main.cpp $(PH3)/graph.cpp $(PH3)/delivery.cpp
 $(CXX) $(CXXFLAGS) $(PH3)/main.cpp $(PH3)/graph.cpp $(PH3)/delivery.cpp -o phase3

precompute: $(PH3)/precompute.cpp $(PH3)/graph.cpp
 $(CXX) $(CXXFLAGS) $(PH3)/precompute.cpp $(PH3)/graph.cpp -o precompute

generate_json:
 python3 testcases/graph_generator.py
 python3 testcases/Phase1_query_generator.py
 python3 testcases/Phase2_query_generator.py
 python3 testcases/Phase3_query_generator.py

run: generate_json precompute phase3 phase2 phase1
 ./precompute graph.json queries_phase3.json precomputed.bin
 ./phase3 graph.json queries_phase3.json output3.json
 ./phase2 graph.json queries_phase2.json output2.json
 ./phase1 graph.json queries_phase1.json output1.json

clean:
 rm -f phase1 phase2 phase3  precompute *.o *.json precomputed.bin
```

Explanation of Targets

- **all** Default target. Builds executables for all three phases and .json files that we used for testing:
  - `phase1`
  - `phase2`
  - `phase3`
  - `precompute`

  - `generate_json`

  Useful for full compilation after cloning the repository.

- **phase1** Compiles all files inside the `Phase-1/` directory into the executable `phase1`. This is the required target for Phase 1 evaluation:

  ```
  make phase1
  ./phase1 graph.json queries.json output.json
  ```

- **phase2** Similar to Phase 1 but uses the `Phase-2/` folder. This is included for completeness since future phases depend on it.

- **phase3** Builds the final Phase 3 solver using:
  - `main.cpp`
  - `graph.cpp`
  - `delivery.cpp`

  Produces the executable `phase3`.

- **precompute** Compiles the preprocessing program used for Phase 3. This generates a file `precomputed.bin`, which is later used by Phase 3.

- **run** A convenience target:
  1. Runs `precompute`
  2. Runs `phase3` with required files
  3. Runs `phase2` with required files
  4. Runs `phase1` with required files

  Usage:

  ```
  make run
  ```

- **clean** Removes all executables and intermediate files:

  ```
  rm -f phase1 phase2 phase3 precompute *.o *.json precomputed.bin
  ```

  Useful when recompilation from scratch is needed.

- **Should always run precompute before phase3**

How to Use the Makefile

Phase1

For Phase 1 evaluation, only the following commands are required:

```
make phase1
./phase1 graph.json queries.json output.json
```

This compiles and executes the Phase 1 shortest-path and KNN implementation.

Phase2

For Phase 2 evaluation, only the following commands are required:

```
make phase2
./phase2 graph.json queries.json output.json
```

This compiles and executes the Phase 2

Phase3

For Phase 3 evaluation

```
make precompute
./precompute graph.json queries.json precomputed.bin
make phase1
./phase1 graph.json queries.json output.json
```

This compiles and executes the Phase 3

For generating the graph and queries

We can simply run

```
make generate_json
```

to produce

```
"graph.json","queries_phase1.json","queries_phase2.json","queries_phase3.json"
```

1 PHASE-1

1.1 Assumptions

- Node IDs are integers in $[0, N-1]$.
- Missing edge length defaults to 0.0 (project spec requirement).
- Missing average-time defaults to 0.0.
- Speed profile may be empty; in that case time = average time.
- Speed profile is a 96-element array (15-minute slots across 24 hours).
- KNN queries assume at least one node has the required POI.
- Euclidean approximation uses flat-plane distance instead of haversine (Phase 1 requirement).

1.2 Python Scripts and Libraries

The Phase-1 testcase generator is:

```
Phase1_query_generator.py
```

Libraries Used

- `json`
- `random`

This produces `queries_phase1.json`.

1.3 Test Cases and Analysis

Graph Input (`graph.json`):

```
{
  "meta": {
    "id": "test_graph_1",
    "nodes": 4,
    "description": "Minimal graph for Phase 1 testing"
  },
  "nodes": [
    { "id": 0, "lat": 0.0, "lon": 0.0, "pois": ["restaurant"] },
    { "id": 1, "lat": 0.0, "lon": 1.0, "pois": ["hospital"] },
    { "id": 2, "lat": 1.0, "lon": 0.0, "pois": ["restaurant"] },
    { "id": 3, "lat": 1.0, "lon": 1.0, "pois": ["pharmacy"] }
  ],
  "edges": [
    { "id": 10, "u": 0, "v": 1, "length": 5.0, "average_time": 4.0,
      "speed_profile": [], "oneway": false, "road_type": "primary" },
    { "id": 11, "u": 1, "v": 3, "length": 6.0, "average_time": 4.0,
      "speed_profile": [], "oneway": false, "road_type": "local" },
    { "id": 12, "u": 0, "v": 2, "length": 3.0, "average_time": 2.0,
      "speed_profile": [], "oneway": false, "road_type": "secondary" },
    { "id": 13, "u": 2, "v": 3, "length": 4.0, "average_time": 2.0,
      "speed_profile": [], "oneway": false, "road_type": "secondary" }
  ]
}
```

Query Input (`queries.json`):

```
{
  "meta": { "id": "test_queries_1" },
  "events": [
    {
      "type": "shortest_path",
      "id": 101,
      "source": 0,
      "target": 3,
      "mode": "distance"
    }
  ]
}
```

Expected Output:

```
{
  "meta": { "id": "test_queries_1" },
  "results": [
    {
      "id": 101,
      "possible": true,
      "minimum_distance": 7.0,
      "path": [0, 2, 3]
    }
  ]
}
```

Explanation of the Expected Output

For the shortest path query from node `0` to node `3` (distance mode), there are two possible simple paths:

- Path $0 \to 1 \to 3$
  Total distance: $5 + 6 = 11$
- Path $0 \to 2 \to 3$
  Total distance: $3 + 4 = 7$

Although both paths are valid, the second path has a smaller total distance. Therefore, the optimal (minimum distance) path returned by the program should be:

$$0 \to 2 \to 3$$

with the corresponding minimum distance value:

$$\text{minimum\_distance} = 7.0$$

In code this can be obtained by dry running dijkstra

1.4 Time and Space Complexity

Dijkstra's Algorithm

$$T = O((V + E)\log V)$$

Space:

$$S = O(V + E)$$

Speed-Profile Cost Computation

For each edge traversal:

$$T = O(k) \quad (k = 96 \text{ slots worst-case})$$

KNN Euclidean

$$T = O(V), \quad S = O(1)$$

KNN Shortest Path

Runs Dijkstra once:

$$T = O((V + E) \log V)$$

2 Approach for Each Query Type

2.1 1. Shortest Path Queries

Shortest path queries use Dijkstra's algorithm. The behaviour differs only in how the edge weight is computed:

- **Distance mode**: The weight of each edge is its length. Standard Dijkstra is applied.
  Standard Dijkstra:
  $$\text{weight}(u, v) = \text{edge.length}$$
- **Time mode**: The weight is travel time. If a speed profile exists, time is computed slot-by-slot using the 15-minute speed profile. Otherwise, `average_time` is used.
  Modified Dijkstra:
  $$\text{weight}(u, v) = \begin{cases} \text{average\_time}, & \text{if no speed profile} \\ \text{computed time using speed-profile}, & \text{otherwise} \end{cases}$$
- **Constraints**: Forbidden nodes are skipped, and forbidden road types are ignored during edge exploration.
- **Output**: Whether a path exists, the minimum distance/time, and the node path.

2.2 2. KNN Queries

KNN queries return the k nearest nodes containing a required POI.

- **Euclidean metric**: For every node with the POI, compute straight-line distance to the query point and return the closest k.
  Computing Euclidean distance:
  $$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
- **Shortest-path metric**:

1. Finds nearest graph node to query point
2. Runs Dijkstra from that node
3. Among POI nodes, picks smallest k distances

2.3 3. Edge Modification Queries

- **remove_edge**: Deletes the edge from adjacency lists and stores a backup.
- **modify_edge**: Removes the old edge, applies JSON patches (length, average time, road type, oneway), and reinserts the updated version.

3 PHASE-2

3.1 Assumptions

- The graph is static.
- Queries are processed independently and sequentially.

3.1.1 `k_shortest_paths`

- The graph is assumed to have non-negative edge weights.
- If fewer than k distinct paths exist between the source and target, the algorithm returns only the available paths.
- If no valid path exists between the source and target, an empty list is returned.
- If multiple distinct paths have the same total distance, they may appear in any order in the output. No explicit tie-breaking rule is applied.

3.1.2 `k_shortest_paths_heuristic`

- The first path is always assumed to be the true shortest path, and all subsequent paths are considered alternatives.

3.1.3 `approx_shortest_path`

- If the time budget is exceeded, the remaining queries are skipped.
- Only valid and completed approximations are returned.

3.2 Test Cases and Analysis

- Test cases were created using small and medium-sized graphs to verify correctness and performance.
- The `k_shortest_paths` and `k_shortest_paths_heuristic` queries were tested for different values of k.

- The `approx_shortest_path` query was tested with varying time budgets and acceptable error percentages.
- Results were verified by checking path validity, distance values, and consistency across multiple runs.

3.3 Python Scripts and Libraries

The Phase-2 testcase generator is:

`Phase2_query_generator.py`

Libraries Used

- `json`
- `random`

3.4 Time and Space Complexity

Let V = number of nodes, E = number of edges, k = number of paths requested, and Q = number of source–target pairs in the approximate query.

- Time complexity for `k_shortest_paths` is $O(k \cdot E \log V)$ and space complexity is $O(V + E + kV)$.
- Time complexity for `k_shortest_paths_heuristic` is $O(k \cdot E \log V)$ and space complexity is $O(V + E + kV)$.
- For `approx_shortest_path`, since not all nodes are explored, the worst-case time complexity is $O(E \log V)$ and the space complexity is $O(V + Q)$.

3.5 Approach for Each Query Type

3.5.1 `k_shortest_paths` (Exact)

- To compute the k shortest simple paths, Yen's Algorithm was used.
- The algorithm first computes the shortest path between the given source and target using Dijkstra's algorithm.
- Then, one edge at a time from the shortest path is removed (called a spur deviation).
- From each removed point, a new shortest path is computed again using Dijkstra's algorithm.
- All newly generated paths are stored, and the next shortest among them is selected.
- This process continues until k unique shortest paths are generated or no more paths exist.

3.5.2 `k_shortest_paths_heuristic`

- First, the true shortest path is computed using Dijkstra's algorithm.
- A penalty is given to edges that have already been used in previous paths.
- Dijkstra's algorithm is re-run on the modified graph to get an alternate route.
- The overlap percentage between paths is measured.
- If the overlap is above the threshold, the path is rejected.

3.5.3 `approx_shortest_path`

- A modified version of A* is used, called A*-Box.
- Normal A* explores many branches of the graph.
- A*-Box restricts the search to a smaller region around the straight-line path from source to target.
- This region is defined using the Euclidean distance between nodes, the error percentage, and the given time budget.
- It ignores nodes that fall outside a certain "box" around the target path.
- The algorithm stops if the given time limit is exceeded.
- This approach greatly reduces runtime at the cost of a small accuracy loss, which is allowed by the problem.

4 PHASE-3

4.1 Assumptions

- All delivery orders contain a valid pickup and dropoff node present in the graph.
- Precomputed distances are available before scheduling begins.
- It is assumed that for every order, a valid path exists between the pickup and dropoff nodes in the graph (i.e., all required pairs are reachable).
- The graph is static during scheduling.
- A driver may pick up several orders before performing dropoffs
- Depot is the starting point for all drivers Every route begins at the depot node.
- Graph nodes contain valid latitude/longitude. Precomputation of polar coordinates assumes all nodes have correct geographic attributes.

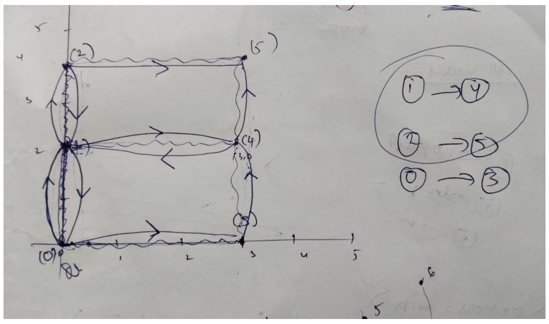
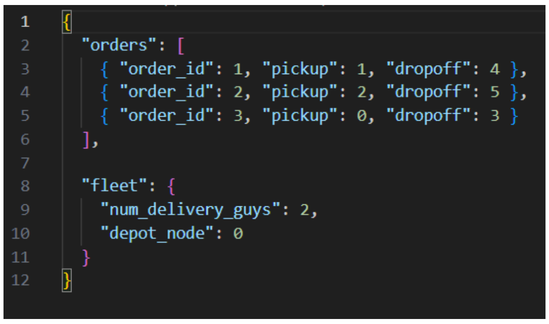


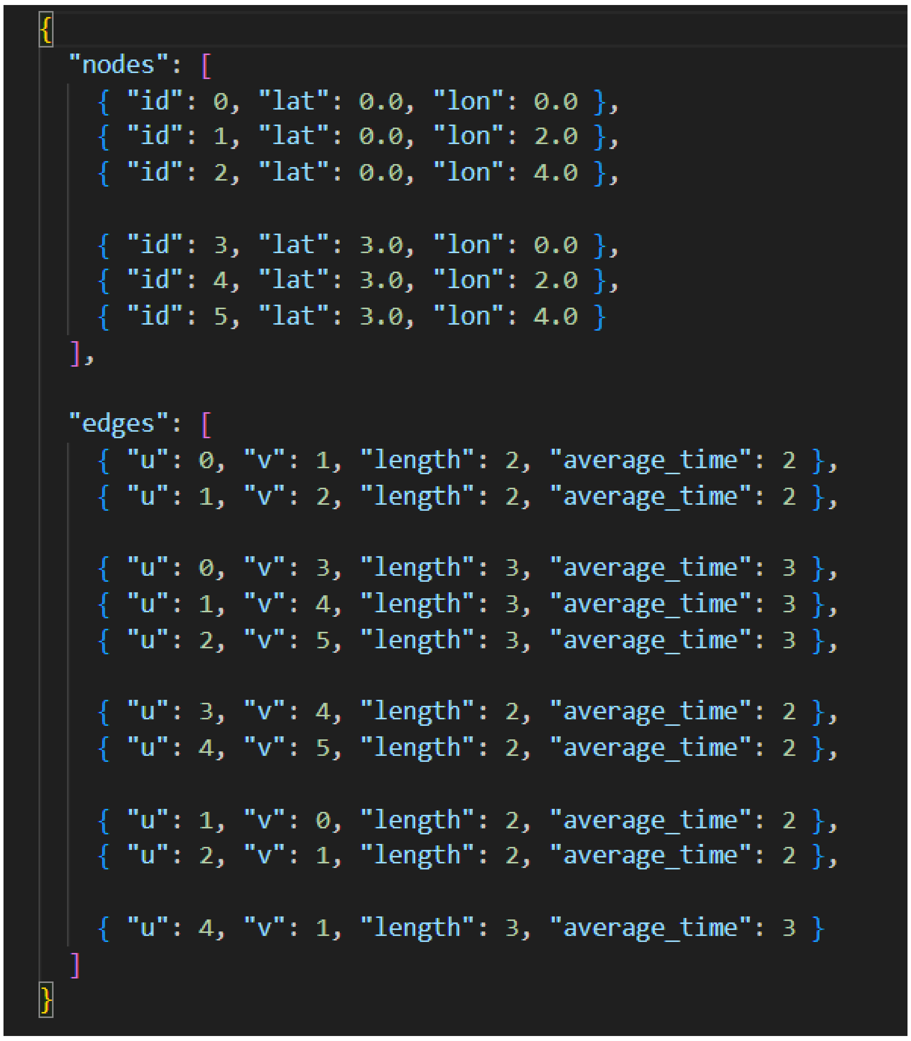
Figure 1: dry run we did



```
{
  "orders": [
    { "order_id": 1, "pickup": 1, "dropoff": 4 },
    { "order_id": 2, "pickup": 2, "dropoff": 5 },
    { "order_id": 3, "pickup": 0, "dropoff": 3 }
  ],

  "fleet": {
    "num_delivery_guys": 2,
    "depot_node": 0
  }
}
```

Figure 2: sample queries we took



```
{
  "nodes": [
    { "id": 0, "lat": 0.0, "lon": 0.0 },
    { "id": 1, "lat": 0.0, "lon": 2.0 },
    { "id": 2, "lat": 0.0, "lon": 4.0 },

    { "id": 3, "lat": 3.0, "lon": 0.0 },
    { "id": 4, "lat": 3.0, "lon": 2.0 },
    { "id": 5, "lat": 3.0, "lon": 4.0 }
  ],

  "edges": [
    { "u": 0, "v": 1, "length": 2, "average_time": 2 },
    { "u": 1, "v": 2, "length": 2, "average_time": 2 },

    { "u": 0, "v": 3, "length": 3, "average_time": 3 },
    { "u": 1, "v": 4, "length": 3, "average_time": 3 },
    { "u": 2, "v": 5, "length": 3, "average_time": 3 },

    { "u": 3, "v": 4, "length": 2, "average_time": 2 },
    { "u": 4, "v": 5, "length": 2, "average_time": 2 },

    { "u": 1, "v": 0, "length": 2, "average_time": 2 },
    { "u": 2, "v": 1, "length": 2, "average_time": 2 },

    { "u": 4, "v": 1, "length": 3, "average_time": 3 }
  ]
}
```

Figure 3: example graph we used

4.2 Testcases and Analysis

4.2.1 1. Single Order, Single Driver

- Expected behaviour: One simple route (depot $\rightarrow$ pickup $\rightarrow$ dropoff).
- Observed result: Algorithm produces exactly this route. Greedy and 2-opt have no effect since the route is minimal.

4.2.2 2. Multiple Orders, Single Driver

- Expected: Algorithm should visit all pickups first if they are closer, then their respective dropoffs without violating feasibility.
- Observed: Greedy routing chooses the nearest available pickup or dropoff at each step. 2-opt improves overall route length while ensuring pickup-before-dropoff constraints remain valid.

4.2.3 3. Multiple Drivers, Balanced Orders

- Expected: Orders should be distributed evenly among drivers based on spatial clustering.
- Observed: Angular–radial clustering assigns geographically close orders to the same driver, balancing workload. Each driver receives a compact region of orders, resulting in short routes.

4.2.4 4. Extreme Case: All Orders Near the Same Location

- Expected: Clustering should place multiple orders into a small number of drivers, with others receiving empty routes.
- Observed: Exactly this behaviour occurs. Idle drivers receive only the depot in their route.

4.2.5 5. Extreme Case: Sparse, Far-Apart Orders

- Expected: Each driver receives orders from different sectors of the graph.
- Observed: Sector-based clustering works well. Greedy routing and 2-opt both reduce travel time despite large distances.

4.2.6 6. Correctness Under Constraint: Pickup Before Dropoff

- Verified using crafted routes that violate pickup–dropoff ordering.
- Algorithm correctly rejects invalid routes in both greedy construction and 2-opt improvement.

4.2.7 7. Performance Test (Large Input)

- Precomputation dominates runtime; scheduling itself is fast.
- **This clustering works very well for graphs with high no.of nodes and pickup points located as colonies**
- For hundreds of orders, clustering + greedy + 2-opt completes within a few hundred milliseconds.

4.2.8 Python Scripts and Libraries

libraries:

- json
- random

The Phase-3 testcase generator is:

`Phase3_query_generator.py`

4.3 Time and Space Complexity

Let V be the number of nodes, E the number of edges, O the number of orders, and D the number of drivers.

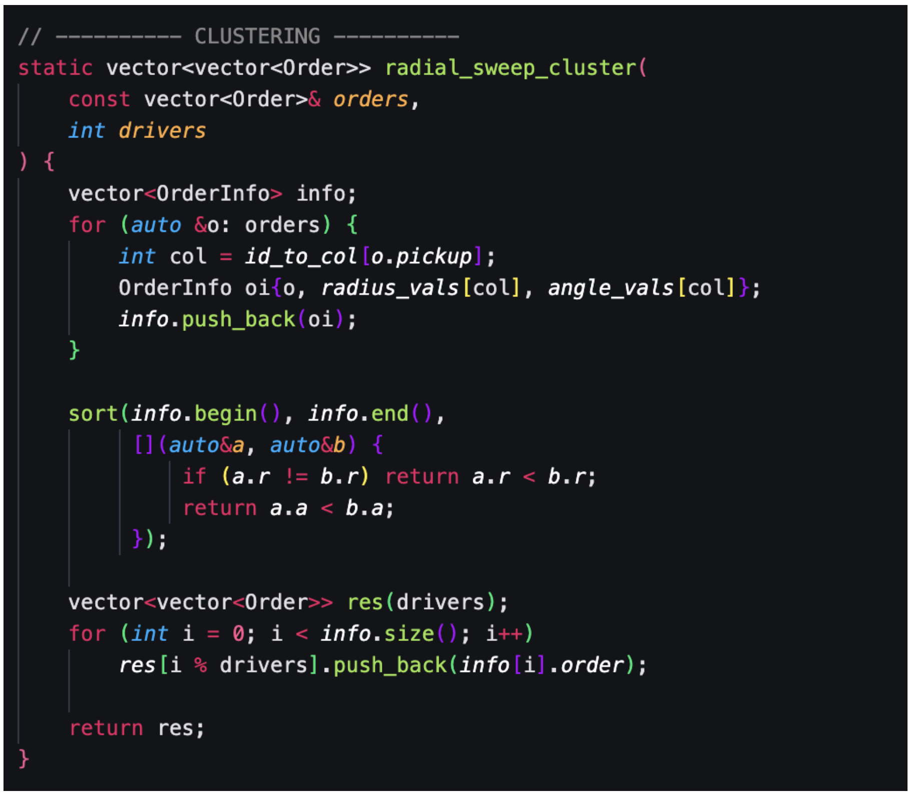
- Precomputation using Dijkstra from each important node: $O(O \cdot (E \log V))$
- Clustering the orders: $O(O \log O)$
- Greedy route construction per driver: $O(O^2)$ in the worst case
- 2-opt route improvement per driver (bounded iterations): Approximately $O(O^2)$
- Overall space complexity: $O(V + E + O \cdot V)$ due to storage of the precomputed distance table

4.4 Approach for Delivery Scheduling

4.4.1 Exploration of Strategies

Clustering Strategies:

- We first considered using k-means clustering to divide orders among drivers. However, it did not always form meaningful geographical clusters, since it is sensitive to initialisation and does not treat the depot as a special reference point.
- We then explored a sweep (angle-based) approach, where orders are grouped according to the angle they make with respect to the depot. This seemed unfair to some drivers as it distributes based on the slope and for some cases like orders being present at (1,1) and (100,100) it is worst distribution



```cpp
// ---------- CLUSTERING ----------
static vector<vector<Order>> radial_sweep_cluster(
    const vector<Order>& orders,
    int drivers
) {
    vector<OrderInfo> info;
    for (auto &o: orders) {
        int col = id_to_col[o.pickup];
        OrderInfo oi{o, radius_vals[col], angle_vals[col]};
        info.push_back(oi);
    }

    sort(info.begin(), info.end(),
         [](auto&a, auto&b) {
             if (a.r != b.r) return a.r < b.r;
             return a.a < b.a;
         });

    vector<vector<Order>> res(drivers);
    for (int i = 0; i < info.size(); i++)
        res[i % drivers].push_back(info[i].order);

    return res;
}
```

Figure 4: One of the earlier clustering methods that we have tried

- We also explored radical based clustering but it seemed unfair for drivers who have to deliver parcel present on nearly diametrically opposite nodes
- Finally, we chose a hybrid radial + sweep approach. In this method, both the angle and the distance from the depot are considered. This helped in creating more balanced and compact clusters and reduced overlap between the regions assigned to different drivers.

Routing Strategies:

- For route construction, we first used a greedy nearest-neighbour approach, where the driver always chooses the closest valid next stop (either a pickup or a drop-off). This method is simple, fast, and works well in practice.
- However, greedy routing alone is not always optimal, as it can lead to locally optimal but globally inefficient paths.
- To improve the quality of the route, a 2-opt optimisation step was added. This attempts to reduce the total route length by reversing certain segments whenever an improvement is found, while still respecting the pickup-before-drop-off constraint.
- We experimented with both the Sweep and Radial (sectoring) algorithms for initial routing, but a Greedy construction followed by 2-opt refinement consistently produced better route quality and lower total distance.

4.4.2 1. Single Order, Single Driver

  – Expected behaviour: One simple route (depot → pickup → dropoff).

    - Observed result: Algorithm produces exactly this route. Greedy and 2-opt have no effect since the route is minimal.

Final Strategy Used:

- Hybrid radial + sweep clustering for assigning orders to drivers
- Greedy nearest-neighbour routing to construct the initial route
- 2-opt optimisation to improve the final route

4.4.3 Precomputation

- All-pairs shortest travel times are not computed for the entire graph.
- Instead, Dijkstra's algorithm is run only from **important nodes**:
    - Depot
    - All pickup nodes
    - All dropoff nodes
- The resulting distances are stored in a binary file for fast lookup during scheduling.
- Since up to 1–2GB of memory was allowed for precomputation, we utilised this to store shortest path distances and significantly reduce the runtime of the scheduling phase.

4.4.4 Order Clustering

- Orders are grouped based on the polar coordinates (radius and angle) of their pickup locations with respect to the depot.
- The space around the depot is divided into angular sectors and radial zones.
- Orders falling in the same sector and zone are grouped into a single cluster.
- Each cluster is assigned to one driver to reduce routing overlap.

4.4.5 Route Construction

- For each driver, a greedy nearest-neighbour approach is used.
- At each step, the closest valid next node (pickup or dropoff) is selected.

4.4.6 Route Optimization

- After building the initial route, a 2-opt improvement is applied.
- Segments of the route are reversed to reduce total travel time.
- Only valid routes (pickup before dropoff) are accepted.

4.4.7 Total Delivery Time Calculation

- Each driver's route is simulated in order.
- When a dropoff node is reached, the elapsed travel time for that order is added to the total delivery time.
- The final metric is the sum over all completed deliveries.

5 CHAT LOGS

- https://chatgpt.com/share/69206022-5894-8008-9c09-d4c47f739478
- https://chatgpt.com/share/69209e86-a770-8008-8be4-41fade3de81d
- https://claude.ai/share/c4ca010e-0914-4610-a659-941fa8f58532
- https://chatgpt.com/share/6920cfc4-7c18-8002-b3a1-536ef56694f1
- https://chatgpt.com/share/6921e7f6-ae2c-8013-912d-6c5c633d20f4
- https://claude.ai/share/b7800e33-4522-4a18-bcee-ae5a28612554