Hanoi University of Science and Technology
School of Information and Communications Technology

PEOPLE AND PARCEL SHARE A RIDE LARGE SIZE

Tran Nhat Uy 202416763

2024.2 Term

ACKNOWLEDGEMENT

I would like to express our appreciation to Dr. Pham Quang Dung who gave us opportunity to work on this project. This work could not be finished without your helpful advices and thoroughly guidance.

ABSTRACT

This report discusses the problem of solving a **People and Parcel Share a Ride Large Size**, also known as **Vehicle Routing Problem with Pickup and Delivery (VRPPD)** involving mixed requests, multiple vehicles(taxis) with different capacities, and strict passenger rules. Next, we illustrate our different approaches to the problem, such as graph-based, mathematical and heuristics. Then, we describe generalizations of the problem and evaluate the efficiency of mentioned methods. Finally, we present our conclusions and discuss some open alternatives.

All the fundamental elements of this project, including solutions and evaluations are coded in Python and publicly available on this GitHub Repository

`https://github.com/Ulysses1lc/mini_project11.git`

Contents

Chapter 1

DESCRIPTION OF THE PROBLEM

1.1 Problem description

K taxis (located at point 0) are scheduled to serve transport requests including N passenger requests 1, 2, . . . , N and M parcel requests 1, 2, . . ., M. Passenger request i (i = 1, . . ., N)) has pickup point i and drop-off point i + N + M, and parcel request i (i = 1, . . . , M) has pickup point i + N and drop-off point i + 2N + M. d(i,j) is the travel distance from point i to point j (i, j = 0, 1, . . ., 2N + 2M). Each passenger must be served by a direct trip without interruption (no stopping point between the pickup point and the drop-off point of the passenger in each route). Each taxi k has capacity Q[k] for serving parcel requests. The parcel request i (i = 1, 2, . . ., M) has quantity q[i].Compute the routes for taxis satifying above contraints such that the length of the longest route among K taxis is minimal (in order to balance between lengths of taxis).A route of a taxi k is represented by a sequence of points visited by that route: r[0], r[1], . . ., r[Lk] in which r[0] = r[Lk] = 0 (the depot)

1.2 Input and Output

According to HUSTack, there are a total of 11 test cases, all have the following format

Input:

- Line 1: contains N, M and K ($1 \leq N, M \leq 500$, $1 \leq K \leq 100$)
- Line 2: contains $q[1], q[2], ...q[M]$ ($1 \leq q[i] \leq 100$)
- Line 3: contains $Q[1], Q[2], ...Q[K]$ ($1 \leq Q[i] \leq 200$)
- Line i + 3: (i = 0, 1, . . ., 2N + 2M): contains the i-th row of the distance matrix

Output:

- Line 1: contains an integer K
- Line 2k (k = 1, 2, . . ., K): contains a positive integer Lk
- Line 2k + 1 (k = 1, 2, . . ., K): contains a sequence of Lk integers r[0], r[1], . . ., r[Lk]

Example:

Input

3 3 2

8 4 5

16 16

0 8 7 9 6 5 11 6 11 12 12 12 13

8 0 4 1 2 8 5 13 19 12 4 8 9

7 4 0 3 3 8 4 12 15 8 5 6 7

9 1 3 0 3 9 4 14 19 11 3 7 8

6 2 3 3 0 6 6 11 17 11 6 9 10

5 8 8 9 6 0 12 5 16 15 12 15 15

11 5 4 4 6 12 0 16 18 7 4 3 4

6 13 12 14 11 5 16 0 15 18 17 18 19

11 19 15 19 17 16 18 15 0 13 21 17 17

12 12 8 11 11 15 7 18 13 0 11 5 4

12 4 5 3 6 12 4 17 21 11 0 7 8

12 8 6 7 9 15 3 18 17 5 7 0 1

13 9 7 8 10 15 4 19 17 4 8 1 0

Output

2

6

0 5 1 7 11 0

10

0 4 6 10 3 9 12 2 8 0

1.3 Math modelling

1.3.1. Sets and Indices:

- K: Set of taxis, indexed by $k \in \{1, \ldots, K\}$.
- N: Set of passenger requests, indexed by $p \in \{1, \ldots, N\}$.
- M: Set of parcel requests, indexed by $m \in \{1, \ldots, M\}$.
- V: Set of all relevant points.
  - Depot: 0.
  - Passenger pickup points: $P_p = p$ for $p \in \{1, \ldots, N\}$.
  - Passenger drop-off points: $D_p = p + N + M$ for $p \in \{1, \ldots, N\}$.
  - Parcel pickup points: $PP_m = m + N$ for $m \in \{1, \ldots, M\}$.
  - Parcel drop-off points: $DP_m = m + 2N + M$ for $m \in \{1, \ldots, M\}$.

  Thus, $V = \{0, 1, \ldots, 2N + 2M\}$.
- Indices for points: $i, j \in V$.

1.3.2. Parameters:

- $d(i, j)$: Travel distance from point i to point j, for all $i, j \in V$.
- $q[m]$: Quantity (or weight/volume) of parcel request m, for $m \in \{1, \ldots, M\}$.
- $Q[k]$: Parcel capacity of taxi k, for $k \in \{1, \ldots, K\}$.

```python
data = sys.stdin.read().split()
    if not data:
        return

    it = iter(data)
    N = int(next(it))
    M = int(next(it))
    K = int(next(it))
    q = [int(next(it)) for _ in range(M)]
    Q = [int(next(it)) for _ in range(K)]

    n_points = 2 * N + 2 * M + 1
    d = []
    for i in range(n_points):
        row = []
        for j in range(n_points):
            row.append(int(next(it)))
        d.append(row)

```

Standard Input Function

1.3.3. Decision Variables:

- $x_{ijk} \in \{0, 1\}$: Binary variable, equal to 1 if taxi k travels directly from point i to point j, and 0 otherwise.
- $y_{pk} \in \{0, 1\}$: Binary variable, equal to 1 if passenger request p is served by taxi k, and 0 otherwise.
- $z_{mk} \in \{0, 1\}$: Binary variable, equal to 1 if parcel request m is served by taxi k, and 0 otherwise.
- u_{ik}: Non-negative integer variable representing the sequential order of visiting point i by taxi k. Used for sub-tour elimination and precedence constraints.
- $L_{\max}$: Non-negative continuous variable representing the length of the longest route among all taxis.

1.3.4. Objective Function:

Minimize the maximum route length among all taxis:

$$\text{Minimize } L_{\max}$$

1.3.5. Constraints:

a) Request Assignment Constraints:

Each passenger request must be served by exactly one taxi:

$$\sum_{k=1}^{K} y_{pk} = 1 \quad \forall p \in \{1, \ldots, N\}$$

Each parcel request must be served by exactly one taxi:

$$\sum_{k=1}^{K} z_{mk} = 1 \quad \forall m \in \{1, \ldots, M\}$$

b) Flow Conservation and Route Structure Constraints:

Each taxi must start and end at the depot (point 0):

$$\sum_{j \in V, j \neq 0} x_{0jk} = 1 \quad \forall k \in \{1, \ldots, K\}$$

$$\sum_{i \in V, i \neq 0} x_{i0k} = 1 \quad \forall k \in \{1, \ldots, K\}$$

For any point j (other than the depot), if a taxi enters it, it must also leave it:

$$\sum_{i \in V, i \neq j} x_{ijk} = \sum_{l \in V, l \neq j} x_{jlk} \quad \forall j \in V \setminus \{0\}, \forall k \in \{1, \ldots, K\}$$

c) Linkage between Requests and Routes:

If passenger p is served by taxi k, then taxi k must visit passenger p's pickup point P_p and drop-off point D_p:

$$\sum_{j \in V, j \neq P_p} x_{P_p jk} = y_{pk} \quad \forall p \in \{1, \dots, N\}, \forall k \in \{1, \dots, K\}$$

$$\sum_{i \in V, i \neq D_p} x_{iD_p k} = y_{pk} \quad \forall p \in \{1, \dots, N\}, \forall k \in \{1, \dots, K\}$$

If parcel m is served by taxi k, then taxi k must visit parcel m's pickup point PP_m and drop-off point DP_m:

$$\sum_{j \in V, j \neq PP_m} x_{PP_m jk} = z_{mk} \quad \forall m \in \{1, \dots, M\}, \forall k \in \{1, \dots, K\}$$

$$\sum_{i \in V, i \neq DP_m} x_{iDP_m k} = z_{mk} \quad \forall m \in \{1, \dots, M\}, \forall k \in \{1, \dots, K\}$$

d) Passenger Direct Trip Constraint:

For any passenger p served by taxi k, the trip from their pickup point P_p to their drop-off point D_p must be direct, without any intermediate stops:

$$x_{P_p D_p k} = y_{pk} \quad \forall p \in \{1, \dots, N\}, \forall k \in \{1, \dots, K\}$$

e) Parcel Capacity Constraint:

The total quantity of parcels served by any taxi k must not exceed its capacity $Q[k]$:

$$\sum_{m=1}^{M} q[m] \cdot z_{mk} \leq Q[k] \quad \forall k \in \{1, \dots, K\}$$

f) Sub-tour Elimination Constraints :

To ensure that each taxi's route forms a single, continuous tour starting and ending at the depot, and to prevent disconnected cycles: For any two points $i, j \in V \setminus \{0\}$ and any taxi k:

$$u_{ik} - u_{jk} + (|V| - 1) \cdot x_{ijk} \leq |V| - 2 \quad \forall i, j \in V \setminus \{0\}, i \neq j, \forall k \in \{1, \dots, K\}$$

The u_{ik} variables must be within a valid range for sequencing:

$$1 \leq u_{ik} \leq |V| - 1 \quad \forall i \in V \setminus \{0\}, \forall k \in \{1, \dots, K\}$$

g) Parcel Pickup Before Drop-off Precedence Constraint:

For any parcel m served by taxi k, its pickup point PP_m must be visited before its drop-off point DP_m in the taxi's route. This is enforced using the sequence variables u_{ik}:

$$u_{PP_m k} \leq u_{DP_m k} - 1 + (|V| - 1) \cdot (1 - z_{mk}) \quad \forall m \in \{1, \dots, M\}, \forall k \in \{1, \dots, K\}$$

The term $(|V| - 1) \cdot (1 - z_{mk})$ acts as a large number (M_{big}) that makes the constraint non-binding if parcel m is not served by taxi k (i.e., $z_{mk} = 0$). If $z_{mk} = 1$, it forces $u_{PP_m k} < u_{DP_m k}$.

h) Longest Route Definition Constraint:

The total length of each taxi's route must be less than or equal to the maximum route length $L_{\max}$:

$$\sum_{i \in V} \sum_{j \in V} d(i,j) \cdot x_{ijk} \leq L_{\max} \quad \forall k \in \{1, \ldots, K\}$$

This complete mathematical model defines the problem as an Integer Linear Program (ILP). Given the large potential values for N and M (up to 500), directly solving this ILP for large instances might be computationally very intensive. For practical applications, exact methods like branch-and-cut or heuristic/metaheuristic approaches (e.g., genetic algorithms, simulated annealing, tabu search) are often employed.

Chapter 2

VRPPD ALGORITHMS

2.1 Dijkstra algorithm combined with Brute Force Assignment

Initial Setup and Pre-computation

- **Read Input:** Parses N, M, K, parcel quantities q, taxi capacities Q, and the distance matrix d.
- **Request List (`requests`):** Creates a list of all requests.
  - Passenger: `("p", passenger_id)`
  - Parcel: `("c", parcel_id)`
- **Problem Size Check (Heuristic Cutoff):**
  - If the total number of requests $(N + M)$ is greater than 10 OR the number of taxis (K) is greater than 5, the problem is considered too large for the exhaustive search.
  - In this case, a default trivial solution is printed (each taxi route is just 0 0 - depot to depot) and the script exits. This is because the number of possible assignments $(K^{(N+M)})$ grows extremely rapidly.

`taxi_router(S, capacity, N, M, q, d)` Function

```python

def taxi_router(S, capacity, N, M, q, d):
    if len(S) == 0:
        return 0, [0, 0]
    reqs = sorted(S)
    R = len(reqs)
    info = []
    for req in reqs:
        if req[0] == "p":
            i = req[1]
            pickup = i
            drop = i + N + M
            qty = 0
        else:
            i = req[1]
            pickup = i + N
            drop = i + 2 * N + M
            qty = q[i]
        info.append((req[0], i, pickup, drop, qty))
```

`taxi_router` Function (Line 45 - 183)

This function is the core of finding the optimal route for a single taxi given a specific set of requests S assigned to it and its capacity. It uses a Dijkstra-like algorithm on a state graph.

- **Arguments:**
  - S: A set of requests (e.g., $\{$`("p", 1), ("c", 2)`$\}$) assigned to this taxi.
  - `capacity`: The carrying capacity of this taxi.
  - N, M, q, d: Global problem parameters.
- **Return Value:** (`cost`, `path`) where `cost` is the minimum cost to reach all requests in S, and `path` is the sequence of points in the optimal route. Returns (10^{18}, `None`) if no valid route is found.
- **State Definition (**`loc`**,** `load`**,** `active`**,** `status`**):**
  - `loc`: Current physical location (point index) of the taxi.
  - `load`: Current load of the taxi (sum of quantities of picked-up parcels).
  - `active`: Index (in the sorted `info` list for this taxi's requests) of a passenger who has been picked up but not yet dropped off. -1 if no passenger is currently "active" in this way. This implies a passenger, once picked up, must be dropped off before any other action related to other requests.
  - `status`: A bitmask representing the completion status of each request in S. For each request j (0 to $R-1$ where R is `len(S)`), the status bits are:
    1. **00 (0):** Not yet picked up.
    2. **01 (1):** Picked up, not yet dropped off.
    3. **10 (2):** Dropped off (completed).
- **Algorithm:**
  - **Initialization:**
    * `info`: A sorted list of tuples (`type, id, pickup_loc, drop_loc, qty`) for requests in S.
    * `dp`: A dictionary mapping `state` to `min_cost` to reach that state.
    * `parent`: A dictionary mapping `state` to `previous_state` to reconstruct the path.
    * `queue`: A list used as a simple (non-priority) queue for Dijkstra's. Stores (`cost, state`).
    * Initial state: `(0, 0, -1, 0)` (at depot, empty, no active passenger, all requests not started). `dp[start_state] = 0`.
  - **Dijkstra-like Loop:** Repeatedly extract the state with the minimum cost from `queue`. (Note: This is done by iterating through the queue, not using a `heapq`, which is less efficient for large queues but might be acceptable for the small subproblems this function handles). If all requests are completed (`status` indicates all are 'dropped off') and no passenger is active, and the taxi is at a location from which it can return to the depot (point 0), this is a potential final state. The cost includes returning to the depot.
  - **Transitions:**
    * If a passenger is active: The only valid next move is to go to that passenger's `drop_loc`. Update `loc, cost, status` (mark passenger as dropped off), and set `active = -1`.
    * If no passenger is active: Iterate through all requests i in S:
      - If request i is a passenger ("p") and not yet picked up (`st_i == 0`): Transition to `pickup_loc` for passenger i. Update `loc, cost, status` (mark as picked up), set `active = i`.
      - If request i is a parcel ("c"):

st_i == 0 **(Not yet picked up):** If `load + qty_val <= capacity`, transition to `pickup_loc`. Update `loc, cost, load`, `status` (mark as picked up).

st_i == 1 **(Picked up but not dropped):** Transition to `drop_loc`. Update `loc, cost, load` (decrease), `status` (mark as dropped off).

* For each valid transition to a `new_state` with `new_cost`: if `new_cost` is better than `dp.get(new_state)`, update `dp` and `parent`, and add to queue.

- **Path Reconstruction:** If a `final_state` is found, backtrack using `parent` to get the sequence of locations. The path starts and ends at the depot (0).

Main Assignment Loop (Exhaustive Search)

```
for assign_index in range(total_assignments):
        assignment_vector = []
        x = assign_index
        for i in range(total_requests):
            r = x % K
            x = x // K
            assignment_vector.append(r)

        sets = [set() for _ in range(K)]
        for idx, req in enumerate(requests):
            taxi_id = assignment_vector[idx]
            sets[taxi_id].add(req)

        valid_assignment = True
        routes_per_taxi = [None] * K
        costs = [0] * K
        for k in range(K):
            S = sets[k]
            cap = Q[k + 1]
            cost_val, route_val = taxi_router(S, cap, N, M, q, d)
            if cost_val >= 10**18:
                valid_assignment = False
                break
            costs[k] = cost_val
            routes_per_taxi[k] = route_val

        if not valid_assignment:
            continue

        max_cost = max(costs)
        if max_cost < best_max_route_cost:
            best_max_route_cost = max_cost
            best_routes_per_taxi = routes_per_taxi

    if best_routes_per_taxi is None:
        print(K)
        for k in range(K):
            print(2)
            print("0 0")
    else:
        print(K)
        for k in range(K):

```

Main Assignment Loop

- `total_requests` $= N + M$.
- `total_assignments` $= K^{\texttt{total\_requests}}$. This is the number of ways to assign each of the `total_requests` to one of the K taxis.
- Initialize `best_max_route_cost` = infinity and `best_routes_per_taxi` = None.
- Iterate `assign_index` from 0 to `total_assignments - 1`:
  - **Decode** `assign_index`**:** Convert `assign_index` into an `assignment_vector`. This vector indicates which taxi each request is assigned to. (e.g., `assignment_vector[j] = k` means request j is assigned to taxi k).

- **Create Request Sets:** For each taxi, create a set `sets[k]` containing the requests assigned to it based on `assignment_vector`.
- **Calculate Routes and Costs for this Assignment:**
  - Initialize `current_assignment_max_cost = 0`.
  - For each taxi k from 0 to $K-1$:
    - Call `taxi_router(sets[k], Q[k+1], N, M, q, d)` to get the `cost_val` and `route_val` for taxi k serving its assigned requests.
    - If `cost_val` is infinity (meaning no valid route for this taxi's requests), this entire assignment is invalid (`valid_assignment = False`), so break and try the next `assign_index`.
    - Store `costs[k] = cost_val` and `routes_per_taxi[k] = route_val`.
  - **Update Best Solution:** If the current assignment is valid:
    - `max_cost_for_this_assignment = max(costs)`.
    - If `max_cost_for_this_assignment < best_max_route_cost`:
    - Update `best_max_route_cost = max_cost_for_this_assignment`.
    - Update `best_routes_per_taxi` with the current `routes_per_taxi`.

Outputting the Result

- If `best_routes_per_taxi` is still `None` after checking all assignments (meaning no valid assignment was found that could serve all requests), print the default trivial solution.
- Otherwise, print K, and then for each taxi, print the length of its route and the route itself from `best_routes_per_taxi`.

Key Features and Approach

- **Exact Solution for Small Instances:** Attempts to find a globally optimal solution (minimizing the maximum route cost) by exploring all possibilities.
- **Exhaustive Assignment:** Iterates through every possible way to assign requests to taxis.
- **Optimal Single-Taxi Routing:** Uses a state-space search (Dijkstra-like on states) within `taxi_router` to find the shortest path for a single taxi given its assigned requests and capacity.
- **Bitmasking for State Representation:** Efficiently represents the status of multiple requests within a single integer.
- **Problem Size Cutoff:** Recognizes the computational infeasibility for larger problems and defaults to a simple output.
- **Capacity Constraints:** Strictly enforces taxi capacity for parcels.
- **Passenger Handling Constraint:** The `active` state in `taxi_router` implies a passenger, once picked up, must be dropped off before any other action related to other requests can be picked up or dropped off by that taxi.

Dependencies

- `sys`: Standard Python library for `sys.stdin`.

Limitations and Potential Improvements

- **Scalability:** The primary limitation is scalability. $K^{(N+M)}$ grows extremely fast, making the exhaustive assignment search feasible only for very small N, M, K. The hardcoded cutoff ($N + M > 10$ or $K > 5$) reflects this.
- `taxi_router` **Efficiency:** While `taxi_router` finds an optimal route for its subproblem, its manual queue management (iterating to find min) is less efficient than using a priority queue (e.g., `heapq`) for the Dijkstra-like search, especially if the number of states becomes large (though limited by `len(S)`).
- **Passenger** `active` **Constraint:** The strict handling of an "active" passenger (must be dropped off next) might be overly restrictive for some real-world scenarios where a taxi might pick up another request en route to a passenger's dropoff if it makes sense.
- **Memory Usage:** The `dp` and `parent` dictionaries in `taxi_router` can grow large depending on the number of states, which is related to $2^{(2\cdot\text{len(S)})} \times$ num_locations $\times$ num_load_levels.
- **No Heuristics for Larger Cases:** Beyond the cutoff, it doesn't fall back to a heuristic approach; it just gives a trivial answer. A more robust solution might integrate heuristic methods for larger instances.

2.2 Local search - Hill Climbing combined with Greedy Scheduler

Initial Data Preparation

- **Read Input:** Parses N, M, K, parcel quantities q, taxi capacities Q, and the distance matrix d.
- **Passenger and Parcel Lists:**

  Passengers: A list of tuples `('passenger', passenger_id, pickup_loc, dropoff_loc)`.

  Parcels: A list of tuples `('parcel', parcel_id, quantity, pickup_loc, delivery_loc)`.
- **Initial Random Assignment:**

  `pass_assign`: A list where `pass_assign[i]` stores the randomly chosen taxi index (0 to $K-1$) for passenger i.

  `parc_assign`: A list where `parc_assign[j]` stores the randomly chosen taxi index for parcel j.

`greedy_scheduler(events, capacity, dist_mat)` Function

This function constructs a route for a single taxi given a list of events (pickup/delivery tasks) assigned to it, its capacity, and the `dist_mat`.

```python
def greedy_scheduler(events, capacity, dist_mat):
        if not events:
            return 0, [0, 0]

        unscheduled_set = set()
        for ev in events:
            unscheduled_set.add(ev)

        current_point = 0
        current_load = 0
        picked_parcels = set()
        total_distance = 0
        points = [0]

        available = []
        for ev in unscheduled_set:
            if ev[0] == "passenger":
                available.append(ev)
            elif ev[0] == "pickup":
                if current_load + ev[3] <= capacity:
                    available.append(ev)
            elif ev[0] == "delivery":
                if ev[1] in picked_parcels:
                    available.append(ev)

```

Greedy Scheduler (Line 42 - 115)

- **Event Tuples:**

  Passenger: `('passenger', passenger_id, pickup_loc, dropoff_loc)`

  Parcel Pickup: `('pickup', parcel_id, pickup_loc, quantity)`

  Parcel Delivery: `('delivery', parcel_id, delivery_loc, -quantity)` (quantity is negative for load calculation consistency)
- **Algorithm:**
  - **Initialization:**

    * `unscheduled_set`: A set of all events to be processed.
    * `current_point` = 0 (depot), `current_load` = 0, `total_distance` = 0.
    * `points` = [0] (route starts at depot).
    * `picked_parcels`: A set to track IDs of parcels currently on board.
  - **Build available events list:** Initially, and after each event is scheduled:
    * All unscheduled `'passenger'` events are available.
    * `'pickup'` events are available if `current_load + event_quantity <= capacity`.
    * `'delivery'` events are available if the corresponding `parcel_id` is in `picked_parcels`.
  - **Main Loop (while available is not empty):**
    * **Select Next Event:** Iterate through all available events. Choose the `next_ev` that is closest to the `current_point` (minimum `dist_mat[current_point][event_location]`). If no `next_ev` can be found (e.g., due to capacity or precedence constraints not allowing any available event), break.
    * **Process `next_ev`:**
      · Remove `next_ev` from `unscheduled_set`.
      · **If `'passenger'`:** Add distance: `current_point` → `pickup_loc` → `dropoff_loc`. Add `pickup_loc` and `dropoff_loc` to `points`. Update `current_point` to `dropoff_loc`.
      · **If `'pickup'`:** Add distance: `current_point` → `pickup_loc`. Add `pickup_loc` to `points`. Update `current_point` to `pickup_loc`. Update `current_load += quantity`. Add `parcel_id` to `picked_parcels`.
      · **If `'delivery'`:** Add distance: `current_point` → `delivery_loc`. Add `delivery_loc` to `points`. Update `current_point` to `delivery_loc`. Update `current_load += quantity` (which is negative for deliveries, effectively reducing load). Remove `parcel_id` from `picked_parcels`.

    Rebuild the available events list based on the new `current_load`, `unscheduled_set`, and `picked_parcels`.
  - **Return to Depot:** Add distance from `current_point` to 0 (depot). Append 0 to `points`.
  - **Return** `total_distance` and the constructed `points` (route).

- **Note on greedy_scheduler's completeness:** If an event cannot be scheduled due to capacity or precedence at some point, it remains in `unscheduled_set`. The loop continues with other available events. This means the `greedy_scheduler` might not schedule all events passed to it if constraints cannot be met with its greedy choices. The main loop doesn't explicitly check for unscheduled events from `greedy_scheduler`.

`compute_routes(pass_assign, parc_assign)` Function

This function takes the current assignment of passengers and parcels to taxis and calculates the routes and costs for all taxis.

```python
def compute_routes(pass_assign, parc_assign):
        events_per_taxi = [[] for _ in range(K)]
        for i in range(N):
            taxi = pass_assign[i]
            ev = passengers[i]
            events_per_taxi[taxi].append(("passenger", ev[1], ev[2], ev[3]))
        for j in range(M):
            taxi = parc_assign[j]
            pcl = parcels[j]
            events_per_taxi[taxi].append(("pickup", pcl[1], pcl[3], pcl[2]))
            events_per_taxi[taxi].append(("delivery", pcl[1], pcl[4], -pcl[2]))

        total_dists = [0] * K
        routes = [None] * K
        for taxi in range(K):
            dist_val, points_route = greedy_scheduler(events_per_taxi[taxi], Q[taxi], d)
            total_dists[taxi] = dist_val
            routes[taxi] = points_route
        return total_dists, routes, events_per_taxi

    total_dists, routes, events_per_taxi = compute_routes(pass_assign, parc_assign)
    current_objective = max(total_dists) if total_dists else 0
    current_routes = routes
    current_pass_assign = pass_assign
    current_parc_assign = parc_assign
    current_events_per_taxi = events_per_taxi
```

`compute_routes(pass_assign, parc_assign)` Function

- **Organize Events:** Creates `events_per_taxi`, a list of lists. For each taxi, it populates its list with event tuples derived from passengers and parcels based on `pass_assign` and `parc_assign`. Parcel events are split into distinct `'pickup'` and `'delivery'` events.
- **Calculate Routes:** For each taxi:
  - Call `greedy_scheduler` with its assigned events, capacity `Q[taxi]`, and distance matrix `d`.
  - Store the returned `dist_val` and `points_route`.
- **Return** `total_dists` (list of costs per taxi), `routes` (list of routes per taxi), and `events_per_taxi`.

Iterative Improvement (Local Search)

```python
n = 100
    for i in range(n):
        move_type = "reassign_task"
        if random.random() < 0.5:
            task_idx = random.randint(0, N - 1)
            old_taxi = current_pass_assign[task_idx]
            new_taxi = random.randint(0, K - 1)
            while new_taxi == old_taxi and K > 1:
                new_taxi = random.randint(0, K - 1)
            new_pass_assign = current_pass_assign[:]
            new_parc_assign = current_parc_assign[:]
            new_pass_assign[task_idx] = new_taxi
        else:
            task_idx = random.randint(0, M - 1)
            old_taxi = current_parc_assign[task_idx]
            new_taxi = random.randint(0, K - 1)
            while new_taxi == old_taxi and K > 1:
                new_taxi = random.randint(0, K - 1)
            new_pass_assign = current_pass_assign[:]
            new_parc_assign = current_parc_assign[:]
            new_parc_assign[task_idx] = new_taxi

        new_total_dists, new_routes, new_events_per_taxi = compute_routes(
            new_pass_assign, new_parc_assign
        )
        new_objective = max(new_total_dists) if new_total_dists else 0

        if new_objective < current_objective:
            current_objective = new_objective
            current_pass_assign = new_pass_assign
            current_parc_assign = new_parc_assign
            current_events_per_taxi = new_events_per_taxi
            current_routes = new_routes
            current_total_dists = new_total_dists
```

Local Search

- **Initial Solution:** Call `compute_routes` with the initial random `pass_assign` and `parc_assign` to get an initial set of routes and their costs.
- **Objective:** `current_objective` = `max(total_dists)`.

- **Store Best:** Keep track of `current_routes`, `current_pass_assign`, `current_parc_assign`, etc.
- **Local Search Loop (for n = 100 iterations):**
  - **Generate Neighbor:** Randomly decide whether to modify a passenger assignment or a parcel assignment (50/50 chance).
    * Select a random `task_idx` (either a passenger or a parcel).
    * Get its `old_taxi` assignment.
    * Choose a `new_taxi` randomly, ensuring it's different from `old_taxi` if $K > 1$.
    * Create `new_pass_assign` and `new_parc_assign` by copying the current assignments and updating the assignment for the selected `task_idx` to `new_taxi`.
  - **Evaluate Neighbor:** Call `compute_routes` with `new_pass_assign` and `new_parc_assign` to get `new_total_dists` and `new_routes`.
  - **Calculate New Objective:** `new_objective` = `max(new_total_dists)`.
  - **Acceptance Criterion:** If `new_objective < current_objective`:
    * Update `current_objective` to `new_objective`.
    * Update `current_pass_assign`, `current_parc_assign`, `current_routes`, etc., to reflect the better solution.

  The `move_type` variable is initialized but not used to select different types of moves.

Outputting the Result

- After the local search loop completes, print K.
- Then, for each taxi, print the length of its route and the route itself from the `current_routes` (which holds the best solution found).

Key Features and Approach

- **Metaheuristic (Local Search):** Uses an iterative improvement strategy starting from a random solution.
- **Random Initial Assignment:** Begins with a randomly generated assignment of tasks to taxis.
- **Greedy Single-Taxi Router (`greedy_scheduler`):** Employs a nearest-neighbor-like greedy heuristic to construct a route for each taxi based on its assigned events, respecting capacity and basic precedence (parcel pickup before delivery).
- **Simple Move Operator:** The local search uses a simple "reassign task" move, where one randomly chosen task is moved to a different random taxi.
- **Descent Heuristic:** Only accepts moves that improve the objective function (minimize the maximum route cost).
- **Fixed Number of Iterations:** The local search runs for a predetermined number of iterations (n=100).

Dependencies

- `sys`: Standard Python library for `sys.stdin`.
- `random`: Standard Python library for generating random numbers (for initial assignment and local search moves).

Limitations and Potential Improvements

- **Local Optima:** Being a simple descent local search, the algorithm can easily get stuck in local optima. The quality of the final solution heavily depends on the starting random solution and the search landscape.
- `greedy_scheduler` **Heuristic:** The `greedy_scheduler` itself is a heuristic and does not guarantee optimal routes for the subproblems (i.e., for a given set of events for one taxi). Its greedy choices might lead to suboptimal overall path lengths or inability to schedule all assigned tasks if complex interactions occur.
- **Basic Move Operator:** The "reassign task" move is quite basic. More sophisticated neighborhood structures or move operators (e.g., swaps, 2-opt for intra-route improvement) could lead to better solutions.
- **Fixed Iterations/No Sophisticated Termination:** Runs for a fixed number of iterations. Could benefit from adaptive termination criteria (e.g., no improvement for X iterations).
- **No Guarantees on Feasibility from** `greedy_scheduler`**:** The `greedy_scheduler` might not be able to schedule all events assigned to a taxi if its greedy choices lead to constraint violations later. The main loop doesn't explicitly handle or penalize partially completed routes from `greedy_scheduler`.
- **Randomness:** The solution quality can vary significantly between runs due to the random initial assignment and random choices in the local search.

2.3 Metaheuristic - Simulated Annealing

Core Logic and Algorithm

Helper Functions

- `read_input()`: Parses input from `stdin` and returns N, M, K, q, Q, d. Includes basic error handling.
- `calculate_route_distance(route, d)`: Calculates the total distance of a given route using the distance matrix d.
- `is_valid_route(route, N, M, q, Q, taxi_id)`: Checks feasibility of a single taxi's route.
  - **Ensures route starts and ends at depot (0).**
  - **Strict Passenger Constraint:** Verifies that if a passenger pickup point ($1 \leq \text{node} \leq N$) is visited, the very next point in the route must be its corresponding dropoff point ($\text{node}+N+M$). This implies direct, uninterrupted trips for passengers once picked up.
  - **Parcel Capacity and Precedence:** Tracks load and carried parcels. Ensures load does not exceed $Q[\text{taxi\_id}]$ upon parcel pickup, and ensures a parcel is carried before it can be dropped off.
  - **Pickup/Dropoff Completeness:** (Partially covered by other checks) Attempts to ensure all picked-up items are eventually dropped. The current implementation of this specific check might be incomplete or redundant given other checks.
- `get_max_route_length(routes, d)`: Calculates the objective function value: the maximum distance among all taxi routes (routes are 1-indexed in the routes list).

`initial_solution(N, M, K, q, Q, d)`

Constructs an initial feasible solution

```python
def initial_solution(N, M, K, q, Q, d):
    taxis_route = [[] for _ in range(K + 1)]
    taxis_passengers = [[] for _ in range(K + 1)]

    for i in range(1, N + 1):
        k = (i - 1) % K + 1
        taxis_passengers[k].append(i)

    for k in range(1, K + 1):
        route = [0]
        unvisited = set(taxis_passengers[k])
        current = 0
        while unvisited:
            next_i = min(unvisited, key=lambda i: d[current][i])
            route.append(next_i)
            route.append(next_i + N + M)
            current = next_i + N + M
            unvisited.remove(next_i)
        route.append(0)
        taxis_route[k] = route
```

`initial_solution(N, M, K, q, Q, d)` function (Line 66 - 129)

- **Passenger Assignment & Routing:**
  - `taxis_passengers`: A global list (also initialized in main) where `taxis_passengers[k]` stores a list of passenger IDs assigned to taxi k. Passengers are assigned round-robin: $k = (i-1) \pmod K + 1$.
  - For each taxi k:
    * A route is built starting at the depot (0).
    * While taxi k has unvisited assigned passengers: Select the `next_i` (unvisited passenger ID) closest to the current location in the route. Append `next_i` (pickup) and then `next_i + N + M` (dropoff) to the route. Update current location to the dropoff point.
    * Append depot (0) to complete the route.
    * Store this in `taxis_route[k]`.
- **Parcel Assignment & Routing (Greedy Insertion):**
  - Sort parcels by quantity in descending order (largest first).
  - For each `parcel_idx`:
    * Identify its pickup (a) and dropoff (b) points.
    * Iterate through all taxis k (if capacity $Q[k]$ is sufficient).
    * Iterate through all possible insertion positions (i, j) in taxi k's current route `taxis_route[k]` to insert a before `route[i]` and b before `route[j]` (where $j \geq i$).
    * Skip insertions that would split a passenger's direct pickup-dropoff sequence.
    * Form `new_route` $=$ `route[:i]` $+ [a] +$ `route[i:j]` $+ [b] +$ `route[j:]`.
    * If `is_valid_route(new_route, ...)` and its cost is the best found so far for this parcel, store it.
    * **Fallback:** If a best valid insertion (`best_k, best_route`) is found, update `taxis_route[best_k]`. If no valid insertion is found, assign the parcel to the taxi k with the current shortest route (if capacity allows). Insert the parcel's pickup and dropoff immediately after the initial depot visit (`route[:1]` $+ [a, b] +$ `route[1:]`). Update if valid.
- **Return** `taxis_route` (1-indexed list of routes).

`neighbor_solution(routes, N, M, q, Q, d)`

Generates a neighboring solution by applying one of three random move types to a deep copy of the current routes:

```python
def neighbor_solution(routes, N, M, q, Q, d):
    routes = [r[:] for r in routes]
    K = len(routes) - 1
    move_type = random.choice(["swap_passenger", "swap_parcel", "reorder"])

    if move_type == "swap_passenger" and N > 0:
        k1, k2 = random.sample(range(1, K + 1), 2)
        if taxis_passengers[k1] and taxis_passengers[k2]:
            p1 = random.choice(taxis_passengers[k1])
            p2 = random.choice(taxis_passengers[k2])
            route1 = routes[k1]
            route2 = routes[k2]
            new_route1 = [x for x in route1 if x not in [p1, p1 + N + M]]
            new_route2 = [x for x in route2 if x not in [p2, p2 + N + M]]
            i1 = random.randint(1, len(new_route1))
            i2 = random.randint(1, len(new_route2))
            new_route1 = new_route1[:i1] + [p2, p2 + N + M] + new_route1[i1:]
            new_route2 = new_route2[:i2] + [p1, p1 + N + M] + new_route2[i2:]
            if is_valid_route(new_route1, N, M, q, Q, k1) and is_valid_route(
                new_route2, N, M, q, Q, k2
            ):
                routes[k1] = new_route1
                routes[k2] = new_route2
```

`neighbor_solution(routes, N, M, q, Q, d)` function (Line 135 - 208)

- **'swap_passenger':**
  - Randomly select two different taxis, $k1$ and $k2$.
  - If both have assigned passengers (relies on global `taxis_passengers`):
    - Randomly pick passenger $p1$ from `taxis_passengers[k1]` and $p2$ from `taxis_passengers[k2]`.
    - Attempt to swap them: remove $p1$'s points from `routes[k1]` and $p2$'s from `routes[k2]`. Then, re-insert $p2$'s points into `routes[k1]` at a random valid position and $p1$'s into `routes[k2]` similarly.
    - Update `routes` only if both new routes are valid.
  - **Note:** This move implicitly changes the underlying passenger-to-taxi assignment, which should ideally be reflected in `taxis_passengers` if it were to be consistently used by other parts of SA.
- **'swap_parcel':**
  - Randomly select two different taxis, $k1$ and $k2$.
  - Identify parcels currently in `routes[k1]` and `routes[k2]`.
  - If both have parcels:
    - Randomly pick parcel $p1$ from taxi $k1$'s route and $p2$ from $k2$'s route.
    - If capacities allow ($q[p1] \leq Q[k2]$ and $q[p2] \leq Q[k1]$):
      - Attempt to swap: remove $p1$'s points from `routes[k1]` and $p2$'s from `routes[k2]`. Then, re-insert $p2$'s pickup/dropoff into `routes[k1]` at random valid positions and $p1$'s into `routes[k2]` similarly.
      - Update `routes` only if both new routes are valid.
- **'reorder' (Intra-route 2-opt like move):**
  - Randomly select one taxi k.
  - If its route `routes[k]` is long enough (> 3 points):
    - Randomly select two distinct indices i, j (excluding start/end depot).
    - Ensure the segment `route[i:j+1]` does not contain a passenger pickup immediately followed by its dropoff (to avoid splitting it incorrectly by simple reversal).
    - Reverse the segment: `new_route` = `route[:i]` + `route[i:j+1][::-1]` + `route[j+1:]`.
    - If `new_route` is valid, update `routes[k]`.
- **Return** the modified `routes`.

`simulated_annealing(N, M, K, q, Q, d)`

The main SA optimization loop:

```python
def simulated_annealing(N, M, K, q, Q, d):
    routes = initial_solution(N, M, K, q, Q, d)
    current_cost = get_max_route_length(routes, d)
    best_routes = [r[:] for r in routes]
    best_cost = current_cost

    T = 1000.0  # Initial temperature
    T_min = 0.1
    alpha = 0.995  # Cooling rate
    max_iterations = 10000

    for _ in range(max_iterations):
        if T < T_min:
            break
        new_routes = neighbor_solution(routes, N, M, q, Q, d)
        new_cost = get_max_route_length(new_routes, d)
        delta = new_cost - current_cost
        if delta <= 0 or random.random() < math.exp(-delta / T):
            routes = new_routes
            current_cost = new_cost
            if current_cost < best_cost:
                best_routes = [r[:] for r in routes]
                best_cost = current_cost
        T *= alpha

    return best_routes
```

`simulated_annealing(N, M, K, q, Q, d)` Function

- **Initialization:**
  - `routes` = `initial_solution(...)`.
  - `current_cost` = `get_max_route_length(routes, d)`.
  - `best_routes` = `routes[:]`, `best_cost` = `current_cost`.
  - SA parameters: T (initial temperature), T_{min} (minimum temperature), α (cooling rate), `max_iterations`.
- **Iteration Loop:** For `max_iterations` or until $T < T_{\text{min}}$:
  - `new_routes` = `neighbor_solution(routes, ...)`.
  - `new_cost` = `get_max_route_length(new_routes, d)`.
  - `delta` = `new_cost - current_cost`.
  - **Acceptance Criterion:**
    * **If** `delta <= 0` (new solution is better or equal), accept: `routes = new_routes`, `current_cost = new_cost`.
    * **Else** (new solution is worse), accept with probability `math.exp(-delta / T)`.
  - If accepted and `current_cost < best_cost`, update `best_routes` and `best_cost`.
  - **Cool down:** $T* = \alpha$.
- **Return** `best_routes`.

`main()` Function

```python
def main():
    try:
        N, M, K, q, Q, d = read_input()
        global taxis_passengers
        taxis_passengers = [[] for _ in range(K + 1)]
        for i in range(1, N + 1):
            k = (i - 1) % K + 1
            taxis_passengers[k].append(i)

        routes = simulated_annealing(N, M, K, q, Q, d)

        print(K)
        for k in range(1, K + 1):
            route = routes[k]
            print(len(route))
            print(" ".join(map(str, route)))
    except Exception as e:
        print(f"Error: {e}", file=sys.stderr)
        raise
```

`main` Function

- Calls `read_input()`.
- Initializes the global `taxis_passengers` based on a round-robin assignment (this is somewhat redundant as `initial_solution` does a similar assignment but `neighbor_solution` relies on this global).
- Calls `simulated_annealing()` to get the final routes.
- Prints the results in the specified format.
- Includes a `try-except` block for overall error handling.

Key Features and Approach

- **Metaheuristic (Simulated Annealing):** Employs SA to escape local optima and find high-quality solutions.
- **Constructive Initial Solution:** Uses a mix of round-robin (passengers) and greedy insertion (parcels) to build a starting point.
- **Neighborhood Search:** Explores the solution space using `swap_passenger`, `swap_parcel`, and `reorder` move operators.
- **Probabilistic Acceptance:** Key feature of SA, allowing occasional acceptance of worse moves to avoid getting stuck.
- **Strict Passenger Routing:** Enforces that passengers are taken directly to their destination once picked up.
- **Capacity Validation:** Checks capacity constraints throughout route construction and modification.

Dependencies

- `sys`: For input/output.
- `math`: For `math.exp` in SA.
- `random`: For random choices in initial solution (implicitly, though not directly used there), neighbor generation, and SA probability.

Limitations and Potential Improvements

- **Global Variable `taxis_passengers`:** The use of a global variable `taxis_passengers` that is modified by `initial_solution` (implicitly through its logic) and read by `neighbor_solution` can make the code harder to follow and maintain. The `neighbor_solution` for passenger swaps should ideally update this assignment structure if it's meant to be the source of truth for passenger assignments.
- **Initial Solution Quality:** The quality of the initial solution can significantly impact SA performance. The current greedy methods are reasonable but could be more sophisticated.
- **Neighbor Operators:**
  - The `swap_passenger` and `swap_parcel` moves re-insert items at random valid positions. More intelligent re-insertion (e.g., best-insertion) could be more effective.
  - The `reorder` move is a simple segment reversal. More advanced intra-route heuristics (e.g., full 2-opt, Or-opt) could be beneficial.
- **SA Parameter Tuning:** SA parameters ($T, T_{\text{min}}, \alpha,$ max_iterations) are hardcoded and might need tuning for different problem sizes or characteristics.
- **Validity of `is_valid_route`'s Completeness Check:** The final loop in `is_valid_route` checking pickups vs dropoffs might be redundant if other parts correctly ensure all assigned tasks are in the route.
- **Parcel Insertion Fallback:** The fallback in `initial_solution` for parcels (inserting right after depot) is very basic and might create poor initial routes.
- **Efficiency of `neighbor_solution`:** Rebuilding and validating routes from scratch after each small change can be computationally intensive. More incremental updates could be faster.

2.4 Greedy combined with Beam Search Heuristics

Core Logic and Algorithm

a. Initialization

```python
requests = []
    for i in range(1, N + 1):
        requests.append(("passenger", i))
    for i in range(1, M + 1):
        requests.append(("parcel", i, q[i - 1]))

    sorted_requests = []
    for req in requests:
        if req[0] == "passenger":
            sorted_requests.append(req)
        else:
            sorted_requests.append(req)
    sorted_requests.sort(
        key=lambda x: (
            0 if x[0] == "passenger" else 1,
            -x[2] if x[0] == "parcel" else 0,
        )
    )

    routes = [[0, 0] for _ in range(K)]
    dists = [0] * K
    load_profiles = [[0, 0] for _ in range(K)]
    assigned_requests = [set() for _ in range(K)]
    beam_size = 5
```

Initialization (Line 6 - 60)

- **Data Ingestion:** The script begins by reading all input values (N, M, K, q, Q_{taxi}, and the elements for $\text{dist}_{\text{matrix}}$) from `sys.stdin` and parsing them into appropriate data structures.
- **Point Information (`point_info`):** A list named `point_info` is created to store metadata associated with each point index. This list has a length of `total_points`.
  - For passenger pickup/dropoff points: `('passenger', 'pickup'/'dropoff', passenger_id)`
  - For parcel pickup/dropoff points: `('parcel', 'pickup'/'dropoff', parcel_id, parcel_quantity)`
  - Point 0 (depot) remains `None` in this list.
- **Request Aggregation (`requests`):** All individual passenger and parcel requests are compiled into a single list called `requests`.
  - Passenger request format: `('passenger', passenger_id)`
  - Parcel request format: `('parcel', parcel_id, parcel_quantity)`
- **Request Sorting (`sorted_requests`):** The `requests` list is sorted to create `sorted_requests`. The sorting key prioritizes:
  - Passengers before parcels.
  - Among parcel requests, those with a larger quantity (`-x[2]`) are prioritized (descending order of quantity). This is a common heuristic to attempt to place larger, potentially more constrained items first.
- **Taxi State Variables:** Several lists are initialized to track the state of each of the K taxis:

  - `routes`: A list of lists. `routes[k]` will store the sequence of point indices visited by taxi k. Each route is initialized to `[0, 0]`, signifying a trip from the depot back to the depot.
  - `dists`: A list where `dists[k]` stores the total accumulated distance of the current route for taxi k. Initialized to 0 for all taxis.
  - `load_profiles`: A list of lists. `load_profiles[k]` is intended to store the load of taxi k after visiting each point in its route. Initialized with `[0,0]` for each taxi.
  - `assigned_requests`: A list of sets. `assigned_requests[k]` will store the unique IDs of the requests (passenger or parcel) assigned to taxi k.
  - `beam_size`: An integer, hardcoded to 5. This parameter is used in the parcel assignment heuristic to limit the search space for insertion points.

b. Request Assignment Loop

The core of the algorithm iterates through each request in the `sorted_requests` list. For every request, it attempts to find the "best" assignment to a taxi. The "best" assignment is defined as the one that, after inserting the new request's pickup and dropoff points into a taxi's route, minimizes the new maximum route length among all taxis.

```python
for req in sorted_requests:
        current_max_route = max(dists) if dists else 0
        best_new_max_route = float("inf")
        best_taxi = None
        best_new_route = None
        best_new_dist = None
        best_new_load = None
        best_new_request_id = None

        if req[0] == "passenger":
            request_id = req[1]
            pickup = request_id
            dropoff = request_id + N + M
            for k in range(K):
                current_route = routes[k]
                current_dist = dists[k]
                current_load = load_profiles[k]
                n = len(current_route)
                for i in range(n - 1):
                    a = current_route[i]
                    b = current_route[i + 1]
```

Request Assignment Loop (Line 77-279)

$$\text{Objective Function Heuristic: } \min(\max(\text{current\_max\_route\_among\_all\_taxis}, \text{new\_dist\_for\_modified\_taxi}))$$

b.1. Passenger Assignment

When the current `req` is a passenger request (e.g., `('passenger', request_id)`):

- The pickup point is `request_id`.
- The dropoff point is `request_id + N + M`.
- The algorithm iterates through each taxi k from 0 to $K-1$.
- For each taxi, it iterates through all possible insertion positions in its current `routes[k]`. An insertion position is defined by an existing edge (a, b) in the route (i.e., `current_route[i]` and `current_route[i+1]`).

- It evaluates the cost of inserting the pickup → dropoff sequence between a and b, forming a new segment $\ldots \to a \to$ pickup $\to$ dropoff $\to b \to \ldots$.
- The additional distance incurred by this insertion is: $d(a, \text{pickup}) + d(\text{pickup}, \text{dropoff}) + d(\text{dropoff}, b) - d(a, b)$.
- The `new_dist` for taxi k and the potential `new_max_route` (the maximum of `current_max_route` and `new_dist`) are calculated.
- The script keeps track of the assignment (which taxi k, the new route, new distance, new load profile, and request ID) that results in the smallest `best_new_max_route` found so far.
- Passenger requests are assumed not to contribute to the load in terms of taxi capacity (the `load_profiles` update for passengers simply duplicates the load at the insertion point, implying passenger load is negligible or handled differently).

b.2. Parcel Assignment

When the current `req` is a parcel request (e.g., `('parcel', request_id, quantity)`):

- The pickup point is `request_id + N`.
- The dropoff point is `request_id + 2*N + M`.
- The algorithm iterates through each taxi k.
- **Capacity Pre-check:** If the taxi's capacity $Q_{\text{taxi}}[k]$ is less than the parcel's quantity, this taxi cannot serve this request and is skipped.
- **Beam Search for Insertion:** Due to the complexity of inserting two separate points (pickup and dropoff) with load constraints, a beam search heuristic is employed:
  - **Candidate Pickup Insertions (`candidates_p`):** The script considers inserting the pickup point at all possible positions i in taxi k's current route. For each potential pickup insertion, it calculates the cost (`cost_p`), the new partial route (`new_route_p`), the new partial distance (`new_dist_p`), and the updated load profile (`new_load_p`). The load increases by `quantity` from the pickup point onwards in this partial route. These pickup insertion candidates are sorted by their `new_dist_p`. Only the top `beam_size` candidates are retained for further processing.
  - **Candidate Dropoff Insertions (`candidates_d`):** For each of the `beam_size` best pickup candidates: The script then considers inserting the dropoff point at all valid positions j after the already inserted pickup point in `new_route_p`. For each potential dropoff insertion, it calculates the additional cost (`cost_d`), the complete new route (`new_route`), the final new distance (`new_dist`), and the final new load profile (`new_load`). The load decreases by `quantity` from the dropoff point onwards. **Load Capacity Check:** A crucial step here is to verify that the `max_load` in the `new_load` profile does not exceed the taxi's capacity $Q_{\text{taxi}}[k]$. Only valid routes (respecting capacity) are considered. Valid combined pickup/dropoff candidates are sorted by their final `new_dist`. Again, only the top `beam_size` of these are retained.
- **Best Parcel Assignment Selection:** From all the (beam-limited) valid combined pickup/dropoff candidates generated across all taxis, the one that minimizes the overall `new_max_route` (the maximum of `current_max_route` and `new_dist` for the modified taxi) is selected as the best potential assignment for this parcel.

c. Fallback Mechanism

After evaluating all primary assignment options (including the beam search for parcels), if no valid assignment was found (i.e., `best_taxi` is still `None`), a fallback mechanism is triggered.

- The fallback logic iterates through all taxis and all possible insertion points again, similar to the primary assignment logic.
- For passengers, it's largely a repeat of the initial passenger assignment logic.
- For parcels, it attempts to insert pickup and then dropoff without the `beam_size` restriction on candidates during the intermediate pickup insertion phase. It still checks capacity for the final route.
- The goal of the fallback is to find any valid assignment, even if it's not optimal under the primary objective. It selects the first valid assignment found or one that improves upon a previously found fallback assignment in terms of `new_max_route`.
- If the fallback mechanism also fails to assign the request (`fallback_success` remains `False`), an error message "`Failed to assign request: [req_info]`" is printed to `stdout`, and the script terminates. This suggests the problem instance might be infeasible with the current heuristics or constraints.

d. Assignment Confirmation

If a best assignment (either from the primary search or the fallback) is found for the current request:

- The state variables for the chosen taxi (`k_assign`) are updated: `routes[k_assign]`, `dists[k_assign]`, `load_profiles[k_assign]`, and `assigned_requests[k_assign]` are updated with the new route, distance, load profile, and the ID of the just-assigned request.

e. `simulate_load` Function

The script defines an inner function `simulate_load(route, taxi_index, new_request_id)`.

```python
def simulate_load(route, taxi_index, new_request_id):
        load = 0
        new_load_profile = []
        assigned_set = assigned_requests[taxi_index] | {new_request_id}
        for point in route:
            info = point_info[point]
            if info is not None:
                if info[0] == "parcel" and info[2] in assigned_set:
                    if info[1] == "pickup":
                        load += info[3]
                    elif info[1] == "dropoff":
                        load -= info[3]
            new_load_profile.append(load)
        return new_load_profile
```

`simulated_load` Function

This function is designed to calculate a new load profile for a given route as if `new_request_id` were added to the `assigned_requests` set of `taxi_index`. It iterates through the points in the route, consults `point_info` for parcel pickup/dropoff details (only for requests in the assigned_set), and accumulates the load.

Key Features and Heuristics

- **Hybrid Request Handling:** Manages both passenger and parcel requests within a unified framework.
- **Capacity Constraints:** Enforces vehicle capacity limits for parcel transportation.
- **Min-Max Objective for Workload Balancing:** The core heuristic aims to minimize the length of the longest taxi route, which tends to distribute the workload more evenly.
- **Greedy Prioritized Insertion:** Requests are processed in a sorted order (passengers first, then larger parcels). Assignments are made greedily based on the min-max objective at each step.
- **Beam Search for Parcel Insertion:** Employs beam search to manage the combinatorial complexity of finding optimal paired pickup and dropoff locations for parcels, pruning the search space.
- **Iterative Route Construction:** Routes are built incrementally by inserting requests one by one.
- **Fallback Assignment Strategy:** Includes a secondary mechanism to try and assign requests if the primary, more restrictive heuristic fails.

Dependencies

- `sys`: A standard Python library, used here for reading from standard input (`sys.stdin`).

Limitations and Potential Improvements

- **Heuristic Nature:** The algorithm is a heuristic. It does not guarantee finding the globally optimal solution; it aims for a good, feasible solution quickly.
- **Static Input:** Assumes all data (requests, distances, capacities) are known upfront and are static. It does not adapt to dynamic changes (e.g., new requests arriving, traffic affecting travel times).
- **No Time Windows:** The current model does not explicitly support time window constraints for pickups or deliveries.
- **Fallback Simplicity:** The fallback logic is quite similar to the primary one. More diverse or relaxed fallback strategies could be explored for greater robustness.
- **Hardcoded `beam_size`:** The `beam_size` is fixed at 5. The optimal value for this parameter might vary depending on the problem instance's size and characteristics
- **Basic Load Profile Update for Passengers:** The way passenger insertions update `load_profiles` (duplicating the previous load) suggests that passenger count or individual passenger "weight" is not a factor in capacity, or it's handled implicitly.

Chapter 3

RESULTS

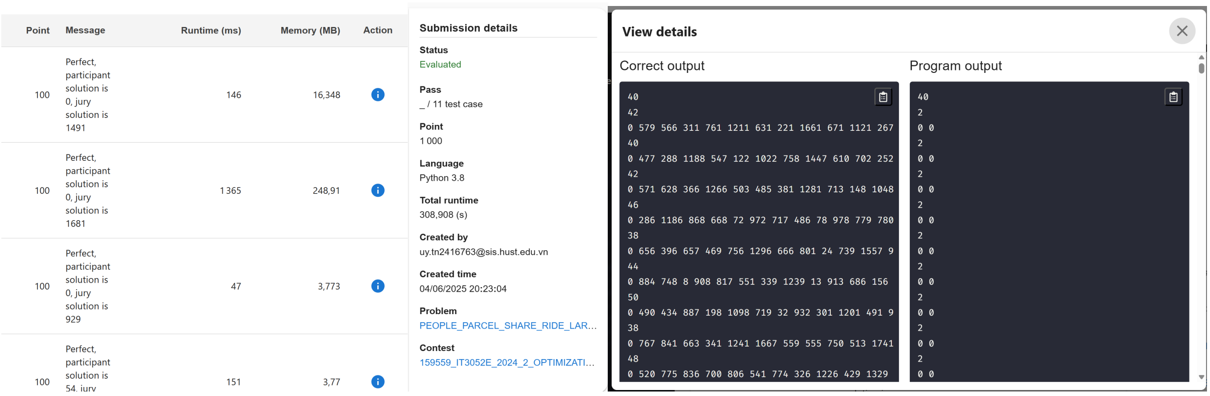
3.1 HUSTack

The result of the mini-project is collected from HUSTack.

ca236a	PEOPLE_PARCEL_SHARE_RIDE_LARGE_EXT1	Evaluated	1 000	_ / 11	Python 3.8	2025-06-04 20:23:04
355de7	PEOPLE_PARCEL_SHARE_RIDE_LARGE_EXT1	Evaluated	1 100	_ / 11	Python 3.8	2025-06-04 14:29:15
09fbcc	PEOPLE_PARCEL_SHARE_RIDE_LARGE_EXT1	Evaluated	1 100	_ / 11	Python 3.8	2025-06-04 14:16:32
f5341d	PEOPLE_PARCEL_SHARE_RIDE_LARGE_EXT1	Evaluated	1 015	_ / 11	Python 3.8	2025-06-04 14:12:29

Maximum score is 1100 for 11 test cases, each of them valued 100.

Overall, the scripts are statistically fair, with all of them managing to give scores. However, what we are looking to in this section is their true performance: the time-space complexities, and their accuracy.



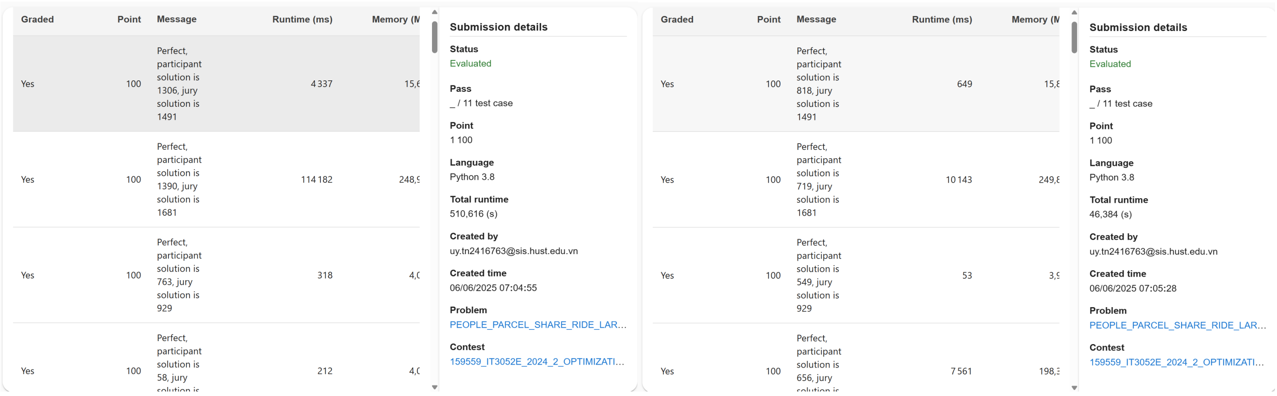
From left to right: Section 2.1 Script (Dijkstra) overall results and output

After looking further and analyzing the test results, we managed to differentiate the eventual attributes of each approach, and select the best option possible for this problem. For example, Dijkstra failed to give us any feasible solution, opting to return a line of 2 zeros when facing large test cases. Others, like Hill Climbing, though having given us acceptable results, have wavering stability.

ID	Problem	Status	Point	Pass	Language	Created time
f60e87	PEOPLE_PARCEL_SHARE_RIDE_LARGE_EXT1	Evaluated	1 002	_ / 11	Python 3.8	2025-06-05 22:31:20
f30cce	PEOPLE_PARCEL_SHARE_RIDE_LARGE_EXT1	Evaluated	1 011	_ / 11	Python 3.8	2025-06-05 22:28:45

The submits of the same Hill Climbing script at 2 different times give different scores. This is understandable due to the randomness of local search applied in this script.

Only two of the scripts give us maximum points. Greedy proves to be much faster, though Simulated Annealing (SA) gives us outputs much closer to the jury solution, indicating higher accuracy.



From left to right: Summary of results from SA and Greedy scripts respectively

This result proved to be far from our actual expectations, and it failed to address our main concerns, such as the validity of the outputs and the ability to indicate common trends of the test cases. We decided to put the solutions through more challenging tests.

3.2 Custom Generator and Testing

After analyzing the HUSTack test cases and the constraints, we found out that its evaluation system lacks strictness and discipline. It falsely validated the invalid output of the Dijkstra method, which returns a "depot - depot" route instead of a real solution. We also found out that the evaluation of HUSTack only evaluated the outputs based on their format, neglected many of the compulsory constraints from the problem, which is vital to find the true best solution of the whole problem.

On the other hand, test cases utilized by HUSTack lack variety and uniqueness, focus on a single format of N = M. They haven't included special test cases with higher discrepancies between the parameters.

Thus, though all the solutions are proven to be theoretically feasible on HUSTack, we decided to implement a separate evaluation pipeline, with the same constraints given by the problem, under the criteria of running time, running cost, and feasibility on larger and more unbalanced test cases (e.g.,N = 350, M = 300).

Simultaneously, we imposed heavy penalties on violations of the constraints. Any of the 4 solutions shall be considered to fail the test case if they encounter one of those violations.

This part of evaluation aims to find the perfect solution of the problem, without being too dependent on pure statistics and scores. Therefore, it is understandable that many solutions get eliminated, though they are void of errors and work well on HUSTack.
We use a custom test case generator `generator.py` and a custom testing evaluate function `tester.py` to create visualizations. The results were mindblowing.

```
[tester.py] Processing test case 7/10: test_300_250_30_7.csv
    [tester.py]   Running algorithm 1/4: dijkstra ... FAILED: Not all passengers served
    [tester.py]   Running algorithm 2/4: greedy ... FAILED: Passenger 100 not direct in taxi 1
    [tester.py]   Running algorithm 3/4: hill_climbing ... FAILED: Not all parcels served
    [tester.py]   Running algorithm 4/4: metaheuristic ... OK (cost=5774, max=232, t=27545.72ms)
[tester.py] Processing test case 8/10: test_350_300_35_8.csv
    [tester.py]   Running algorithm 1/4: dijkstra ... FAILED: Not all passengers served
    [tester.py]   Running algorithm 2/4: greedy ... FAILED: Passenger 1 not direct in taxi 1
    [tester.py]   Running algorithm 3/4: hill_climbing ... FAILED: Not all parcels served
    [tester.py]   Running algorithm 4/4: metaheuristic ... OK (cost=6404, max=188, t=42887.10ms)
[tester.py] Processing test case 9/10: test_400_350_40_9.csv
    [tester.py]   Running algorithm 1/4: dijkstra ... FAILED: Not all passengers served
    [tester.py]   Running algorithm 2/4: greedy ... FAILED: Passenger 5 not direct in taxi 1
    [tester.py]   Running algorithm 3/4: hill_climbing ... FAILED: Not all parcels served
    [tester.py]   Running algorithm 4/4: metaheuristic ... OK (cost=7478, max=193, t=48249.70ms)
```

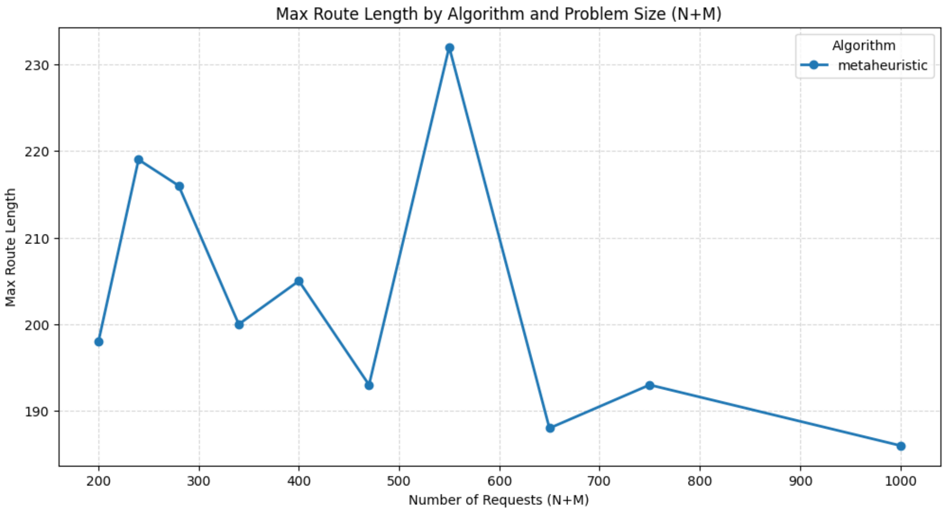
Only Metaheuristic (Simulated Annealing) made it after the testing. The rest failed all of the test cases due to different violations

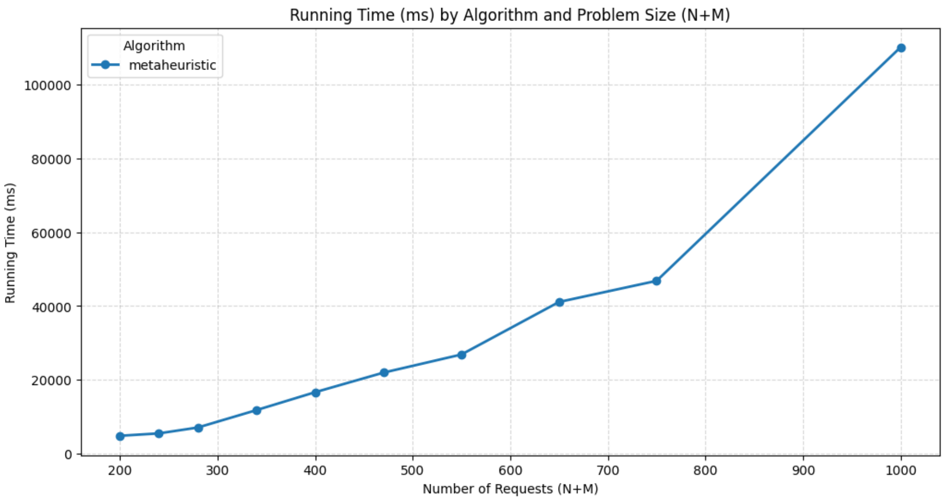
	N	M	K	dijkstra	greedy	hill_climbing	metaheuristic
test_100_100_10_1.csv	100	100	10	F: Not all passengers served	F: Passenger 91 not direct in taxi 1	F: Not all parcels served	cost=1942, max=198, t=4844.35ms
test_120_120_12_2.csv	120	120	12	F: Not all passengers served	F: Passenger 69 not direct in taxi 1	F: Not all parcels served	cost=2552, max=219, t=5495.02ms
test_150_130_15_3.csv	150	130	15	F: Not all passengers served	F: Passenger 112 not direct in taxi 1	F: Not all parcels served	cost=3149, max=216, t=7096.39ms
test_180_160_18_4.csv	180	160	18	F: Not all passengers served	F: Passenger 13 not direct in taxi 1	F: Not all parcels served	cost=3448, max=200, t=11807.56ms
test_200_200_20_5.csv	200	200	20	F: Not all passengers served	F: Passenger 129 not direct in taxi 1	F: Not all parcels served	cost=3987, max=205, t=16646.02ms
test_250_220_25_6.csv	250	220	25	F: Not all passengers served	F: Passenger 104 not direct in taxi 1	F: Not all parcels served	cost=4700, max=193, t=21949.80ms
test_300_250_30_7.csv	300	250	30	F: Not all passengers served	F: Passenger 100 not direct in taxi 1	F: Not all parcels served	cost=5774, max=232, t=26870.87ms
test_350_300_35_8.csv	350	300	35	F: Not all passengers served	F: Passenger 1 not direct in taxi 1	F: Not all parcels served	cost=6404, max=188, t=41106.36ms
test_400_350_40_9.csv	400	350	40	F: Not all passengers served	F: Passenger 5 not direct in taxi 1	F: Not all parcels served	cost=7478, max=193, t=46829.66ms
test_500_500_50_10.csv	500	500	50	F: Not all passengers served	F: Passenger 69 not direct in taxi 1	F: Not all parcels served	cost=9145, max=186, t=110118.33ms

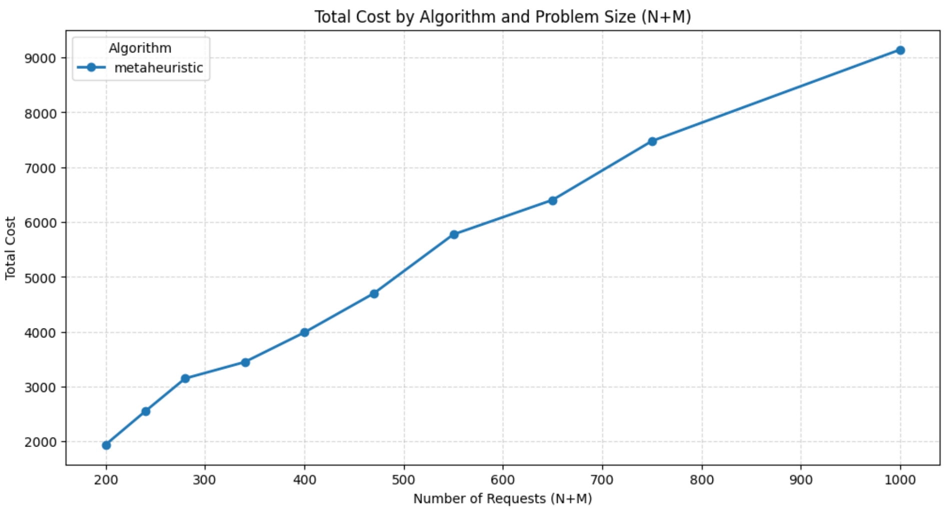
Summary of Testing Results

The **Metaheuristic (Simulated Annealing) approach** turned out to be dominant in every aspect of the test. **We concluded that it is the only suitable and feasible solution for this problem.**

Other visualizations of the performance of the solution







Chapter 4

CONCLUSION AND POSSIBLE EXTENSIONS

4.1. Overall Conclusion on the Analyzed VRPPD Solvers

The set of Python scripts analyzed demonstrates a diverse range of algorithmic strategies for tackling the Vehicle Routing Problem with Pickup and Delivery (VRPPD). Key takeaways include:

- **Diverse Strategies Employed:** The approaches span from exact methods (optimal for minuscule instances) and constructive heuristics (fast, basic solutions) to greedy iterative methods, simple local search, and more advanced metaheuristics like Simulated Annealing.
- **Core VRPPD Constraint Handling:**
  - **Paired Locations & Precedence:** All scripts inherently recognize that requests involve distinct pickup and delivery points, with pickup needing to occur before delivery. This is managed through route construction logic, state representations, or explicit checks.
  - **Capacity:** Vehicle capacity, especially for parcels, is a common constraint addressed at various stages—assignment, route building, or validation.
- **Primary Objective Focus:** Most scripts, implicitly or explicitly, aim to minimize the makespan (the longest route time/distance among all taxis). This objective naturally promotes a more balanced workload across the fleet.
- **Evident Trade-offs:**
  - **Optimality vs. Scalability:** Exact methods guarantee optimal solutions but are only feasible for very small problem sizes. Heuristic approaches sacrifice this guarantee for the ability to solve larger, practical instances in reasonable time.
  - **Solution Quality vs. Computational Speed:** More sophisticated metaheuristics (e.g., Simulated Annealing) generally yield higher-quality solutions but demand more computation time compared to simpler constructive or greedy algorithms.
- **Common Heuristic Patterns Observed:**
  - **Greedy Choices:** Many algorithms rely on making locally optimal decisions at various points (e.g., prioritizing larger parcels, selecting the nearest next stop, inserting requests where the immediate cost is lowest).
  - **Iterative Refinement:** Local search and Simulated Annealing highlight the effectiveness of starting with a feasible solution and progressively attempting to improve it.
- **The "Best" Solution:** The optimal choice depends heavily on the specific problem's characteristics: its size, the relative importance of solution quality versus computational speed, and the precise

nature of its constraints. **Script 2.3 (Simulated Annealing)** appear to offer a strong balance for achieving good-quality solutions on moderately complex problems, and therefore shall be considered universally superior in this case.

4.2. Possible Extensions for VRPPD Solvers

The VRPPD remains a fertile ground for research and practical application. The analyzed solvers could be significantly enhanced or adapted through various extensions:

- **Advanced Metaheuristics:**
  - **Tabu Search:** Incorporate memory structures to guide the search away from recently visited solutions, helping to overcome local optima more effectively.
  - **Genetic Algorithms (GA):** Maintain a population of candidate solutions and apply evolutionary operators (crossover, mutation) to discover superior solutions, excellent for exploring diverse areas of the solution space.
  - **Ant Colony Optimization (ACO):** Leverage principles inspired by the foraging behavior of ants, a technique often well-suited for routing problems.
- **Hybrid Algorithmic Approaches:**
  - Combine the strengths of multiple techniques. For example, use a fast constructive heuristic to generate a high-quality initial solution for a more powerful metaheuristic like SA or a GA.
  - Integrate Machine Learning to learn effective heuristic choices or to dynamically tune algorithm parameters based on observed problem characteristics.
- **Richer Problem Definitions and Constraints:**

  Time Windows: Enforce that pickups and/or deliveries must occur within specified time intervals—a critical constraint in many real-world logistics operations.

  Dynamic VRPPD (DVRPPD): Develop capabilities to handle new requests that arrive in real-time while vehicles are already operational.

  Stochastic VRPPD (SVRPPD): Model and manage uncertainties, such as variable travel times, unpredictable service durations, or demand fluctuations.

  Multi-Objective Optimization: Optimize for several criteria simultaneously (e.g., minimizing total distance, minimizing the number of vehicles, maximizing customer satisfaction levels) using specialized techniques.

  Heterogeneous Fleet: Accommodate vehicles with differing capacities, operating costs, speeds, or specialized equipment (e.g., refrigeration).

  Backhauls: Allow vehicles to pick up new loads on their return trips after completing initial deliveries, improving asset utilization.

  Split Deliveries/Pickups: Permit a single large request to be serviced by multiple vehicles if it exceeds individual vehicle capacity or for efficiency.
- **Enhanced Algorithmic Components:**

  Sophisticated Intra-Route Optimizers: For a given sequence of stops assigned to a single vehicle, employ powerful Traveling Salesperson Problem (TSP) solvers or advanced heuristics (like Lin-Kernighan) to optimize the order of those stops.

Intelligent Insertion/Removal Heuristics: Design more sophisticated methods for deciding where and how to insert or remove requests from routes, considering longer-term impacts rather than just immediate costs.

Operations Research Techniques: For very large-scale problems, explore exact methods like Column Generation or Set Covering/Partitioning formulations, which can find optimal or near-optimal solutions by implicitly considering a vast number of potential routes.

- **Parallelization and Distributed Computing:** For population-based metaheuristics (like GAs) or to perform multiple independent runs of stochastic algorithms like SA, leverage parallel processing to significantly reduce overall computation time.
- **Integration with Real-World Systems and Data:** Incorporate real-time traffic data, GPS tracking, and Geographic Information Systems (GIS) for more accurate travel time and distance estimations, leading to more practical and robust solutions.

By pursuing these extensions, VRPPD solvers can become more powerful, flexible, and capable of addressing the increasingly complex logistics challenges faced in various industries.