Optimization of Last-Mile Logistics via Vehicle Routing Problem (MILP)

Authors: Munazza Nadir, Ray Zhang, Eesha Danish, Ryan Lin, Oyun Erdene Adilbish

Course: INDENG 240, Optimization Analytics (Fall 2025)

Date: Dec 17, 2025

Keywords: Mixed Integer Linear Programming (MILP),vehicle routing, last-mile logistics.

Abstract

We modeled a last-mile delivery system as a Vehicle Routing Problem (VRP). We used real-world pharmaceutical distribution dataset, and formulated as a Mixed-Integer Linear Program (MILP) but solved using Constraint Programming with Guided Local Search to handle scale. This was tested across single-day, multi-day, and variable demand scenarios such as: optimistic, pessimistic, and most-likely. We used Google OR-Tools due to the NP-hard complexity of the problem. Our computational experiments, spanning nine days of operation, demonstrated that the solver consistently found highly efficient solutions, requiring only 3 to 4 vehicles from the available fleet while successfully prioritizing fleet reduction through a minimization objective with given constraints.

1. Introduction

Last-mile delivery is often the most expensive part of the supply chain, accounting for up to 53% of total shipping costs[1]. The challenge is to deliver a set of homogeneous goods to geographically dispersed customers while minimizing two conflicting objectives: the number of vehicles used (fixed costs) and the total time utilized (variable costs). This project explores the Capacitated Vehicle Routing Problem with Time Windows (CVRP-TW). We utilize real-world delivery data to determine optimal routes for a fleet of 15 vehicles subject to capacity (weight and volume) and driver shift limits.

We found real-world dataset[2] with pharmaceutical distribution from an order management system. It includes nine days of most-likely, pessimistic, and optimistic matrices of delivery distance and time,as well as pairwise node distance and time.

2. Formulation (Course Method - MILP)

Sets and Parameters:

- $N = 1, \dots, n$: set of customers that needs to be visited.
- $V = 0 \cup N$: set of all nodes, including depot node *0*.
- t_{ij}: travel time between node i and j.
- d_i^W, d_i^V: demand of customer i in weight and volume.
- Q^W, Q^V: capacity limits of a vehicle (1200 weight, 600 volume).
- H: planning horizon (max shift time per vehicle).

Decision Variables:

- $x_{ijk} \in \{0,1\}$: 1 if vehicle k travels from i to j, 0 otherwise.
- $y_k \in \{0,1\}$: 1 if vehicle k is utilized (leaves the depot), 0 otherwise.
- $s_{ik} \geq 0$: Service start time for vehicle k at node i.

Objective Function: We minimize a weighted sum of fleet size and total travel time. Let $\alpha = 200$ be the penalty for activating a vehicle:

$$min\ (\ \alpha \sum_{k \in K} y_k + \sum_{k \in K} \sum_{i \in V} \sum_{j \in V\ j \neq i} t_{ij}\, x_{ijk})$$

Constraints: i. Flow Conservation: Each customer (node) must be visited exactly once; flow in equals flow out:

$\sum_{i \in N} x_{ij} = 1 \quad \forall i \in N$ (Inflow); $\sum_{j \in N} x_{ij} = 1 \quad \forall j \in N$ (Outflow)

ii. Capacity: The sum of demands on any route k must not exceed Q^w and Q^v:

$$\sum_{i \in N} \sum_{j \in N} d_j^w x_{ijk} \leq Q^w \quad \forall k \in V$$

iii. Time Windows & Horizon: Arrival times must be valid, and the total route duration cannot exceed H:

$$s_{0k} + route\_duration_k \leq H$$

iv. Vehicle Activation: If a route starts, the vehicle is marked as used: $\sum_{j \in N} \quad x_{0jk} \leq y_k$, $\forall i \in N$

3. Solution Methodology (Method Not Covered)

While small VRP instances can be solved to optimality using exact Branch-and-Cut algorithms (like those in Gurobi), these methods scale poorly for large datasets (NP-Hard). Therefore, we utilized Google OR-Tools, which employs Constraint Programming (CP) coupled with Metaheuristics. Unlike the Simplex method or Gradient Descent, this approach has 2 advantages: construction heuristic (path cheapest Arc strategy to greedily build initial feasible solution), and metaheuristic guided local search (GLS): escapes local minima by penalizing features (edges) of the current solution that have high cost, effectively changing the objective function dynamically to force the solver to explore new neighborhoods. This hybrid approach allows us to find near-optimal solutions for complex multi-constraint problems in reasonable computation time (capped at 120 seconds).

4. Computational Experiments

4.1 Phase I: Baseline Feasibility (Day 1) See Appendix Figure 1.1 & Code block 1.1

We first validated our model on the dataset: Day 1, most-likely scenario. The goal was to minimize fleet usage while satisfying the relaxed time window constraints, hence α is chosen a higher number to penalize the more usage of the fleet.

- Input: 80 orders, 5 available vehicles.
- Parameters: $\alpha = 200$, Weight Capacity =1200, Volume Capacity =600.
- Results: Feasible solution found, Vehicles Used: 3 out of 5, Max Total Time: 381 min/vehicle

4.2 Phase II: Multi-Day Consistency (Day 1 - Day 9) See Appendix Code block 1.2

- Metric: We measured the variance in "Vehicles Required" across different days.
- Hypothesis: If demand fluctuates significantly, a fixed fleet size of 15 might be inefficient on slow days or insufficient on busy days.
- Result: Optimization results show consistent feasible solutions with 3-4 vehicles with ranging demands and constraints.

Day	Orders (N)	Vehicles Used	Total Travel Time Cost (Min)	Max Route End Time (Min)
1	78	3	491	387
2	63	3	530	368
3	67	3	488	336
4	69	4	582	319
5	75	4	590	379
6	77	3	556	386
7	66	3	551	325
8	74	4	640	323
9	84	4	652	339

4.3 Phase III: Sensitivity Analysis (Optimistic vs. Pessimistic) on Day 9

We simulated two scenarios to test the resilience of our logistics network:

1. Optimistic (Low Friction): Travel times reduced by 15% (simulating no traffic) and lower service times.
2. Pessimistic (High Friction): Travel times increased by 20% and tighter time windows.

- Goal: Determine if the 15-vehicle fleet is robust enough to handle the "Pessimistic" scenario without dropping orders.

Metric	Optimistic Scenario	Pessimistic Scenario	Impact
Fleet Size	4 Vehicles	5 Vehicles	+1 Truck Required
Total Travel Time	521 Minutes	882 Minutes	+69% Traffic Delay
Max Shift Duration	379 min (Violation)	354 min (Safe)	Safety vs. Risk
Objective Cost	1321	1882	+42% Cost Increase
Operational State	Brittle (High Risk)	Robust (Low Risk)	–

Solver 1: Google OR-Tools Routing Solver, which is based on Constraint Programming (CP) combined with Metaheuristics. We did further analysis on state(pessimistic/optimistic) of each vehicle for day 9 (see appendix table 1.1 -1.4)

Solver 2: Branch and Bound: It never stopped running…

For pessimistic and optimistic scenario details and code block, see Appendix code block 1.3 and Appendix table 1.1-1.4.

5. Conclusion

From Phase I (4.1), we observed that the penalty successfully forced the solver to pack trucks tighter, prioritizing fleet reduction over purely shortest paths. Our model’s limitations are that it assumes homogeneous cargo compatibility and ignores real-time traffic updates. In Phase II, we strengthened the findings from Phase I and demonstrated the model's operational efficiency across multiple days with fluctuating demand. It consistently required 3-4 vehicles, much less than 15 fleets given. In Phase III (Sensitivity Analysis) heightened the network's resilience. While the pessimistic scenario used 5 vehicles and incurred a 69% increase in total travel time, all orders were delivered successfully. This proves that the solution methodology we tested is resilient and robust even in adverse conditions. However, the model is limited by the homogeneous cargo and real-time traffic dynamics. Future iterations could implement "Split Delivery" (allowing one order to be split across two trucks) to further optimize capacity usage given constraints. We could explore how to adapt to real-time traffic events and explore dynamic routing.

APPENDIX:

Appendix Figure 1: Day 1 - Model output

Appendix Table 1.1. Summary for day 9 (Optimistic scenario)

Metric	Value	Notes
Total Customers Served	84 / 84	100% Service Level.
Fleet Size Used	4 Vehicles	Highly Efficient. Maintained 4 trucks despite higher demand.
Total Travel Cost	521 min	Very low for 84 stops (~6.2 min/stop).
Objective Value	1,321	(4 Trucks × 200) + (521 Travel Cost).
Longest Shift	379 min	VIOLATION WARNING: This exceeds the standard 360 min deadline.(because we have buffer time in our constraint)
Metric	Value	Notes

Appendix Table 1.2. Statistics of each vehicle at day 9 (Optimistic scenario)

Vehicle ID	Stops	Shift End Time	Pure Travel Time	Efficiency Note
Vehicle 1	34	379 min	171 min	Overloaded. Exceeds 6-hour shift limit (360).
Vehicle 2	19	260 min	116 min	Comfortable shift (4h 20m).
Vehicle 3	15	219 min	111 min	Underutilized (~3.5 hours).
Vehicle 4	16	239 min	123 min	Underutilized (~4 hours).
TOTAL	84	-	521 min	-

Appendix Table 1.3. Summary of Day 9 (Pessimistic scenario)

Metric	Value	Notes
Total Customers Served	84 / 84	100% Service Level.
Fleet Size Used	5 Vehicles	+1 Vehicle compared to Day 9 Optimistic.
Total Travel Time	882 min	+69% increase over Optimistic (521 min).
Objective Value	1,882	(5 Trucks × 200) + (882 Travel Cost).
Longest Shift	354 min	SAFE. Vehicle 1 is close to the limit (360), but valid.

Appendix Table 1.4. Statistics of each vehicle at day 9 (Pessimistic scenario)

Vehicle ID	Stops	Shift End Time	Pure Travel Time	Efficiency Note
Vehicle 1	16	275 min	151 min	Very safe buffer (~1.5 hours slack).
Vehicle 2	25	354 min	206 min	Critical Route. Only 6 minutes of slack remains!

Vehicle 3	15	273 min	169 min	--
Vehicle 4	15	281 min	185 min	--
Vehicle 5	19	275 min	171 min	The "relief" vehicle taking the extra load.
TOTAL	84	-	882 min	-

Code Appendix

Code block 1.1

```
# %% Method 2

import pandas as pd
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import numpy as np

# --- 1. Load DataFrames ---
df_orders = pd.read_csv('orders_day1.csv')
df_distance = pd.read_csv('distance_day1.csv', index_col=0)
df_time = pd.read_csv('time_day1.csv', index_col=0)

# --- PARAMETERS BASED ON MIP FORMULATION ---
# Alpha (α): Coefficient to penalize the number of vehicles. Must be large to prioritize
# vehicle count minimization over total travel time minimization.
ALPHA_VEHICLE_PENALTY = 200

# Physical Fleet Specifications (for sanity checks and fleet sizing)
PHYSICAL_FLEET_SPECS = {
  'weight': 2000,  # Max weight capacity of each truck
  'volume': 1000   # Max volume capacity of each truck
}

# Buffer time to ensure depot is open when vehicles return
BUFFER_RETURN_TIME = 200

# Extra vehicles to add beyond theoretical minimum
ROUTING_BUFFER = 5

# ---------------------------------------------------------------------
# --- 2. Data Structure Preparation for OR-Tools ---
# ---------------------------------------------------------------------

import math

def create_data_model_smart(orders_df, distance_df, time_df, physical_fleet_specs={}):
  """
  Initializes VRP data dynamically based on input statistics to ensure feasibility.
```

```
physical_fleet_specs: dict containing 'weight_limit', 'volume_limit' of your REAL trucks.
"""
data = {}

# --- 1. Data Cleaning (Keep existing logic) ---
distance_df.dropna(how='all', axis=0, inplace=True)
distance_df.dropna(how='all', axis=1, inplace=True)
time_df.dropna(how='all', axis=0, inplace=True)
time_df.dropna(how='all', axis=1, inplace=True)

data['distance_matrix'] = distance_df.values.astype(int).tolist()
data['time_matrix'] = time_df.values.astype(int).tolist()

# --- 2. Smart Time Window Parsing ---
# Parse actual windows first to find the "Horizon" (Latest possible deadline)
parsed_windows = []
max_deadline_in_data = 0

# Depot is always (0,0) or (0, Open_Duration)
# Let's assume Depot is open as long as the latest customer needs.
parsed_windows.append((0, 0))

for tw_str in orders_df['TIME WINDOW']:
    # Parse the string "(900, 1200)" -> 900, 1200
    clean_str = tw_str.strip('()')
    if ',' in clean_str:
        parts = clean_str.split(',')
        start = int(parts[0])
        end = int(parts[1])
        parsed_windows.append((start, end))
        # Track the latest time anyone needs service
        if end > max_deadline_in_data:
            max_deadline_in_data = end
    else:
        # Fallback for bad data
        parsed_windows.append((0, 10000)) # Default fallback

# Update Depot's window to extend to the latest deadline (plus return trip buffer)
# This prevents the "Depot closed before driver returns" error.
```

```
    # approx time to drive back to depot after last delivery
    horizon = max_deadline_in_data + BUFFER_RETURN_TIME
    parsed_windows[0] = (0, horizon)

    data['time_windows'] = parsed_windows
    data['vehicle_max_travel_time'] = horizon # Set Horizon dynamically

    # --- 3. Demand & Capacity Sanity Check ---
    data['weights'] = [0] + orders_df['WEIGHT'].round().astype(int).tolist()

    # Notice: Multiplier is consistent (was 100 in your snippet, 1000 in previous. Check your data!)
    data['volumes'] = [0] + (orders_df['VOLUME'] * 100).round().astype(int).tolist()

    # CONSTANTS (Physical limits of your trucks)
    TRUCK_W_CAP = physical_fleet_specs.get('weight', 2000)
    TRUCK_V_CAP = physical_fleet_specs.get('volume', 1000)

    # Sanity Check: Does the biggest order fit in a truck?
    max_order_w = max(data['weights'])
    if max_order_w > TRUCK_W_CAP:
        raise ValueError(f"CRITICAL ERROR: Order exists with weight {max_order_w}, but truck limit is
{TRUCK_W_CAP}.")

    # --- 4. Smart Fleet Sizing (The Lower Bound Calculation) ---
    total_weight = sum(data['weights'])
    total_volume = sum(data['volumes'])

    # Minimum trucks needed purely for capacity (Bin Packing Lower Bound)
    min_trucks_weight = math.ceil(total_weight / TRUCK_W_CAP)
    min_trucks_volume = math.ceil(total_volume / TRUCK_V_CAP)

    theoretical_min_vehicles = max(min_trucks_weight, min_trucks_volume)

    # Add a "Routing Buffer" (e.g., 20% or +2 trucks)
    # Vehicles can rarely be 100% full because they run out of Time or Distance first.
    recommended_fleet_size = int(theoretical_min_vehicles * 1.2) + ROUTING_BUFFER

    print(f"--- Initialization Report ---")
```

```
  print(f"Total Weight: {total_weight} | Max Truck W: {TRUCK_W_CAP} -> Min Trucks:
{min_trucks_weight}")
  print(f"Total Volume: {total_volume} | Max Truck V: {TRUCK_V_CAP} -> Min Trucks:
{min_trucks_volume}")
  print(f"Latest Deadline found: {max_deadline_in_data}")
  print(f"Setting Fleet Size to: {recommended_fleet_size} (Theoretical Min: {theoretical_min_vehicles})")

  data['num_vehicles'] = recommended_fleet_size
  data['vehicle_capacities_weight'] = [TRUCK_W_CAP] * data['num_vehicles']
  data['vehicle_capacities_volume'] = [TRUCK_V_CAP] * data['num_vehicles']

  # --- 5. Other Data ---
  data['service_times'] = [0] + orders_df['SERVICE_TIME'].astype(int).tolist()
  data['depot'] = 0
  data['penalty'] = 100000 # Keep high

  return data

# ---------------------------------------------------------------
# --- 3. Initialize Solver Model and Constraints ---
# ---------------------------------------------------------------

def print_solution(data, manager, routing, solution):
  """Prints the solution found by the solver."""
  total_distance = 0
  total_time_cost = 0 # Cost is now based on time, not distance
  total_time = 0
  time_dimension = routing.GetDimensionOrDie('Time')

  # Calculate the total travel time for the objective reporting
  for vehicle_id in range(data['num_vehicles']):
      index = routing.Start(vehicle_id)
      if routing.IsEnd(solution.Value(routing.NextVar(index))):
          continue

      while not routing.IsEnd(index):
          previous_index = index
          index = solution.Value(routing.NextVar(index))
```

```
        # The actual total travel time component of the objective
        from_node = manager.IndexToNode(previous_index)
        to_node = manager.IndexToNode(index)
        # The cost being minimized is based on the time matrix
        total_time_cost += data['time_matrix'][from_node][to_node]

  # Calculate objective based on MIP formulation: alpha * Yk + sum(tij * xijk)
  num_used_vehicles = len([v for v in range(data['num_vehicles']) if not
routing.IsEnd(solution.Value(routing.NextVar(routing.Start(v))))])
  vehicle_penalty_component = num_used_vehicles * ALPHA_VEHICLE_PENALTY

  # NOTE: The OR-Tools objective value here may include dropped node penalties,
  # but the custom calculation below reflects the MIP goal:
  mip_objective = vehicle_penalty_component + total_time_cost

  print(f'OR-Tools Objective (Time Cost + Penalties): {solution.ObjectiveValue()}')
  print(f'MIP Objective (Alpha*Vehicles + Total Time): {mip_objective} ({num_used_vehicles} vehicles *
{ALPHA_VEHICLE_PENALTY} + {total_time_cost})')

  # Print routes (unchanged)
  for vehicle_id in range(data['num_vehicles']):
      index = routing.Start(vehicle_id)
      if routing.IsEnd(solution.Value(routing.NextVar(index))):
          continue

      plan_output = f'Route for vehicle {vehicle_id} (Depot: 0):'
      route_distance = 0

      while not routing.IsEnd(index):
          time_var = time_dimension.CumulVar(index)
          node_index = manager.IndexToNode(index)

          previous_index = index
          index = solution.Value(routing.NextVar(index))

          route_distance += routing.GetArcCostForVehicle(
              previous_index, index, vehicle_id
          )
```

```
            plan_output += (
                f' {node_index} -> Time({solution.Min(time_var)})'
            )

        time_var = time_dimension.CumulVar(index)
        plan_output += (
            f' {manager.IndexToNode(index)} -> Time({solution.Min(time_var)})'
        )
        plan_output += f'\n  Route Distance: {route_distance}'
        plan_output += f'\n  Route End Time: {solution.Min(time_var)}'

        print(plan_output)
        total_distance += route_distance
        total_time = max(total_time, solution.Min(time_var))

    print(f'\nTotal distance of all used routes: {total_distance}')
    print(f'Max route end time: {total_time}')
    print(f'Total travel time cost: {total_time_cost} (The minimized sum component)')

    # --- Report Dropped Nodes ---
    dropped_nodes = []
#    for node in range(1, len(data['distance_matrix'])):
#        if solution.Value(routing.NextVar(manager.NodeToIndex(node))) == manager.NodeToIndex(node):
#            dropped_nodes.append(node)

    if dropped_nodes:
        print(f'\n!!! WARNING: {len(dropped_nodes)} Nodes Were DROPPED (Unserved) !!!')
        print(f'Dropped Nodes (NODE_ID): {dropped_nodes}')
    else:
        print('\nSUCCESS: All nodes were served.')


def solve_vrp():
    """Entry point for the VRP solver with MIP objective."""
    data = create_data_model_smart(df_orders, df_distance, df_time)

    manager = pywrapcp.RoutingIndexManager(
        len(data['distance_matrix']), data['num_vehicles'], data['depot']
    )
```

```
routing = pywrapcp.RoutingModel(manager)

# --- A. Define Cost (Time - as per MIP Objective) ---
def time_cost_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    # The cost minimized is the travel time (t_ij)
    return data['time_matrix'][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(time_cost_callback)
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

# A.1. Set Objective to Minimize Time Cost (t_ij) AND Penalize Vehicles (alpha * y_k)

# Fixed cost (alpha * y_k): Apply a large penalty for using each vehicle
for vehicle_id in range(data['num_vehicles']):
    routing.SetFixedCostOfVehicle(ALPHA_VEHICLE_PENALTY, vehicle_id)

# A.2. Add Dropped Node Penalty
for node in range(1, len(data['distance_matrix'])):
    routing.AddDisjunction([manager.NodeToIndex(node)], data['penalty'])

# --- B. Add Capacity Constraints (Weight and Volume) ---

def add_capacity_dimension(capacity_name, capacities, demands):
    def demand_callback(index):
        node = manager.IndexToNode(index)
        return demands[node]

    demand_callback_index = routing.RegisterUnaryTransitCallback(demand_callback)

    # Capacity bounds are imposed at every node [cite: 87]
    routing.AddDimensionWithVehicleCapacity(
        demand_callback_index,
        0, # Slack
        capacities,
        True, # This dimension is cumulative
        capacity_name
```

```
    )

add_capacity_dimension('WeightCapacity', data['vehicle_capacities_weight'], data['weights'])
add_capacity_dimension('VolumeCapacity', data['vehicle_capacities_volume'], data['volumes'])


# --- C. Add Time Window Constraints ---

def time_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    travel_time = data['time_matrix'][from_node][to_node]
    service_time = data['service_times'][from_node]
    # Time propagation: T_jk >= T_ik + s_i + t_ij [cite: 99]
    return travel_time + service_time

time_callback_index = routing.RegisterTransitCallback(time_callback)

# Time dimension: tracks T_ik (time vehicle k starts service at node i) [cite: 45]
routing.AddDimension(
    time_callback_index,
    0,                          # slack_max (0 is fine here)
    data['vehicle_max_travel_time'], # Planning horizon H [cite: 28]
    False,
    'Time'
)
time_dimension = routing.GetDimensionOrDie('Time')

# Apply Time Windows [cite: 94]
for node in range(len(data['time_windows'])):
    index = manager.NodeToIndex(node)
    start, end = data['time_windows'][node]
    time_dimension.CumulVar(index).SetRange(start, end)

# Apply Route Duration Constraint (limited by H) [cite: 104]
for i in range(data['num_vehicles']):
    time_dimension.SetSpanUpperBoundForVehicle(
        data['vehicle_max_travel_time'],
        i
```

```
    )

  # --- D. Set Search Parameters and Solve ---
  search_parameters = pywrapcp.DefaultRoutingSearchParameters()
  search_parameters.first_solution_strategy = (
      routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
  )
  search_parameters.local_search_metaheuristic = (
      routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
  )
  search_parameters.time_limit.seconds = 120

  # Solve the problem
  solution = routing.SolveWithParameters(search_parameters)

  # --- E. Print Solution ---
  if solution:
      print_solution(data, manager, routing, solution)
  else:
      print('No solution found!')


# ---------------------------------------------------------------------
# --- EXECUTION ---
# ---------------------------------------------------------------------

print("Running Vehicle Routing Problem Solver (MIP Objective: Minimize Vehicles, then Time)...")
solve_vrp()
```

Appendix Code block 1.2

```
# !pip install ortools

# %% Method 2

import pandas as pd
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import numpy as np

# --- 1. Load DataFrames ---
df_orders = pd.read_csv('orders_day1.csv')
df_distance = pd.read_excel("distance_matrix_1.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_1.xlsx', index_col=0)

# --- PARAMETERS BASED ON MIP FORMULATION ---
# Alpha (α): Coefficient to penalize the number of vehicles. Must be
large to prioritize
# vehicle count minimization over total travel time minimization.
ALPHA_VEHICLE_PENALTY = 200

# Physical Fleet Specifications (for sanity checks and fleet sizing)
PHYSICAL_FLEET_SPECS = {
    'weight': 2000,  # Max weight capacity of each truck
    'volume': 1000   # Max volume capacity of each truck
}

# Buffer time to ensure depot is open when vehicles return
BUFFER_RETURN_TIME = 200

# Extra vehicles to add beyond theoretical minimum
ROUTING_BUFFER = 5

#
----------------------------------------------------------------------
# --- 2. Data Structure Preparation for OR-Tools ---
#
----------------------------------------------------------------------

import math

def create_data_model_smart(orders_df, distance_df, time_df,
physical_fleet_specs={}):
    """
    Initializes VRP data dynamically based on input statistics to
ensure feasibility.

    physical_fleet_specs: dict containing 'weight_limit',
'volume_limit' of your REAL trucks.
    """
    data = {}
```

```
    # --- 1. Data Cleaning (Keep existing logic) ---
    distance_df.dropna(how='all', axis=0, inplace=True)
    distance_df.dropna(how='all', axis=1, inplace=True)
    time_df.dropna(how='all', axis=0, inplace=True)
    time_df.dropna(how='all', axis=1, inplace=True)

    data['distance_matrix'] = distance_df.values.astype(int).tolist()
    data['time_matrix'] = time_df.values.astype(int).tolist()

    # --- 2. Smart Time Window Parsing ---
    # Parse actual windows first to find the "Horizon" (Latest
possible deadline)
    parsed_windows = []
    max_deadline_in_data = 0

    # Depot is always (0,0) or (0, Open_Duration)
    # Let's assume Depot is open as long as the latest customer needs.
    parsed_windows.append((0, 0))

    for tw_str in orders_df['TIME WINDOW']:
        # Parse the string "(900, 1200)" -> 900, 1200
        clean_str = tw_str.strip('()')
        if ',' in clean_str:
            parts = clean_str.split(',')
            start = int(parts[0])
            end = int(parts[1])
            parsed_windows.append((start, end))
            # Track the latest time anyone needs service
            if end > max_deadline_in_data:
                max_deadline_in_data = end
        else:
            # Fallback for bad data
            parsed_windows.append((0, 10000)) # Default fallback

    # Update Depot's window to extend to the latest deadline (plus
return trip buffer)
    # This prevents the "Depot closed before driver returns" error.
    # approx time to drive back to depot after last delivery
    horizon = max_deadline_in_data + BUFFER_RETURN_TIME
    parsed_windows[0] = (0, horizon)

    data['time_windows'] = parsed_windows
    data['vehicle_max_travel_time'] = horizon # Set Horizon
dynamically

    # --- 3. Demand & Capacity Sanity Check ---
    data['weights'] = [0] +
orders_df['WEIGHT'].round().astype(int).tolist()

    # Notice: Multiplier is consistent (was 100 in your snippet, 1000
```

```
in previous. Check your data!)
    data['volumes'] = [0] + (orders_df['VOLUME'] *
100).round().astype(int).tolist()

    # CONSTANTS (Physical limits of your trucks)
    TRUCK_W_CAP = physical_fleet_specs.get('weight', 2000)
    TRUCK_V_CAP = physical_fleet_specs.get('volume', 1000)

    # Sanity Check: Does the biggest order fit in a truck?
    max_order_w = max(data['weights'])
    if max_order_w > TRUCK_W_CAP:
        raise ValueError(f"CRITICAL ERROR: Order exists with weight
{max_order_w}, but truck limit is {TRUCK_W_CAP}.")

    # --- 4. Smart Fleet Sizing (The Lower Bound Calculation) ---
    total_weight = sum(data['weights'])
    total_volume = sum(data['volumes'])

    # Minimum trucks needed purely for capacity (Bin Packing Lower
Bound)
    min_trucks_weight = math.ceil(total_weight / TRUCK_W_CAP)
    min_trucks_volume = math.ceil(total_volume / TRUCK_V_CAP)

    theoretical_min_vehicles = max(min_trucks_weight,
min_trucks_volume)

    # Add a "Routing Buffer" (e.g., 20% or +2 trucks)
    # Vehicles can rarely be 100% full because they run out of Time or
Distance first.
    recommended_fleet_size = int(theoretical_min_vehicles * 1.2) +
ROUTING_BUFFER

    print(f"--- Initialization Report ---")
    print(f"Total Weight: {total_weight} | Max Truck W: {TRUCK_W_CAP}
-> Min Trucks: {min_trucks_weight}")
    print(f"Total Volume: {total_volume} | Max Truck V: {TRUCK_V_CAP}
-> Min Trucks: {min_trucks_volume}")
    print(f"Latest Deadline found: {max_deadline_in_data}")
    print(f"Setting Fleet Size to: {recommended_fleet_size}
(Theoretical Min: {theoretical_min_vehicles})")

    data['num_vehicles'] = recommended_fleet_size
    data['vehicle_capacities_weight'] = [TRUCK_W_CAP] *
data['num_vehicles']
    data['vehicle_capacities_volume'] = [TRUCK_V_CAP] *
data['num_vehicles']

    # --- 5. Other Data ---
    data['service_times'] = [0] +
orders_df['SERVICE_TIME'].astype(int).tolist()
```

```
    data['depot'] = 0
    data['penalty'] = 100000 # Keep high

    return data

#
---------------------------------------------------------------------
# --- 3. Initialize Solver Model and Constraints ---
#
---------------------------------------------------------------------

def print_solution(data, manager, routing, solution):
   """Prints the solution found by the solver."""
   total_distance = 0
   total_time_cost = 0 # Cost is now based on time, not distance
   total_time = 0
   time_dimension = routing.GetDimensionOrDie('Time')

   # Calculate the total travel time for the objective reporting
   for vehicle_id in range(data['num_vehicles']):
       index = routing.Start(vehicle_id)
       if routing.IsEnd(solution.Value(routing.NextVar(index))):
           continue

       while not routing.IsEnd(index):
           previous_index = index
           index = solution.Value(routing.NextVar(index))

           # The actual total travel time component of the objective
           from_node = manager.IndexToNode(previous_index)
           to_node = manager.IndexToNode(index)
           # The cost being minimized is based on the time matrix
           total_time_cost += data['time_matrix'][from_node][to_node]

   # Calculate objective based on MIP formulation: alpha * Yk +
sum(tij * xijk)
   num_used_vehicles = len([v for v in range(data['num_vehicles']) if
not routing.IsEnd(solution.Value(routing.NextVar(routing.Start(v))))])
   vehicle_penalty_component = num_used_vehicles *
ALPHA_VEHICLE_PENALTY

   # NOTE: The OR-Tools objective value here may include dropped node
penalties,
   # but the custom calculation below reflects the MIP goal:
   mip_objective = vehicle_penalty_component + total_time_cost

   print(f'OR-Tools Objective (Time Cost + Penalties):
{solution.ObjectiveValue()}')
   print(f'MIP Objective (Alpha*Vehicles + Total Time):
{mip_objective} ({num_used_vehicles} vehicles *
```

```
{ALPHA_VEHICLE_PENALTY} + {total_time_cost})')

    # Print routes (unchanged)
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        if routing.IsEnd(solution.Value(routing.NextVar(index))):
            continue

        plan_output = f'Route for vehicle {vehicle_id} (Depot: 0):'
        route_distance = 0

        while not routing.IsEnd(index):
            time_var = time_dimension.CumulVar(index)
            node_index = manager.IndexToNode(index)

            previous_index = index
            index = solution.Value(routing.NextVar(index))

            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id
            )

            plan_output += (
                f' {node_index} -> Time({solution.Min(time_var)})'
            )

        time_var = time_dimension.CumulVar(index)
        plan_output += (
            f' {manager.IndexToNode(index)} ->
Time({solution.Min(time_var)})'
        )
        plan_output += f'\n  Route Distance: {route_distance}'
        plan_output += f'\n  Route End Time: {solution.Min(time_var)}'

        print(plan_output)
        total_distance += route_distance
        total_time = max(total_time, solution.Min(time_var))

    print(f'\nTotal distance of all used routes: {total_distance}')
    print(f'Max route end time: {total_time}')
    print(f'Total travel time cost: {total_time_cost} (The minimized
sum component)')

    # --- Report Dropped Nodes ---
    dropped_nodes = []
#     for node in range(1, len(data['distance_matrix'])):
#         if solution.Value(routing.NextVar(manager.NodeToIndex(node)))
== manager.NodeToIndex(node):
#             dropped_nodes.append(node)
```

```
    if dropped_nodes:
        print(f'\n!!! WARNING: {len(dropped_nodes)} Nodes Were DROPPED
(Unserved) !!!')
        print(f'Dropped Nodes (NODE_ID): {dropped_nodes}')
    else:
        print('\nSUCCESS: All nodes were served.')


def solve_vrp():
    """Entry point for the VRP solver with MIP objective."""
    data = create_data_model_smart(df_orders, df_distance, df_time)

    manager = pywrapcp.RoutingIndexManager(
        len(data['distance_matrix']), data['num_vehicles'],
data['depot']
    )

    routing = pywrapcp.RoutingModel(manager)

    # --- A. Define Cost (Time - as per MIP Objective) ---
    def time_cost_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        # The cost minimized is the travel time (t_ij)
        return data['time_matrix'][from_node][to_node]

    transit_callback_index =
routing.RegisterTransitCallback(time_cost_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # A.1. Set Objective to Minimize Time Cost (t_ij) AND Penalize
Vehicles (alpha * y_k)

    # Fixed cost (alpha * y_k): Apply a large penalty for using each
vehicle
    for vehicle_id in range(data['num_vehicles']):
        routing.SetFixedCostOfVehicle(ALPHA_VEHICLE_PENALTY,
vehicle_id)

    # A.2. Add Dropped Node Penalty
    for node in range(1, len(data['distance_matrix'])):
        routing.AddDisjunction([manager.NodeToIndex(node)],
data['penalty'])

    # --- B. Add Capacity Constraints (Weight and Volume) ---

    def add_capacity_dimension(capacity_name, capacities, demands):
        def demand_callback(index):
            node = manager.IndexToNode(index)
            return demands[node]
```

```
        demand_callback_index =
routing.RegisterUnaryTransitCallback(demand_callback)

        # Capacity bounds are imposed at every node [cite: 87]
        routing.AddDimensionWithVehicleCapacity(
            demand_callback_index,
            0, # Slack
            capacities,
            True, # This dimension is cumulative
            capacity_name
        )

    add_capacity_dimension('WeightCapacity',
data['vehicle_capacities_weight'], data['weights'])
    add_capacity_dimension('VolumeCapacity',
data['vehicle_capacities_volume'], data['volumes'])


    # --- C. Add Time Window Constraints ---

    def time_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        travel_time = data['time_matrix'][from_node][to_node]
        service_time = data['service_times'][from_node]
        # Time propagation: T_jk >= T_ik + s_i + t_ij [cite: 99]
        return travel_time + service_time

    time_callback_index =
routing.RegisterTransitCallback(time_callback)

    # Time dimension: tracks T_ik (time vehicle k starts service at
node i) [cite: 45]
    routing.AddDimension(
        time_callback_index,
        0,                                # slack_max (0 is fine here)
        data['vehicle_max_travel_time'], # Planning horizon H [cite:
28]
        False,
        'Time'
    )
    time_dimension = routing.GetDimensionOrDie('Time')

    # Apply Time Windows [cite: 94]
    for node in range(len(data['time_windows'])):
        index = manager.NodeToIndex(node)
        start, end = data['time_windows'][node]
        time_dimension.CumulVar(index).SetRange(start, end)
```

```
    # Apply Route Duration Constraint (limited by H) [cite: 104]
    for i in range(data['num_vehicles']):
        time_dimension.SetSpanUpperBoundForVehicle(
            data['vehicle_max_travel_time'],
            i
        )

    # --- D. Set Search Parameters and Solve ---
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
    )
    search_parameters.time_limit.seconds = 120

    # Solve the problem
    solution = routing.SolveWithParameters(search_parameters)

    # --- E. Print Solution ---
    if solution:
        print_solution(data, manager, routing, solution)
    else:
        print('No solution found!')


#
# ---------------------------------------------------------------------
# --- EXECUTION ---
#
# ---------------------------------------------------------------------

# print("Running Vehicle Routing Problem Solver (MIP Objective:
Minimize Vehicles, then Time)...")
# print("Day 1:")
# solve_vrp()

def check_day(df_orders, df_distance, df_time, day):
    print(f"Shapes Day {day}: orders={df_orders.shape},
dist={df_distance.shape}, time={df_time.shape}")

    assert df_distance.shape[0] == df_distance.shape[1], "Distance
matrix not square"
    assert df_time.shape[0] == df_time.shape[1], "Time matrix not
square"
    assert df_distance.shape == df_time.shape, "Distance and time
shapes differ"

    n_orders = len(df_orders)
```

```
    n_nodes = df_distance.shape[0]
    assert n_nodes == n_orders + 1, f"Expected matrix size
{n_orders+1}x{n_orders+1} (depot+orders), got {n_nodes}x{n_nodes}"

print("Day 1:")
check_day(df_orders, df_distance, df_time, 1)
solve_vrp()
```

```
Day 1:
Shapes Day 1: orders=(78, 7), dist=(79, 79), time=(79, 79)
--- Initialization Report ---
Total Weight: 1877 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 898 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1091
MIP Objective (Alpha*Vehicles + Total Time): 1091 (3 vehicles * 200 +
491)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 2 -> Time(18) 4 ->
Time(32) 26 -> Time(41) 28 -> Time(62) 24 -> Time(77) 25 -> Time(89)
53 -> Time(96) 52 -> Time(110) 23 -> Time(123) 19 -> Time(131) 55 ->
Time(138) 20 -> Time(156) 22 -> Time(172) 50 -> Time(180) 21 ->
Time(187) 49 -> Time(196) 48 -> Time(205) 47 -> Time(211) 45 ->
Time(220) 44 -> Time(230) 46 -> Time(240) 42 -> Time(248) 43 ->
Time(257) 41 -> Time(267) 60 -> Time(283) 75 -> Time(291) 62 ->
Time(300) 61 -> Time(312) 74 -> Time(321) 73 -> Time(331) 70 ->
Time(345) 71 -> Time(355) 0 -> Time(387)
  Route Distance: 359
  Route End Time: 387
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 16 -> Time(20) 11 ->
Time(35) 5 -> Time(47) 56 -> Time(57) 54 -> Time(68) 58 -> Time(79) 57
-> Time(93) 59 -> Time(101) 15 -> Time(109) 76 -> Time(119) 77 ->
Time(128) 78 -> Time(140) 72 -> Time(150) 69 -> Time(166) 68 ->
Time(186) 64 -> Time(193) 13 -> Time(203) 67 -> Time(210) 66 ->
Time(215) 65 -> Time(220) 63 -> Time(225) 9 -> Time(233) 3 ->
Time(239) 8 -> Time(248) 1 -> Time(255) 10 -> Time(264) 6 -> Time(286)
0 -> Time(316)
  Route Distance: 364
  Route End Time: 316
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 7 -> Time(22) 14 ->
Time(51) 17 -> Time(65) 12 -> Time(73) 18 -> Time(84) 27 -> Time(113)
30 -> Time(127) 32 -> Time(137) 31 -> Time(142) 29 -> Time(157) 51 ->
Time(167) 35 -> Time(180) 40 -> Time(195) 39 -> Time(201) 38 ->
Time(207) 37 -> Time(214) 36 -> Time(223) 34 -> Time(237) 33 ->
Time(247) 0 -> Time(284)
  Route Distance: 368
  Route End Time: 284

Total distance of all used routes: 1091
Max route end time: 387
```

```
Total travel time cost: 491 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 2:")
df_orders = pd.read_csv('orders_day2.csv')
df_distance = pd.read_excel("distance_matrix_2.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_2.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 2)
solve_vrp()
```

```
Day 2:
Shapes Day 2: orders=(63, 7), dist=(64, 64), time=(64, 64)
--- Initialization Report ---
Total Weight: 1394 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 519 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1130
MIP Objective (Alpha*Vehicles + Total Time): 1130 (3 vehicles * 200 +
530)
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 58 -> Time(17) 44 ->
Time(38) 48 -> Time(48) 45 -> Time(55) 49 -> Time(77) 47 -> Time(93)
57 -> Time(102) 61 -> Time(126) 62 -> Time(146) 60 -> Time(152) 63 ->
Time(159) 43 -> Time(178) 19 -> Time(206) 18 -> Time(211) 0 ->
Time(241)
  Route Distance: 341
  Route End Time: 241
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 34 -> Time(21) 8 ->
Time(55) 40 -> Time(87) 51 -> Time(98) 50 -> Time(108) 4 -> Time(127)
6 -> Time(136) 7 -> Time(154) 9 -> Time(168) 15 -> Time(176) 10 ->
Time(181) 55 -> Time(192) 52 -> Time(199) 53 -> Time(204) 5 ->
Time(213) 46 -> Time(227) 1 -> Time(241) 3 -> Time(246) 59 ->
Time(264) 36 -> Time(281) 37 -> Time(289) 35 -> Time(300) 28 ->
Time(314) 29 -> Time(324) 27 -> Time(342) 0 -> Time(368)
  Route Distance: 412
  Route End Time: 368
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 56 -> Time(21) 54 ->
Time(30) 17 -> Time(49) 13 -> Time(56) 11 -> Time(65) 14 -> Time(74)
16 -> Time(84) 12 -> Time(109) 2 -> Time(124) 25 -> Time(132) 21 ->
Time(143) 22 -> Time(157) 23 -> Time(178) 24 -> Time(192) 26 ->
Time(200) 20 -> Time(207) 30 -> Time(217) 32 -> Time(228) 31 ->
Time(237) 33 -> Time(249) 39 -> Time(266) 38 -> Time(279) 42 ->
Time(287) 41 -> Time(300) 0 -> Time(329)
  Route Distance: 377
  Route End Time: 329
```

```
Total distance of all used routes: 1130
Max route end time: 368
Total travel time cost: 530 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 3:")
df_orders = pd.read_csv('orders_day3.csv')
df_distance = pd.read_excel("distance_matrix_3.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_3.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 3)
solve_vrp()
```

```
Day 3:
Shapes Day 3: orders=(67, 7), dist=(68, 68), time=(68, 68)
--- Initialization Report ---
Total Weight: 682 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 401 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1088
MIP Objective (Alpha*Vehicles + Total Time): 1088 (3 vehicles * 200 +
488)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 50 -> Time(16) 42 ->
Time(25) 46 -> Time(34) 17 -> Time(49) 20 -> Time(63) 23 -> Time(77)
22 -> Time(86) 21 -> Time(100) 2 -> Time(126) 6 -> Time(135) 3 ->
Time(152) 5 -> Time(157) 7 -> Time(164) 39 -> Time(192) 40 ->
Time(198) 44 -> Time(205) 41 -> Time(214) 0 -> Time(232)
  Route Distance: 336
  Route End Time: 232
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 31 -> Time(18) 26 ->
Time(25) 25 -> Time(42) 24 -> Time(55) 8 -> Time(82) 9 -> Time(92) 4 -
> Time(107) 15 -> Time(117) 13 -> Time(127) 16 -> Time(133) 14 ->
Time(138) 11 -> Time(150) 12 -> Time(166) 10 -> Time(182) 27 ->
Time(195) 28 -> Time(209) 30 -> Time(215) 38 -> Time(223) 33 ->
Time(233) 32 -> Time(240) 34 -> Time(251) 29 -> Time(260) 35 ->
Time(279) 36 -> Time(289) 37 -> Time(308) 0 -> Time(336)
  Route Distance: 380
  Route End Time: 336
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 47 -> Time(12) 49 ->
Time(25) 57 -> Time(31) 61 -> Time(40) 53 -> Time(45) 51 -> Time(51)
66 -> Time(63) 63 -> Time(71) 65 -> Time(86) 58 -> Time(97) 64 ->
Time(108) 59 -> Time(118) 45 -> Time(134) 67 -> Time(148) 1 ->
Time(176) 52 -> Time(218) 56 -> Time(224) 54 -> Time(229) 55 ->
Time(238) 60 -> Time(243) 62 -> Time(250) 19 -> Time(258) 18 ->
Time(269) 48 -> Time(278) 43 -> Time(284) 0 -> Time(304)
  Route Distance: 372
```

```
  Route End Time: 304

Total distance of all used routes: 1088
Max route end time: 336
Total travel time cost: 488 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 4:")
df_orders = pd.read_csv('orders_day4.csv')
df_distance = pd.read_excel("distance_matrix_4.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_4.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 4)
solve_vrp()
```

```


Day 4:
Shapes Day 4: orders=(69, 7), dist=(70, 70), time=(70, 70)
--- Initialization Report ---
Total Weight: 1690 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 469 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1382
MIP Objective (Alpha*Vehicles + Total Time): 1382 (4 vehicles * 200 +
582)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 30 -> Time(31) 31 ->
Time(47) 57 -> Time(63) 53 -> Time(72) 54 -> Time(87) 55 -> Time(95)
46 -> Time(108) 41 -> Time(116) 43 -> Time(129) 44 -> Time(137) 40 ->
Time(144) 45 -> Time(152) 42 -> Time(168) 49 -> Time(181) 48 ->
Time(186) 50 -> Time(191) 51 -> Time(202) 47 -> Time(211) 52 ->
Time(223) 69 -> Time(238) 63 -> Time(264) 0 -> Time(291)
  Route Distance: 359
  Route End Time: 291
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 8 -> Time(21) 7 ->
Time(36) 28 -> Time(62) 29 -> Time(69) 26 -> Time(75) 32 -> Time(92)
27 -> Time(108) 56 -> Time(126) 23 -> Time(131) 35 -> Time(144) 37 ->
Time(158) 34 -> Time(166) 36 -> Time(176) 38 -> Time(196) 39 ->
Time(210) 33 -> Time(230) 25 -> Time(245) 24 -> Time(250) 12 ->
Time(266) 11 -> Time(277) 9 -> Time(285) 10 -> Time(293) 0 ->
Time(319)
  Route Distance: 375
  Route End Time: 319
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 18 -> Time(29) 17 ->
Time(46) 13 -> Time(66) 14 -> Time(73) 15 -> Time(89) 16 -> Time(94)
21 -> Time(104) 19 -> Time(121) 20 -> Time(126) 22 -> Time(142) 4 ->
Time(161) 0 -> Time(196)
  Route Distance: 328
```

```
  Route End Time: 196
Route for vehicle 5 (Depot: 0): 0 -> Time(0) 62 -> Time(26) 66 ->
Time(42) 65 -> Time(48) 61 -> Time(57) 59 -> Time(66) 60 -> Time(82)
58 -> Time(91) 67 -> Time(104) 68 -> Time(116) 64 -> Time(123) 1 ->
Time(139) 3 -> Time(147) 2 -> Time(158) 5 -> Time(166) 6 -> Time(173)
0 -> Time(204)
  Route Distance: 320
  Route End Time: 204

Total distance of all used routes: 1382
Max route end time: 319
Total travel time cost: 582 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 5:")
df_orders = pd.read_csv('orders_day5.csv')
df_orders.columns = df_orders.columns.str.strip()

if 'TIME WINDOW' not in df_orders.columns and 'TIME_WINDOW' in
df_orders.columns:
    df_orders = df_orders.rename(columns={'TIME_WINDOW': 'TIME
WINDOW'})
df_distance = pd.read_excel("distance_matrix_5.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_5.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 5)
solve_vrp()
```

```
Day 5:
Shapes Day 5: orders=(75, 7), dist=(76, 76), time=(76, 76)
--- Initialization Report ---
Total Weight: 1271 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 446 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1390
MIP Objective (Alpha*Vehicles + Total Time): 1390 (4 vehicles * 200 +
590)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 47 -> Time(25) 45 ->
Time(31) 46 -> Time(36) 48 -> Time(45) 49 -> Time(53) 51 -> Time(60)
50 -> Time(69) 44 -> Time(81) 42 -> Time(96) 43 -> Time(115) 40 ->
Time(139) 33 -> Time(156) 38 -> Time(175) 37 -> Time(197) 56 ->
Time(202) 41 -> Time(216) 16 -> Time(235) 0 -> Time(264)
  Route Distance: 348
  Route End Time: 264
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 9 -> Time(19) 60 ->
Time(36) 57 -> Time(46) 58 -> Time(55) 62 -> Time(63) 59 -> Time(72)
```

```
63 -> Time(82) 65 -> Time(98) 61 -> Time(106) 36 -> Time(117) 39 ->
Time(138) 30 -> Time(169) 35 -> Time(178) 28 -> Time(184) 34 ->
Time(192) 29 -> Time(200) 32 -> Time(209) 64 -> Time(220) 31 ->
Time(226) 27 -> Time(235) 71 -> Time(247) 73 -> Time(260) 67 ->
Time(273) 68 -> Time(279) 69 -> Time(286) 70 -> Time(298) 66 ->
Time(311) 72 -> Time(325) 75 -> Time(336) 74 -> Time(346) 0 ->
Time(379)
  Route Distance: 403
  Route End Time: 379
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 55 -> Time(18) 54 ->
Time(28) 52 -> Time(39) 53 -> Time(45) 3 -> Time(63) 11 -> Time(85) 22
-> Time(96) 25 -> Time(109) 21 -> Time(128) 18 -> Time(138) 12 ->
Time(158) 17 -> Time(175) 19 -> Time(196) 20 -> Time(204) 24 ->
Time(217) 23 -> Time(225) 26 -> Time(238) 0 -> Time(268)
  Route Distance: 356
  Route End Time: 268
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 1 -> Time(16) 4 ->
Time(26) 5 -> Time(38) 6 -> Time(45) 7 -> Time(51) 10 -> Time(62) 13 -
> Time(82) 14 -> Time(89) 15 -> Time(102) 8 -> Time(120) 2 ->
Time(128) 0 -> Time(155)
  Route Distance: 283
  Route End Time: 155

Total distance of all used routes: 1390
Max route end time: 379
Total travel time cost: 590 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 6:")
df_orders = pd.read_csv('orders_day6.csv')
df_orders.columns = df_orders.columns.str.strip()

if 'TIME WINDOW' not in df_orders.columns and 'TIME_WINDOW' in
df_orders.columns:
    df_orders = df_orders.rename(columns={'TIME_WINDOW': 'TIME
WINDOW'})
df_distance = pd.read_excel("distance_matrix_6.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_6.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 6)
solve_vrp()
```

```
Day 6:
Shapes Day 6: orders=(77, 7), dist=(78, 78), time=(78, 78)
--- Initialization Report ---
Total Weight: 1302 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 455 | Max Truck V: 1000 -> Min Trucks: 1
```

```
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1156
MIP Objective (Alpha*Vehicles + Total Time): 1156 (3 vehicles * 200 +
556)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 75 -> Time(18) 73 ->
Time(24) 67 -> Time(45) 37 -> Time(65) 39 -> Time(82) 36 -> Time(96)
35 -> Time(112) 46 -> Time(129) 43 -> Time(146) 40 -> Time(155) 48 ->
Time(172) 44 -> Time(181) 42 -> Time(190) 47 -> Time(205) 45 ->
Time(212) 49 -> Time(219) 41 -> Time(233) 64 -> Time(247) 60 ->
Time(255) 70 -> Time(263) 61 -> Time(270) 63 -> Time(276) 62 ->
Time(283) 59 -> Time(296) 0 -> Time(321)
  Route Distance: 377
  Route End Time: 321
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 52 -> Time(19) 58 ->
Time(31) 57 -> Time(37) 77 -> Time(58) 38 -> Time(68) 33 -> Time(84)
34 -> Time(100) 32 -> Time(108) 31 -> Time(115) 30 -> Time(125) 29 ->
Time(144) 28 -> Time(152) 51 -> Time(168) 50 -> Time(178) 22 ->
Time(189) 7 -> Time(197) 2 -> Time(213) 54 -> Time(226) 21 ->
Time(234) 5 -> Time(248) 8 -> Time(255) 4 -> Time(266) 1 -> Time(277)
3 -> Time(295) 0 -> Time(313)
  Route Distance: 361
  Route End Time: 313
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 19 -> Time(22) 18 ->
Time(36) 16 -> Time(45) 76 -> Time(60) 26 -> Time(71) 56 -> Time(93)
74 -> Time(114) 72 -> Time(133) 71 -> Time(147) 68 -> Time(155) 69 ->
Time(161) 66 -> Time(171) 65 -> Time(196) 55 -> Time(207) 53 ->
Time(218) 6 -> Time(232) 9 -> Time(242) 11 -> Time(249) 10 ->
Time(255) 12 -> Time(264) 13 -> Time(277) 14 -> Time(283) 15 ->
Time(292) 20 -> Time(300) 23 -> Time(310) 17 -> Time(322) 24 ->
Time(334) 25 -> Time(344) 27 -> Time(354) 0 -> Time(386)
  Route Distance: 418
  Route End Time: 386

Total distance of all used routes: 1156
Max route end time: 386
Total travel time cost: 556 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 7:")
df_orders = pd.read_csv('orders_day7.csv')
df_orders.columns = df_orders.columns.str.strip()

if 'TIME WINDOW' not in df_orders.columns and 'TIME_WINDOW' in
df_orders.columns:
    df_orders = df_orders.rename(columns={'TIME_WINDOW': 'TIME
WINDOW'})

df_distance = pd.read_excel("distance_matrix_7.xlsx", index_col=0)
```

```
df_time = pd.read_excel('time_matrix_mostlikely_7.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 7)
solve_vrp()



Day 7:
Shapes Day 7: orders=(66, 7), dist=(67, 67), time=(67, 67)
--- Initialization Report ---
Total Weight: 1030 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 359 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1151
MIP Objective (Alpha*Vehicles + Total Time): 1151 (3 vehicles * 200 +
551)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 3 -> Time(14) 8 ->
Time(19) 2 -> Time(31) 7 -> Time(46) 1 -> Time(61) 9 -> Time(72) 55 ->
Time(99) 53 -> Time(116) 58 -> Time(136) 60 -> Time(144) 50 ->
Time(154) 62 -> Time(166) 59 -> Time(187) 57 -> Time(200) 61 ->
Time(208) 63 -> Time(223) 64 -> Time(242) 49 -> Time(253) 48 ->
Time(259) 46 -> Time(266) 66 -> Time(276) 0 -> Time(298)
  Route Distance: 370
  Route End Time: 298
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 6 -> Time(24) 41 ->
Time(39) 29 -> Time(55) 31 -> Time(78) 51 -> Time(88) 28 -> Time(96)
26 -> Time(102) 27 -> Time(107) 52 -> Time(114) 45 -> Time(129) 25 ->
Time(137) 30 -> Time(156) 35 -> Time(174) 34 -> Time(180) 44 ->
Time(195) 36 -> Time(200) 37 -> Time(207) 38 -> Time(216) 39 ->
Time(226) 43 -> Time(233) 40 -> Time(240) 42 -> Time(251) 5 ->
Time(263) 32 -> Time(275) 33 -> Time(281) 0 -> Time(320)
  Route Distance: 380
  Route End Time: 320
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 16 -> Time(19) 15 ->
Time(41) 13 -> Time(60) 12 -> Time(72) 14 -> Time(77) 65 -> Time(93)
17 -> Time(103) 18 -> Time(118) 19 -> Time(134) 11 -> Time(148) 10 ->
Time(160) 22 -> Time(170) 54 -> Time(180) 47 -> Time(190) 4 ->
Time(208) 21 -> Time(223) 23 -> Time(235) 24 -> Time(244) 56 ->
Time(261) 20 -> Time(290) 0 -> Time(325)
  Route Distance: 401
  Route End Time: 325

Total distance of all used routes: 1151
Max route end time: 325
Total travel time cost: 551 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 8:")
df_orders = pd.read_csv('orders_day8.csv')
df_orders.columns = df_orders.columns.str.strip()

if 'TIME WINDOW' not in df_orders.columns and 'TIME_WINDOW' in
df_orders.columns:
    df_orders = df_orders.rename(columns={'TIME_WINDOW': 'TIME
WINDOW'})

df_distance = pd.read_excel("distance_matrix_8.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_8.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 8)
solve_vrp()
```

```
Day 8:
Shapes Day 8: orders=(74, 7), dist=(75, 75), time=(75, 75)
--- Initialization Report ---
Total Weight: 1090 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 318 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1440
MIP Objective (Alpha*Vehicles + Total Time): 1440 (4 vehicles * 200 +
640)
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 40 -> Time(19) 25 ->
Time(36) 50 -> Time(56) 49 -> Time(70) 48 -> Time(90) 47 -> Time(104)
45 -> Time(121) 33 -> Time(132) 32 -> Time(141) 29 -> Time(149) 41 ->
Time(162) 43 -> Time(179) 42 -> Time(185) 20 -> Time(198) 19 ->
Time(206) 15 -> Time(219) 14 -> Time(229) 0 -> Time(258)
  Route Distance: 366
  Route End Time: 258
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 72 -> Time(23) 52 ->
Time(35) 56 -> Time(47) 57 -> Time(57) 54 -> Time(62) 55 -> Time(67)
58 -> Time(77) 59 -> Time(86) 60 -> Time(95) 61 -> Time(106) 62 ->
Time(112) 63 -> Time(120) 64 -> Time(127) 65 -> Time(132) 71 ->
Time(143) 68 -> Time(155) 69 -> Time(170) 46 -> Time(187) 74 ->
Time(201) 21 -> Time(216) 73 -> Time(230) 70 -> Time(244) 66 ->
Time(257) 67 -> Time(286) 0 -> Time(323)
  Route Distance: 407
  Route End Time: 323
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 23 -> Time(14) 51 ->
Time(36) 53 -> Time(50) 28 -> Time(66) 37 -> Time(80) 36 -> Time(97)
26 -> Time(104) 38 -> Time(112) 34 -> Time(127) 1 -> Time(148) 35 ->
Time(156) 39 -> Time(164) 30 -> Time(177) 27 -> Time(193) 24 ->
Time(204) 31 -> Time(212) 0 -> Time(238)
  Route Distance: 326
  Route End Time: 238
```

```
Route for vehicle 5 (Depot: 0): 0 -> Time(0) 22 -> Time(24) 13 ->
Time(33) 12 -> Time(41) 11 -> Time(52) 16 -> Time(61) 18 -> Time(69)
17 -> Time(91) 10 -> Time(103) 9 -> Time(110) 8 -> Time(126) 5 ->
Time(137) 6 -> Time(147) 7 -> Time(162) 3 -> Time(179) 2 -> Time(198)
4 -> Time(214) 44 -> Time(228) 0 -> Time(257)
  Route Distance: 341
  Route End Time: 257

Total distance of all used routes: 1440
Max route end time: 323
Total travel time cost: 640 (The minimized sum component)

SUCCESS: All nodes were served.
```

```
print("\n\nDay 9:")
df_orders = pd.read_csv('orders_day9.csv')
df_orders.columns = df_orders.columns.str.strip()

if 'TIME WINDOW' not in df_orders.columns and 'TIME_WINDOW' in
df_orders.columns:
    df_orders = df_orders.rename(columns={'TIME_WINDOW': 'TIME
WINDOW'})

df_distance = pd.read_excel("distance_matrix_9.xlsx", index_col=0)
df_time = pd.read_excel('time_matrix_mostlikely_9.xlsx', index_col=0)

check_day(df_orders, df_distance, df_time, 9)
solve_vrp()
```

```
Day 9:
Shapes Day 9: orders=(84, 7), dist=(85, 85), time=(85, 85)
--- Initialization Report ---
Total Weight: 1303 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 524 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1452
MIP Objective (Alpha*Vehicles + Total Time): 1452 (4 vehicles * 200 +
652)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 28 -> Time(30) 27 ->
Time(43) 29 -> Time(51) 25 -> Time(66) 52 -> Time(81) 30 -> Time(97)
31 -> Time(110) 26 -> Time(131) 24 -> Time(145) 21 -> Time(156) 23 ->
Time(170) 65 -> Time(189) 22 -> Time(196) 75 -> Time(212) 74 ->
Time(224) 73 -> Time(238) 72 -> Time(245) 76 -> Time(255) 81 ->
Time(269) 80 -> Time(278) 77 -> Time(288) 79 -> Time(297) 78 ->
Time(306) 0 -> Time(339)
  Route Distance: 351
  Route End Time: 339
```

```
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 61 -> Time(19) 19 ->
Time(50) 20 -> Time(60) 82 -> Time(71) 15 -> Time(79) 17 -> Time(84)
16 -> Time(93) 14 -> Time(110) 12 -> Time(126) 11 -> Time(143) 10 ->
Time(158) 13 -> Time(164) 3 -> Time(181) 8 -> Time(191) 9 -> Time(198)
18 -> Time(211) 7 -> Time(218) 36 -> Time(225) 6 -> Time(235) 5 ->
Time(244) 2 -> Time(261) 44 -> Time(269) 1 -> Time(277) 4 -> Time(290)
0 -> Time(330)
  Route Distance: 378
  Route End Time: 330
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 68 -> Time(14) 66 ->
Time(25) 67 -> Time(34) 69 -> Time(44) 47 -> Time(62) 56 -> Time(84)
55 -> Time(97) 57 -> Time(117) 50 -> Time(142) 54 -> Time(160) 53 ->
Time(174) 43 -> Time(190) 59 -> Time(208) 58 -> Time(216) 60 ->
Time(230) 63 -> Time(243) 62 -> Time(251) 64 -> Time(260) 0 ->
Time(291)
  Route Distance: 359
  Route End Time: 291
Route for vehicle 4 (Depot: 0): 0 -> Time(0) 70 -> Time(19) 71 ->
Time(30) 83 -> Time(55) 84 -> Time(62) 45 -> Time(71) 46 -> Time(85)
49 -> Time(98) 48 -> Time(108) 51 -> Time(123) 41 -> Time(147) 42 ->
Time(157) 34 -> Time(166) 32 -> Time(184) 35 -> Time(192) 33 ->
Time(200) 37 -> Time(212) 40 -> Time(223) 38 -> Time(231) 39 ->
Time(240) 0 -> Time(268)
  Route Distance: 364
  Route End Time: 268

Total distance of all used routes: 1452
Max route end time: 339
Total travel time cost: 652 (The minimized sum component)

SUCCESS: All nodes were served.
```

Appendix Code block 1.3.

Optimistic

```
!pip install ortools
```

```
Collecting ortools
  Downloading ortools-9.14.6206-cp312-cp312-
manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl.metadata (3.3 kB)
Collecting absl-py>=2.0.0 (from ortools)
  Downloading absl_py-2.3.1-py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: numpy>=1.13.3 in
/usr/local/lib/python3.12/dist-packages (from ortools) (2.0.2)
Requirement already satisfied: pandas>=2.0.0 in
/usr/local/lib/python3.12/dist-packages (from ortools) (2.2.2)
Collecting protobuf<6.32,>=6.31.1 (from ortools)
  Downloading protobuf-6.31.1-cp39-abi3-
manylinux2014_x86_64.whl.metadata (593 bytes)
Requirement already satisfied: typing-extensions>=4.12 in
/usr/local/lib/python3.12/dist-packages (from ortools) (4.15.0)
Requirement already satisfied: immutabledict>=3.0.0 in
/usr/local/lib/python3.12/dist-packages (from ortools) (4.2.2)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.12/dist-packages (from pandas>=2.0.0->ortools)
(2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.12/dist-packages (from pandas>=2.0.0->ortools)
(2025.2)
Requirement already satisfied: tzdata>=2022.7 in
/usr/local/lib/python3.12/dist-packages (from pandas>=2.0.0->ortools)
(2025.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2-
>pandas>=2.0.0->ortools) (1.17.0)
Downloading ortools-9.14.6206-cp312-cp312-
manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl (27.7 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 27.7/27.7 MB 53.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 135.8/135.8 kB 11.9 MB/s eta
0:00:00
anylinux2014_x86_64.whl (321 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 321.1/321.1 kB 23.9 MB/s eta
0:00:00
pting uninstall: protobuf
    Found existing installation: protobuf 5.29.5
    Uninstalling protobuf-5.29.5:
      Successfully uninstalled protobuf-5.29.5
  Attempting uninstall: absl-py
    Found existing installation: absl-py 1.4.0
    Uninstalling absl-py-1.4.0:
```

```
      Successfully uninstalled absl-py-1.4.0
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of
the following dependency conflicts.
grpcio-status 1.71.2 requires protobuf<6.0dev,>=5.26.1, but you have
protobuf 6.31.1 which is incompatible.
tensorflow 2.19.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!
=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3, but you have protobuf
6.31.1 which is incompatible.
google-ai-generativelanguage 0.6.15 requires protobuf!=4.21.0,!
=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but
you have protobuf 6.31.1 which is incompatible.
Successfully installed absl-py-2.3.1 ortools-9.14.6206 protobuf-6.31.1
```


```
{"id":"f31488deff974b88b2cf141279328a79","pip_warning":{"packages":
["google"]}}
```


```
import pandas as pd
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import numpy as np

# --- 1. Load DataFrames ---
df_orders = pd.read_csv('/content/orders_day9.csv')
df_distance = pd.read_csv('/content/distance_day9.csv', index_col=0)
df_time = pd.read_csv('/content/time_day9_optimistic.csv',
index_col=0)

# --- PARAMETERS BASED ON MIP FORMULATION ---
# Alpha (α): Coefficient to penalize the number of vehicles. Must be
large to prioritize
# vehicle count minimization over total travel time minimization.
ALPHA_VEHICLE_PENALTY = 200

# Physical Fleet Specifications (for sanity checks and fleet sizing)
PHYSICAL_FLEET_SPECS = {
    'weight': 2000,  # Max weight capacity of each truck
    'volume': 1000   # Max volume capacity of each truck
}

# Buffer time to ensure depot is open when vehicles return
BUFFER_RETURN_TIME = 200

# Extra vehicles to add beyond theoretical minimum
ROUTING_BUFFER = 5

#
----------------------------------------------------------------------
# --- 2. Data Structure Preparation for OR-Tools ---
#
```

```
--------------------------------------------------------------------

import math

def create_data_model_smart(orders_df, distance_df, time_df,
physical_fleet_specs={}):
    """
    Initializes VRP data dynamically based on input statistics to
ensure feasibility.

    physical_fleet_specs: dict containing 'weight_limit',
'volume_limit' of your REAL trucks.
    """
    data = {}

    # --- 1. Data Cleaning (Keep existing logic) ---
    distance_df.dropna(how='all', axis=0, inplace=True)
    distance_df.dropna(how='all', axis=1, inplace=True)
    time_df.dropna(how='all', axis=0, inplace=True)
    time_df.dropna(how='all', axis=1, inplace=True)

    data['distance_matrix'] = distance_df.values.astype(int).tolist()
    data['time_matrix'] = time_df.values.astype(int).tolist()

    # Determine the number of actual nodes from the distance matrix
    # Assuming node 0 is the depot and subsequent nodes are customers.
    num_problem_nodes = len(data['distance_matrix'])
    num_customers_from_matrices = num_problem_nodes - 1

    # Slice orders_df to match the number of customers implied by the
matrices.
    # If df_orders has more customers than the distance matrix can
support, it's truncated.
    # If df_orders has fewer customers, this will still work
correctly.
    actual_orders_df = orders_df.head(num_customers_from_matrices)

    # --- 2. Smart Time Window Parsing ---
    parsed_windows = []
    max_deadline_in_data = 0

    # Depot is always (0,0) or (0, Open_Duration)
    # Let's assume Depot is open as long as the latest customer needs.
    parsed_windows.append((0, 0)) # Initial window for the depot (node
0)

    for tw_str in actual_orders_df['TIME_WINDOW']:
        # Parse the string "(900, 1200)" -> 900, 1200
        clean_str = tw_str.strip('()')
        if ',' in clean_str:
```

```
            parts = clean_str.split(',')
            start = int(parts[0])
            end = int(parts[1])
            parsed_windows.append((start, end))
            # Track the latest time anyone needs service
            if end > max_deadline_in_data:
                max_deadline_in_data = end
        else:
            # Fallback for bad data
            parsed_windows.append((0, 10000)) # Default fallback

    # Update Depot's window to extend to the latest deadline (plus
return trip buffer)
    # This prevents the "Depot closed before driver returns" error.
    # approx time to drive back to depot after last delivery
    horizon = max_deadline_in_data + BUFFER_RETURN_TIME
    parsed_windows[0] = (0, horizon)

    data['time_windows'] = parsed_windows
    data['vehicle_max_travel_time'] = horizon # Set Horizon
dynamically

    # --- 3. Demand & Capacity Sanity Check ---
    # Ensure weights are aligned with actual_orders_df
    data['weights'] = [0] +
actual_orders_df['WEIGHT'].round().astype(int).tolist()

    # Notice: Multiplier is consistent (was 100 in your snippet, 1000
in previous. Check your data!)
    # Ensure volumes are aligned with actual_orders_df
    data['volumes'] = [0] + (actual_orders_df['VOLUME'] *
100).round().astype(int).tolist()

    # CONSTANTS (Physical limits of your trucks)
    TRUCK_W_CAP = physical_fleet_specs.get('weight', 2000)
    TRUCK_V_CAP = physical_fleet_specs.get('volume', 1000)

    # Sanity Check: Does the biggest order fit in a truck? (from
actual_orders_df)
    if len(data['weights']) > 1: # Check only if there are customers
        max_order_w = max(data['weights'][1:]) # Exclude depot's 0
weight
        if max_order_w > TRUCK_W_CAP:
            raise ValueError(f"CRITICAL ERROR: Order exists with
weight {max_order_w}, but truck limit is {TRUCK_W_CAP}.")

    # --- 4. Smart Fleet Sizing (The Lower Bound Calculation) ---
    total_weight = sum(data['weights'])
    total_volume = sum(data['volumes'])
```

```
    # Minimum trucks needed purely for capacity (Bin Packing Lower
Bound)
    min_trucks_weight = math.ceil(total_weight / TRUCK_W_CAP)
    min_trucks_volume = math.ceil(total_volume / TRUCK_V_CAP)

    theoretical_min_vehicles = max(min_trucks_weight,
min_trucks_volume)

    # Add a "Routing Buffer" (e.g., 20% or +2 trucks)
    # Vehicles can rarely be 100% full because they run out of Time or
Distance first.
    recommended_fleet_size = int(theoretical_min_vehicles * 1.2) +
ROUTING_BUFFER

    print(f"--- Initialization Report ---")
    print(f"Total Weight: {total_weight} | Max Truck W: {TRUCK_W_CAP}
-> Min Trucks: {min_trucks_weight}")
    print(f"Total Volume: {total_volume} | Max Truck V: {TRUCK_V_CAP}
-> Min Trucks: {min_trucks_volume}")
    print(f"Latest Deadline found: {max_deadline_in_data}")
    print(f"Number of nodes derived from distance matrix:
{num_problem_nodes}")
    print(f"Number of customers used from orders_df:
{len(actual_orders_df)}")
    print(f"Setting Fleet Size to: {recommended_fleet_size}
(Theoretical Min: {theoretical_min_vehicles})")

    data['num_vehicles'] = recommended_fleet_size
    data['vehicle_capacities_weight'] = [TRUCK_W_CAP] *
data['num_vehicles']
    data['vehicle_capacities_volume'] = [TRUCK_V_CAP] *
data['num_vehicles']

    # --- 5. Other Data ---
    # Ensure service times are aligned with actual_orders_df
    data['service_times'] = [0] +
actual_orders_df['SERVICE_TIME'].astype(int).tolist()
    data['depot'] = 0
    data['penalty'] = 100000 # Keep high

    return data

#
------------------------------------------------------------------------
# --- 3. Initialize Solver Model and Constraints ---
#
------------------------------------------------------------------------

def print_solution(data, manager, routing, solution):
   """Prints the solution found by the solver and returns structured
```

```
route data."""
    total_distance = 0
    total_time_cost = 0 # Cost is now based on time, not distance
    total_time = 0
    time_dimension = routing.GetDimensionOrDie('Time')

    # --- Collect served nodes for later dropped node reporting ---
    served_nodes_set = set()
    all_routes_data = [] # List to store structured route data

    # Pre-calculate total travel time for objective reporting first
    # This loop only calculates total_time_cost and served_nodes_set
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        if routing.IsEnd(solution.Value(routing.NextVar(index))):
            continue

        route_path_nodes = []
        current_route_time_cost = 0
        current_route_distance = 0

        # Add depot as the start of the route
        route_path_nodes.append(manager.IndexToNode(index))

        while not routing.IsEnd(index):
            previous_index = index
            current_node_index = manager.IndexToNode(index)
            if current_node_index != data['depot']: # Don't add depot
to served_nodes_set
                served_nodes_set.add(current_node_index)

            index = solution.Value(routing.NextVar(index))

            from_node = manager.IndexToNode(previous_index)
            to_node = manager.IndexToNode(index)
            current_route_time_cost += data['time_matrix'][from_node]
[to_node]
            current_route_distance +=
routing.GetArcCostForVehicle(previous_index, index, vehicle_id)

            # Add node to current route path, except if it's the final
depot return
            if not routing.IsEnd(index):
                route_path_nodes.append(to_node)
            else:
                route_path_nodes.append(data['depot']) # Add depot for
end of route

        total_time_cost += current_route_time_cost
        total_distance += current_route_distance
```

```
   # Calculate objective based on MIP formulation: alpha * Yk +
sum(tij * xijk)
   num_used_vehicles = len([v for v in range(data['num_vehicles']) if
not routing.IsEnd(solution.Value(routing.NextVar(routing.Start(v))))])
   vehicle_penalty_component = num_used_vehicles *
ALPHA_VEHICLE_PENALTY

   mip_objective = vehicle_penalty_component + total_time_cost

   print(f'OR-Tools Objective (Time Cost + Penalties):
{solution.ObjectiveValue()}')
   print(f'MIP Objective (Alpha*Vehicles + Total Time):
{mip_objective} ({num_used_vehicles} vehicles *
{ALPHA_VEHICLE_PENALTY} + {total_time_cost})')

   # Print routes and collect data
   for vehicle_id in range(data['num_vehicles']):
       index = routing.Start(vehicle_id)
       if routing.IsEnd(solution.Value(routing.NextVar(index))):
           continue

       plan_output = f'Route for vehicle {vehicle_id} (Depot: 0):'
       route_distance = 0
       route_nodes_for_plot = []

       node_index_at_start =
manager.IndexToNode(routing.Start(vehicle_id))
       route_nodes_for_plot.append(node_index_at_start)
       plan_output += f' {node_index_at_start} ->
Time({solution.Min(time_dimension.CumulVar(index))})'

       while not routing.IsEnd(index):
           previous_index = index
           index = solution.Value(routing.NextVar(index))

           route_distance += routing.GetArcCostForVehicle(
               previous_index, index, vehicle_id
           )
           node_index = manager.IndexToNode(index)
           route_nodes_for_plot.append(node_index)
           time_var = time_dimension.CumulVar(index)
           plan_output += (
               f' {node_index} -> Time({solution.Min(time_var)})'
           )

       route_end_time = solution.Min(time_dimension.CumulVar(index))
       plan_output += f'\n  Route Distance: {route_distance}'
       plan_output += f'\n  Route End Time: {route_end_time}'
```

```
        print(plan_output)
        total_time = max(total_time, route_end_time)

        all_routes_data.append({
            'vehicle_id': vehicle_id,
            'route_nodes': route_nodes_for_plot,
            'route_distance': route_distance,
            'route_end_time': route_end_time
        })

    print(f'\nTotal distance of all used routes: {total_distance}')
    print(f'Max route end time: {total_time}')
    print(f'Total travel time cost: {total_time_cost} (The minimized
sum component)')

    # --- Report Dropped Nodes (new approach) ---
    # All nodes from 1 to num_problem_nodes-1 are customer nodes
    all_customer_nodes = set(range(1, manager.GetNumberOfNodes()))
    dropped_nodes = list(all_customer_nodes - served_nodes_set)
    dropped_nodes.sort() # For consistent output

    if dropped_nodes:
        print(f'\n!!! WARNING: {len(dropped_nodes)} Nodes Were DROPPED
(Unserved) !!!')
        print(f'Dropped Nodes (NODE_ID): {dropped_nodes}')
    else:
        print('\nSUCCESS: All nodes were served.')

    return all_routes_data, data


def solve_vrp():
    """Entry point for the VRP solver with MIP objective."""
    data = create_data_model_smart(df_orders, df_distance, df_time)

    manager = pywrapcp.RoutingIndexManager(
        len(data['distance_matrix']), data['num_vehicles'],
data['depot']
    )

    routing = pywrapcp.RoutingModel(manager)

    # --- A. Define Cost (Time - as per MIP Objective) ---
    def time_cost_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        # The cost minimized is the travel time (t_ij)
        return data['time_matrix'][from_node][to_node]

    transit_callback_index =
```

```
routing.RegisterTransitCallback(time_cost_callback)
   routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

   # A.1. Set Objective to Minimize Time Cost (t_ij) AND Penalize
Vehicles (alpha * y_k)

   # Fixed cost (alpha * y_k): Apply a large penalty for using each
vehicle
   for vehicle_id in range(data['num_vehicles']):
       routing.SetFixedCostOfVehicle(ALPHA_VEHICLE_PENALTY,
vehicle_id)

   # A.2. Add Dropped Node Penalty
   # Iterate over customer nodes, not including the depot (node 0).
   # The range should be from 1 to num_problem_nodes - 1 (inclusive of
the last customer).
   for node in range(1, len(data['distance_matrix'])):
       routing.AddDisjunction([manager.NodeToIndex(node)],
data['penalty'])

   # --- B. Add Capacity Constraints (Weight and Volume) ---

   def add_capacity_dimension(capacity_name, capacities, demands):
       def demand_callback(index):
           node = manager.IndexToNode(index)
           return demands[node]

       demand_callback_index =
routing.RegisterUnaryTransitCallback(demand_callback)

       # Capacity bounds are imposed at every node [cite: 87]
       routing.AddDimensionWithVehicleCapacity(
           demand_callback_index,
           0, # Slack
           capacities,
           True, # This dimension is cumulative
           capacity_name
       )

   add_capacity_dimension('WeightCapacity',
data['vehicle_capacities_weight'], data['weights'])
   add_capacity_dimension('VolumeCapacity',
data['vehicle_capacities_volume'], data['volumes'])


   # --- C. Add Time Window Constraints ---

   def time_callback(from_index, to_index):
       from_node = manager.IndexToNode(from_index)
       to_node = manager.IndexToNode(to_index)
```

```
        travel_time = data['time_matrix'][from_node][to_node]
        service_time = data['service_times'][from_node]
        # Time propagation: T_jk >= T_ik + s_i + t_ij [cite: 99]
        return travel_time + service_time

    time_callback_index =
routing.RegisterTransitCallback(time_callback)

    # Time dimension: tracks T_ik (time vehicle k starts service at
node i) [cite: 45]
    routing.AddDimension(
        time_callback_index,
        0,                                # slack_max (0 is fine here)
        data['vehicle_max_travel_time'], # Planning horizon H [cite:
28]
        False,
        'Time'
    )
    time_dimension = routing.GetDimensionOrDie('Time')

    # Apply Time Windows [cite: 94]
    # The loop should iterate over the actual number of nodes in the
problem
    for node in range(len(data['distance_matrix'])):
        index = manager.NodeToIndex(node)
        start, end = data['time_windows'][node]
        time_dimension.CumulVar(index).SetRange(start, end)

    # Apply Route Duration Constraint (limited by H) [cite: 104]
    for i in range(data['num_vehicles']):
        time_dimension.SetSpanUpperBoundForVehicle(
            data['vehicle_max_travel_time'],
            i
        )

    # --- D. Set Search Parameters and Solve ---
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
    )
    search_parameters.time_limit.seconds = 120

    # Solve the problem
    solution = routing.SolveWithParameters(search_parameters)

    # --- E. Print Solution ---
    if solution:
```

```
        return print_solution(data, manager, routing, solution)
    else:
        print('No solution found!')
        return [], data # Return empty list and data object in case of
no solution


#
# ---------------------------------------------------------------------
# --- EXECUTION --- (Moved to the visualization cell)
#
# ---------------------------------------------------------------------

import matplotlib.pyplot as plt
import random

# Call the solver to get the routes and data model
all_routes_data, data = solve_vrp() # Changed to solve_vrp()
num_nodes = len(data['distance_matrix'])

# --- Generate Dummy Coordinates for Visualization ---
# For a better visualization, we'll create random coordinates for each
node.
# Node 0 (depot) will be at the center (0,0).
# Other nodes will be randomly placed.

coordinates = [(random.uniform(-50, 50), random.uniform(-50, 50)) for
_ in range(num_nodes)]
coordinates[data['depot']] = (0, 0) # Set depot at origin

# --- Plotting the Routes ---
plt.figure(figsize=(12, 10))

# Plot all nodes first
node_x = [coord[0] for coord in coordinates]
node_y = [coord[1] for coord in coordinates]
plt.scatter(node_x, node_y, color='black', s=50, zorder=5,
label='Customers')

# Highlight depot
depot_x, depot_y = coordinates[data['depot']]
plt.scatter(depot_x, depot_y, color='red', s=150, marker='s',
zorder=6, label='Depot')
plt.text(depot_x, depot_y, 'Depot', fontsize=9, ha='right',
va='bottom', color='white', weight='bold')

# Add node labels
for i, (x, y) in enumerate(coordinates):
    if i != data['depot']:
        plt.text(x, y, str(i), fontsize=8, ha='right', va='bottom')
```

```
# Define a color palette for vehicles
colors = plt.cm.get_cmap('tab10', len(all_routes_data))

for idx, route_info in enumerate(all_routes_data):
    vehicle_id = route_info['vehicle_id']
    route_nodes = route_info['route_nodes']
    color = colors(idx)

    # Plot each segment of the route
    for i in range(len(route_nodes) - 1):
        start_node_idx = route_nodes[i]
        end_node_idx = route_nodes[i+1]

        start_x, start_y = coordinates[start_node_idx]
        end_x, end_y = coordinates[end_node_idx]

        plt.plot([start_x, end_x], [start_y, end_y], color=color,
linewidth=2, alpha=0.7,
                 label=f'Vehicle {vehicle_id}' if i == 0 else "") #
Label only once per vehicle
        plt.arrow(start_x, start_y, end_x - start_x, end_y - start_y,
                  head_width=1.5, head_length=2.5, fc=color, ec=color,
length_includes_head=True, alpha=0.7)

plt.title('Vehicle Routes (Dummy Coordinates)')
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

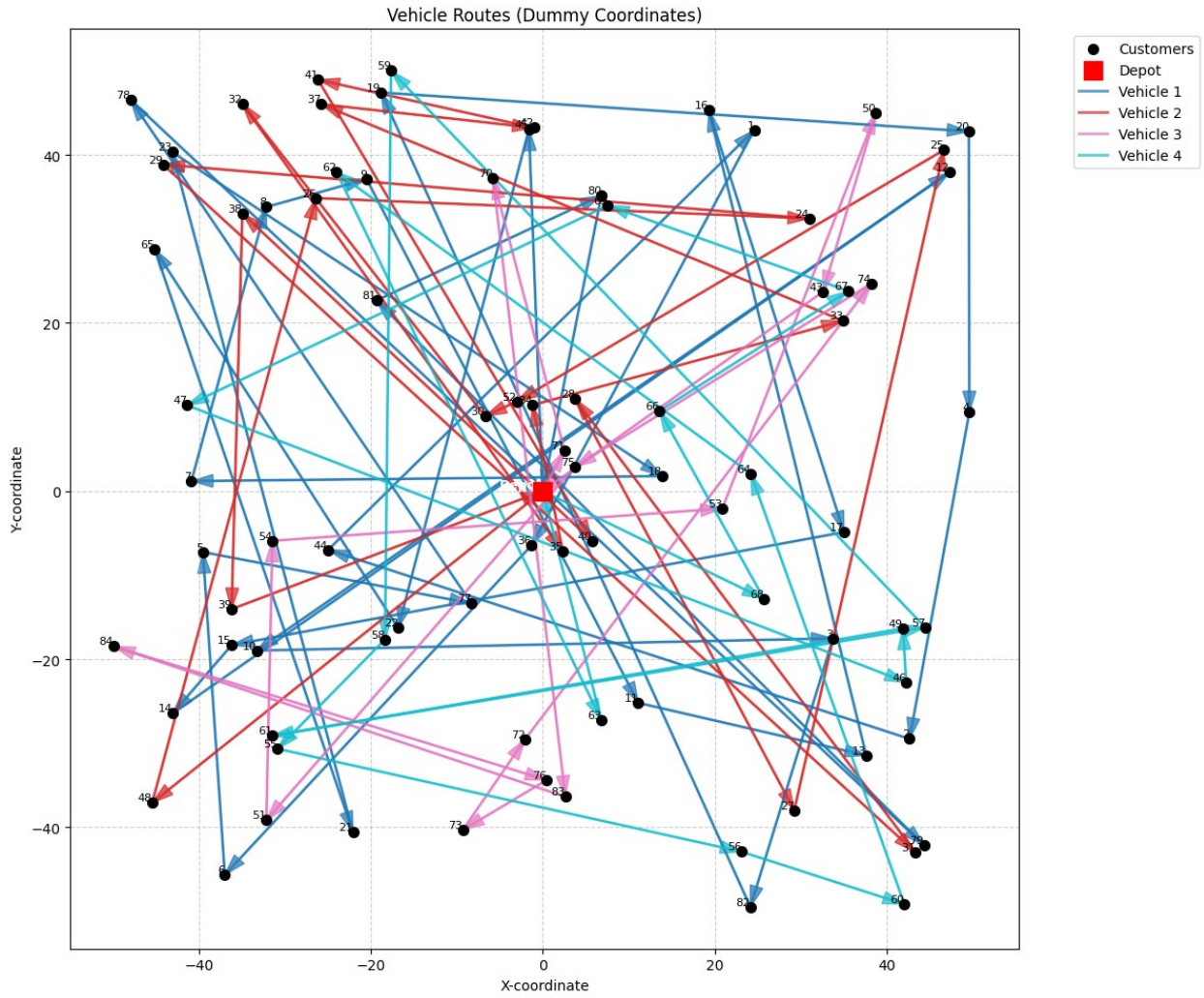
```
--- Initialization Report ---
Total Weight: 1303 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 524 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Number of nodes derived from distance matrix: 85
Number of customers used from orders_df: 84
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1321
MIP Objective (Alpha*Vehicles + Total Time): 1321 (4 vehicles * 200 +
521)
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 45 -> Time(18) 22 ->
Time(28) 65 -> Time(35) 21 -> Time(45) 23 -> Time(57) 18 -> Time(74) 7
-> Time(79) 8 -> Time(85) 9 -> Time(91) 11 -> Time(102) 13 ->
Time(116) 16 -> Time(133) 17 -> Time(147) 15 -> Time(153) 14 ->
Time(161) 12 -> Time(176) 10 -> Time(192) 3 -> Time(200) 82 ->
Time(207) 19 -> Time(219) 20 -> Time(228) 4 -> Time(243) 2 ->
Time(260) 44 -> Time(267) 1 -> Time(274) 36 -> Time(280) 6 ->
```

```
Time(288) 5 -> Time(297) 77 -> Time(311) 78 -> Time(320) 79 ->
Time(329) 81 -> Time(340) 80 -> Time(349) 0 -> Time(379)
  Route Distance: 371
  Route End Time: 379
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 48 -> Time(17) 26 ->
Time(35) 24 -> Time(49) 29 -> Time(58) 31 -> Time(71) 28 -> Time(85)
27 -> Time(98) 25 -> Time(105) 52 -> Time(119) 30 -> Time(134) 32 ->
Time(149) 35 -> Time(156) 34 -> Time(165) 33 -> Time(180) 37 ->
Time(190) 42 -> Time(202) 41 -> Time(212) 40 -> Time(220) 38 ->
Time(228) 39 -> Time(236) 0 -> Time(260)
  Route Distance: 316
  Route End Time: 260
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 71 -> Time(20) 70 ->
Time(38) 83 -> Time(50) 84 -> Time(56) 76 -> Time(68) 73 -> Time(82)
72 -> Time(89) 74 -> Time(96) 75 -> Time(107) 51 -> Time(131) 54 ->
Time(152) 53 -> Time(165) 50 -> Time(179) 43 -> Time(196) 0 ->
Time(219)
  Route Distance: 311
  Route End Time: 219
Route for vehicle 4 (Depot: 0): 0 -> Time(0) 68 -> Time(13) 66 ->
Time(24) 67 -> Time(33) 69 -> Time(42) 47 -> Time(59) 46 -> Time(79)
49 -> Time(92) 61 -> Time(103) 57 -> Time(122) 59 -> Time(141) 58 ->
Time(149) 55 -> Time(156) 56 -> Time(169) 60 -> Time(186) 64 ->
Time(198) 62 -> Time(206) 63 -> Time(213) 0 -> Time(239)
  Route Distance: 323
  Route End Time: 239

Total distance of all used routes: 1321
Max route end time: 379
Total travel time cost: 521 (The minimized sum component)

SUCCESS: All nodes were served.

/tmp/ipython-input-813097466.py:35: MatplotlibDeprecationWarning: The
get_cmap function was deprecated in Matplotlib 3.7 and will be removed
in 3.11. Use ``matplotlib.colormaps[name]`` or
``matplotlib.colormaps.get_cmap()`` or ``pyplot.get_cmap()`` instead.
  colors = plt.cm.get_cmap('tab10', len(all_routes_data))
```



Optimistic Solver (For Visualization - Original `solve_vrp` untouched)

```
import pandas as pd
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import numpy as np

# --- 1. Load DataFrames ---
# Using the same dataframes as the optimistic scenario for this copy
df_orders_viz = pd.read_csv('/content/orders_day9.csv')
df_distance_viz = pd.read_csv('/content/distance_day9.csv',
index_col=0)
df_time_viz = pd.read_csv('/content/time_day9_optimistic.csv',
index_col=0)

# --- PARAMETERS BASED ON MIP FORMULATION ---
ALPHA_VEHICLE_PENALTY_VIZ = 200
PHYSICAL_FLEET_SPECS_VIZ = {
    'weight': 2000,
```

```
    'volume': 1000
}
BUFFER_RETURN_TIME_VIZ = 200
ROUTING_BUFFER_VIZ = 5

# Re-define create_data_model_smart to use the _viz dataframes
import math

def create_data_model_smart_viz(orders_df, distance_df, time_df,
physical_fleet_specs={}):
    data = {}
    distance_df.dropna(how='all', axis=0, inplace=True)
    distance_df.dropna(how='all', axis=1, inplace=True)
    time_df.dropna(how='all', axis=0, inplace=True)
    time_df.dropna(how='all', axis=1, inplace=True)

    data['distance_matrix'] = distance_df.values.astype(int).tolist()
    data['time_matrix'] = time_df.values.astype(int).tolist()

    num_problem_nodes = len(data['distance_matrix'])
    num_customers_from_matrices = num_problem_nodes - 1
    actual_orders_df = orders_df.head(num_customers_from_matrices)

    parsed_windows = []
    max_deadline_in_data = 0
    parsed_windows.append((0, 0))

    for tw_str in actual_orders_df['TIME_WINDOW']:
        clean_str = tw_str.strip('()')
        if ',' in clean_str:
            parts = clean_str.split(',')
            start = int(parts[0])
            end = int(parts[1])
            parsed_windows.append((start, end))
            if end > max_deadline_in_data:
                max_deadline_in_data = end
        else:
            parsed_windows.append((0, 10000))

    horizon = max_deadline_in_data + BUFFER_RETURN_TIME_VIZ
    parsed_windows[0] = (0, horizon)

    data['time_windows'] = parsed_windows
    data['vehicle_max_travel_time'] = horizon

    data['weights'] = [0] +
actual_orders_df['WEIGHT'].round().astype(int).tolist()
    data['volumes'] = [0] + (actual_orders_df['VOLUME'] *
100).round().astype(int).tolist()
```

```
    TRUCK_W_CAP = physical_fleet_specs.get('weight', 2000)
    TRUCK_V_CAP = physical_fleet_specs.get('volume', 1000)

    if len(data['weights']) > 1:
        max_order_w = max(data['weights'][1:])
        if max_order_w > TRUCK_W_CAP:
            raise ValueError(f"CRITICAL ERROR: Order exists with
weight {max_order_w}, but truck limit is {TRUCK_W_CAP}.")

    total_weight = sum(data['weights'])
    total_volume = sum(data['volumes'])

    min_trucks_weight = math.ceil(total_weight / TRUCK_W_CAP)
    min_trucks_volume = math.ceil(total_volume / TRUCK_V_CAP)
    theoretical_min_vehicles = max(min_trucks_weight,
min_trucks_volume)

    recommended_fleet_size = int(theoretical_min_vehicles * 1.2) +
ROUTING_BUFFER_VIZ

    print(f"--- Initialization Report (for Visualization) ---")
    print(f"Total Weight: {total_weight} | Max Truck W: {TRUCK_W_CAP}
-> Min Trucks: {min_trucks_weight}")
    print(f"Total Volume: {total_volume} | Max Truck V: {TRUCK_V_CAP}
-> Min Trucks: {min_trucks_volume}")
    print(f"Latest Deadline found: {max_deadline_in_data}")
    print(f"Number of nodes derived from distance matrix:
{num_problem_nodes}")
    print(f"Number of customers used from orders_df:
{len(actual_orders_df)}")
    print(f"Setting Fleet Size to: {recommended_fleet_size}
(Theoretical Min: {theoretical_min_vehicles})")

    data['num_vehicles'] = recommended_fleet_size
    data['vehicle_capacities_weight'] = [TRUCK_W_CAP] *
data['num_vehicles']
    data['vehicle_capacities_volume'] = [TRUCK_V_CAP] *
data['num_vehicles']

    data['service_times'] = [0] +
actual_orders_df['SERVICE_TIME'].astype(int).tolist()
    data['depot'] = 0
    data['penalty'] = 100000

    return data

def print_solution_viz(data, manager, routing, solution):
   """Prints the solution found by the solver and returns structured
route data."""
   total_distance = 0
```

```
    total_time_cost = 0
    total_time = 0
    time_dimension = routing.GetDimensionOrDie('Time')

    served_nodes_set = set()
    all_routes_data = []

    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        if routing.IsEnd(solution.Value(routing.NextVar(index))):
            continue

        route_path_nodes = []
        current_route_time_cost = 0
        current_route_distance = 0

        route_path_nodes.append(manager.IndexToNode(index))

        while not routing.IsEnd(index):
            previous_index = index
            current_node_index = manager.IndexToNode(index)
            if current_node_index != data['depot']:
                served_nodes_set.add(current_node_index)

            index = solution.Value(routing.NextVar(index))

            from_node = manager.IndexToNode(previous_index)
            to_node = manager.IndexToNode(index)
            current_route_time_cost += data['time_matrix'][from_node]
[to_node]
            current_route_distance +=
routing.GetArcCostForVehicle(previous_index, index, vehicle_id)

            if not routing.IsEnd(index):
                route_path_nodes.append(to_node)
            else:
                route_path_nodes.append(data['depot'])

        total_time_cost += current_route_time_cost
        total_distance += current_route_distance

    num_used_vehicles = len([v for v in range(data['num_vehicles']) if
not routing.IsEnd(solution.Value(routing.NextVar(routing.Start(v))))])
    vehicle_penalty_component = num_used_vehicles *
ALPHA_VEHICLE_PENALTY_VIZ

    mip_objective = vehicle_penalty_component + total_time_cost

    print(f'OR-Tools Objective (Time Cost + Penalties):
{solution.ObjectiveValue()}')
```

```
    print(f'MIP Objective (Alpha*Vehicles + Total Time):
{mip_objective} ({num_used_vehicles} vehicles *
{ALPHA_VEHICLE_PENALTY_VIZ} + {total_time_cost})')

    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        if routing.IsEnd(solution.Value(routing.NextVar(index))):
            continue

        plan_output = f'Route for vehicle {vehicle_id} (Depot: 0):'
        route_distance = 0
        route_nodes_for_plot = []

        node_index_at_start =
manager.IndexToNode(routing.Start(vehicle_id))
        route_nodes_for_plot.append(node_index_at_start)
        plan_output += f' {node_index_at_start} ->
Time({solution.Min(time_dimension.CumulVar(index))})'

        while not routing.IsEnd(index):
            previous_index = index
            index = solution.Value(routing.NextVar(index))

            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id
            )
            node_index = manager.IndexToNode(index)
            route_nodes_for_plot.append(node_index)
            time_var = time_dimension.CumulVar(index)
            plan_output += (
                f' {node_index} -> Time({solution.Min(time_var)})'
            )

        route_end_time = solution.Min(time_dimension.CumulVar(index))
        plan_output += f'\n  Route Distance: {route_distance}'
        plan_output += f'\n  Route End Time: {route_end_time}'

        print(plan_output)
        total_time = max(total_time, route_end_time)

        all_routes_data.append({
            'vehicle_id': vehicle_id,
            'route_nodes': route_nodes_for_plot,
            'route_distance': route_distance,
            'route_end_time': route_end_time
        })

    print(f'\nTotal distance of all used routes: {total_distance}')
    print(f'Max route end time: {total_time}')
    print(f'Total travel time cost: {total_time_cost} (The minimized
```

```
sum component)')

    all_customer_nodes = set(range(1, manager.GetNumberOfNodes()))
    dropped_nodes = list(all_customer_nodes - served_nodes_set)
    dropped_nodes.sort()

    if dropped_nodes:
        print(f'\n!!! WARNING: {len(dropped_nodes)} Nodes Were DROPPED
(Unserved) !!!')
        print(f'Dropped Nodes (NODE_ID): {dropped_nodes}')
    else:
        print('\nSUCCESS: All nodes were served.')

    return all_routes_data, data

def solve_vrp_for_visualization():
    """Entry point for the VRP solver with MIP objective, returning
data for visualization."""
    data = create_data_model_smart_viz(df_orders_viz, df_distance_viz,
df_time_viz, PHYSICAL_FLEET_SPECS_VIZ)

    manager = pywrapcp.RoutingIndexManager(
        len(data['distance_matrix']), data['num_vehicles'],
data['depot']
    )

    routing = pywrapcp.RoutingModel(manager)

    def time_cost_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return data['time_matrix'][from_node][to_node]

    transit_callback_index =
routing.RegisterTransitCallback(time_cost_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    for vehicle_id in range(data['num_vehicles']):
        routing.SetFixedCostOfVehicle(ALPHA_VEHICLE_PENALTY_VIZ,
vehicle_id)

    for node in range(1, len(data['distance_matrix'])):
        routing.AddDisjunction([manager.NodeToIndex(node)],
data['penalty'])

    def add_capacity_dimension(capacity_name, capacities, demands):
        def demand_callback(index):
            node = manager.IndexToNode(index)
            return demands[node]
```

```
        demand_callback_index =
routing.RegisterUnaryTransitCallback(demand_callback)

        routing.AddDimensionWithVehicleCapacity(
            demand_callback_index,
            0,
            capacities,
            True,
            capacity_name
        )

    add_capacity_dimension('WeightCapacity',
data['vehicle_capacities_weight'], data['weights'])
    add_capacity_dimension('VolumeCapacity',
data['vehicle_capacities_volume'], data['volumes'])

    def time_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        travel_time = data['time_matrix'][from_node][to_node]
        service_time = data['service_times'][from_node]
        return travel_time + service_time

    time_callback_index =
routing.RegisterTransitCallback(time_callback)

    routing.AddDimension(
        time_callback_index,
        0,
        data['vehicle_max_travel_time'],
        False,
        'Time'
    )
    time_dimension = routing.GetDimensionOrDie('Time')

    for node in range(len(data['distance_matrix'])):
        index = manager.NodeToIndex(node)
        start, end = data['time_windows'][node]
        time_dimension.CumulVar(index).SetRange(start, end)

    for i in range(data['num_vehicles']):
        time_dimension.SetSpanUpperBoundForVehicle(
            data['vehicle_max_travel_time'],
            i
        )

    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
```

```
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
    )
    search_parameters.time_limit.seconds = 120

    solution = routing.SolveWithParameters(search_parameters)

    if solution:
        return print_solution_viz(data, manager, routing, solution)
    else:
        print('No solution found!')
        return [], data
```

Visualization of Optimistic Scenario Routes

```
import matplotlib.pyplot as plt
import random

# Call the new solver function to get the routes and data model for
visualization
all_routes_data_optimistic, data_optimistic =
solve_vrp_for_visualization()
num_nodes_optimistic = len(data_optimistic['distance_matrix'])

# --- Generate Dummy Coordinates for Visualization ---
coordinates_optimistic = [(random.uniform(-50, 50), random.uniform(-
50, 50)) for _ in range(num_nodes_optimistic)]
coordinates_optimistic[data_optimistic['depot']] = (0, 0) # Set depot
at origin

# --- Plotting the Routes ---
plt.figure(figsize=(12, 10))

# Plot all nodes first
node_x_optimistic = [coord[0] for coord in coordinates_optimistic]
node_y_optimistic = [coord[1] for coord in coordinates_optimistic]
plt.scatter(node_x_optimistic, node_y_optimistic, color='black', s=50,
zorder=5, label='Customers')

# Highlight depot
depot_x_optimistic, depot_y_optimistic =
coordinates_optimistic[data_optimistic['depot']]
plt.scatter(depot_x_optimistic, depot_y_optimistic, color='red',
s=150, marker='s', zorder=6, label='Depot')
plt.text(depot_x_optimistic, depot_y_optimistic, 'Depot', fontsize=9,
ha='right', va='bottom', color='white', weight='bold')

# Add node labels
for i, (x, y) in enumerate(coordinates_optimistic):
    if i != data_optimistic['depot']:
```

```
        plt.text(x, y, str(i), fontsize=8, ha='right', va='bottom')

# Define a color palette for vehicles
colors = plt.cm.get_cmap('tab10', len(all_routes_data_optimistic))

for idx, route_info in enumerate(all_routes_data_optimistic):
    vehicle_id = route_info['vehicle_id']
    route_nodes = route_info['route_nodes']
    color = colors(idx)

    # Plot each segment of the route
    for i in range(len(route_nodes) - 1):
        start_node_idx = route_nodes[i]
        end_node_idx = route_nodes[i+1]

        start_x, start_y = coordinates_optimistic[start_node_idx]
        end_x, end_y = coordinates_optimistic[end_node_idx]

        plt.plot([start_x, end_x], [start_y, end_y], color=color,
linewidth=2, alpha=0.7,
                 label=f'Vehicle {vehicle_id}' if i == 0 else "") #
Label only once per vehicle
        plt.arrow(start_x, start_y, end_x - start_x, end_y - start_y,
                  head_width=1.5, head_length=2.5, fc=color, ec=color,
length_includes_head=True, alpha=0.7)

plt.title('Vehicle Routes (Optimistic Scenario - Dummy Coordinates)')
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

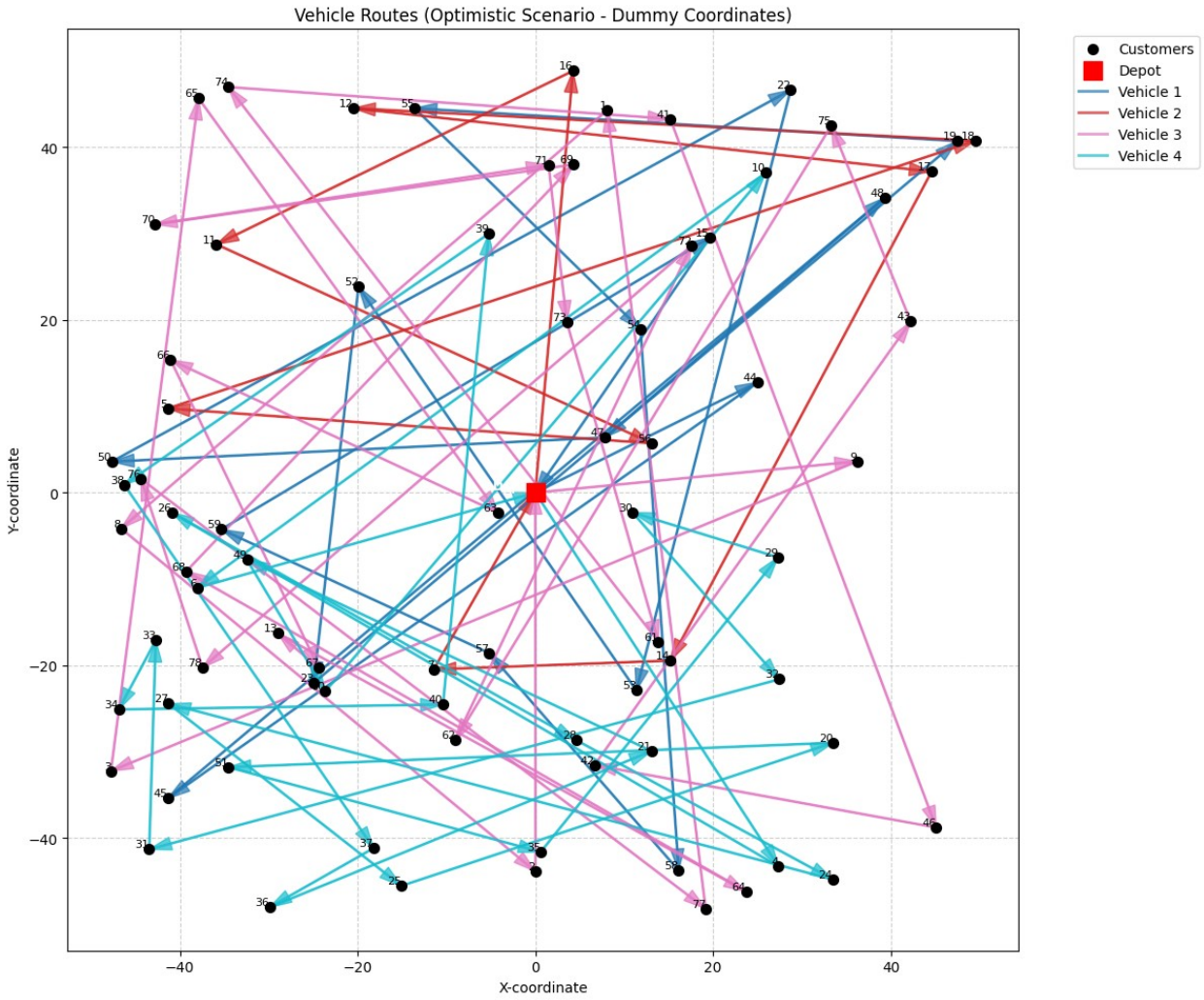
```
--- Initialization Report (for Visualization) ---
Total Weight: 1155 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 446 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Number of nodes derived from distance matrix: 79
Number of customers used from orders_df: 78
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1250
MIP Objective (Alpha*Vehicles + Total Time): 1250 (4 vehicles * 200 +
450)
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 44 -> Time(26) 45 ->
Time(34) 48 -> Time(42) 47 -> Time(47) 50 -> Time(63) 22 -> Time(77)
53 -> Time(85) 52 -> Time(99) 23 -> Time(112) 19 -> Time(127) 55 ->
Time(134) 54 -> Time(155) 58 -> Time(173) 57 -> Time(179) 59 ->
Time(194) 15 -> Time(201) 0 -> Time(223)
  Route Distance: 295
```

```
  Route End Time: 223
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 16 -> Time(18) 11 ->
Time(32) 56 -> Time(52) 5 -> Time(68) 18 -> Time(88) 12 -> Time(102)
17 -> Time(116) 14 -> Time(128) 7 -> Time(156) 0 -> Time(180)
  Route Distance: 300
  Route End Time: 180
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 9 -> Time(16) 3 ->
Time(22) 65 -> Time(30) 63 -> Time(35) 66 -> Time(40) 67 -> Time(45)
13 -> Time(55) 64 -> Time(71) 68 -> Time(78) 69 -> Time(90) 70 ->
Time(99) 71 -> Time(105) 73 -> Time(122) 61 -> Time(127) 74 ->
Time(140) 41 -> Time(154) 46 -> Time(162) 42 -> Time(173) 43 ->
Time(178) 75 -> Time(184) 62 -> Time(193) 72 -> Time(201) 78 ->
Time(208) 76 -> Time(219) 77 -> Time(228) 1 -> Time(242) 8 ->
Time(247) 2 -> Time(256) 0 -> Time(276)
  Route Distance: 312
  Route End Time: 276
Route for vehicle 4 (Depot: 0): 0 -> Time(0) 4 -> Time(14) 26 ->
Time(22) 28 -> Time(42) 24 -> Time(57) 27 -> Time(64) 25 -> Time(71)
20 -> Time(85) 51 -> Time(93) 35 -> Time(106) 29 -> Time(113) 30 ->
Time(126) 32 -> Time(140) 31 -> Time(145) 33 -> Time(163) 34 ->
Time(173) 40 -> Time(193) 39 -> Time(199) 38 -> Time(205) 37 ->
Time(211) 36 -> Time(219) 21 -> Time(231) 49 -> Time(238) 60 ->
Time(249) 10 -> Time(263) 6 -> Time(283) 0 -> Time(311)
  Route Distance: 343
  Route End Time: 311

Total distance of all used routes: 1250
Max route end time: 311
Total travel time cost: 450 (The minimized sum component)

SUCCESS: All nodes were served.

/tmp/ipython-input-1627322346.py:31: MatplotlibDeprecationWarning: The
get_cmap function was deprecated in Matplotlib 3.7 and will be removed
in 3.11. Use ``matplotlib.colormaps[name]`` or
``matplotlib.colormaps.get_cmap()`` or ``pyplot.get_cmap()`` instead.
  colors = plt.cm.get_cmap('tab10', len(all_routes_data_optimistic))
```



Visualization of MPSolver (MIP) Scenario Routes

Pessimistic

```
import pandas as pd
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import numpy as np

# --- 1. Load DataFrames ---
df_orders = pd.read_csv('/content/orders_day9.csv')
df_distance = pd.read_csv('/content/distance_day9.csv', index_col=0)
df_time = pd.read_csv('/content/time_day9_pessimistic.csv',
index_col=0)

# --- PARAMETERS BASED ON MIP FORMULATION ---
# Alpha (α): Coefficient to penalize the number of vehicles. Must be
```

```
large to prioritize
# vehicle count minimization over total travel time minimization.
ALPHA_VEHICLE_PENALTY = 200

# Physical Fleet Specifications (for sanity checks and fleet sizing)
PHYSICAL_FLEET_SPECS = {
    'weight': 2000,  # Max weight capacity of each truck
    'volume': 1000   # Max volume capacity of each truck
}

# Buffer time to ensure depot is open when vehicles return
BUFFER_RETURN_TIME = 200

# Extra vehicles to add beyond theoretical minimum
ROUTING_BUFFER = 5

#
--------------------------------------------------------------------
# --- 2. Data Structure Preparation for OR-Tools ---
#
--------------------------------------------------------------------

import math

def create_data_model_smart(orders_df, distance_df, time_df,
physical_fleet_specs={}):
    """
    Initializes VRP data dynamically based on input statistics to
ensure feasibility.

    physical_fleet_specs: dict containing 'weight_limit',
'volume_limit' of your REAL trucks.
    """
    data = {}

    # --- 1. Data Cleaning (Keep existing logic) ---
    distance_df.dropna(how='all', axis=0, inplace=True)
    distance_df.dropna(how='all', axis=1, inplace=True)
    time_df.dropna(how='all', axis=0, inplace=True)
    time_df.dropna(how='all', axis=1, inplace=True)

    data['distance_matrix'] = distance_df.values.astype(int).tolist()
    data['time_matrix'] = time_df.values.astype(int).tolist()

    # Determine the number of actual nodes from the distance matrix
    # Assuming node 0 is the depot and subsequent nodes are customers.
    num_problem_nodes = len(data['distance_matrix'])
    num_customers_from_matrices = num_problem_nodes - 1

    # Slice orders_df to match the number of customers implied by the
```

```
matrices.
    # If df_orders has more customers than the distance matrix can
support, it's truncated.
    # If df_orders has fewer customers, this will still work
correctly.
    actual_orders_df = orders_df.head(num_customers_from_matrices)

    # --- 2. Smart Time Window Parsing ---
    parsed_windows = []
    max_deadline_in_data = 0

    # Depot is always (0,0) or (0, Open_Duration)
    # Let's assume Depot is open as long as the latest customer needs.
    parsed_windows.append((0, 0)) # Initial window for the depot (node
0)

    for tw_str in actual_orders_df['TIME_WINDOW']:
        # Parse the string "(900, 1200)" -> 900, 1200
        clean_str = tw_str.strip('()')
        if ',' in clean_str:
            parts = clean_str.split(',')
            start = int(parts[0])
            end = int(parts[1])
            parsed_windows.append((start, end))
            # Track the latest time anyone needs service
            if end > max_deadline_in_data:
                max_deadline_in_data = end
        else:
            # Fallback for bad data
            parsed_windows.append((0, 10000)) # Default fallback

    # Update Depot's window to extend to the latest deadline (plus
return trip buffer)
    # This prevents the "Depot closed before driver returns" error.
    # approx time to drive back to depot after last delivery
    horizon = max_deadline_in_data + BUFFER_RETURN_TIME
    parsed_windows[0] = (0, horizon)

    data['time_windows'] = parsed_windows
    data['vehicle_max_travel_time'] = horizon # Set Horizon
dynamically

    # --- 3. Demand & Capacity Sanity Check ---
    # Ensure weights are aligned with actual_orders_df
    data['weights'] = [0] +
actual_orders_df['WEIGHT'].round().astype(int).tolist()

    # Notice: Multiplier is consistent (was 100 in your snippet, 1000
in previous. Check your data!)
    # Ensure volumes are aligned with actual_orders_df
```

```
    data['volumes'] = [0] + (actual_orders_df['VOLUME'] *
100).round().astype(int).tolist()

    # CONSTANTS (Physical limits of your trucks)
    TRUCK_W_CAP = physical_fleet_specs.get('weight', 2000)
    TRUCK_V_CAP = physical_fleet_specs.get('volume', 1000)

    # Sanity Check: Does the biggest order fit in a truck? (from
actual_orders_df)
    if len(data['weights']) > 1: # Check only if there are customers
        max_order_w = max(data['weights'][1:]) # Exclude depot's 0
weight
        if max_order_w > TRUCK_W_CAP:
            raise ValueError(f"CRITICAL ERROR: Order exists with
weight {max_order_w}, but truck limit is {TRUCK_W_CAP}.")

    # --- 4. Smart Fleet Sizing (The Lower Bound Calculation) ---
    total_weight = sum(data['weights'])
    total_volume = sum(data['volumes'])

    # Minimum trucks needed purely for capacity (Bin Packing Lower
Bound)
    min_trucks_weight = math.ceil(total_weight / TRUCK_W_CAP)
    min_trucks_volume = math.ceil(total_volume / TRUCK_V_CAP)

    theoretical_min_vehicles = max(min_trucks_weight,
min_trucks_volume)

    # Add a "Routing Buffer" (e.g., 20% or +2 trucks)
    # Vehicles can rarely be 100% full because they run out of Time or
Distance first.
    recommended_fleet_size = int(theoretical_min_vehicles * 1.2) +
ROUTING_BUFFER

    print(f"--- Initialization Report ---")
    print(f"Total Weight: {total_weight} | Max Truck W: {TRUCK_W_CAP}
-> Min Trucks: {min_trucks_weight}")
    print(f"Total Volume: {total_volume} | Max Truck V: {TRUCK_V_CAP}
-> Min Trucks: {min_trucks_volume}")
    print(f"Latest Deadline found: {max_deadline_in_data}")
    print(f"Number of nodes derived from distance matrix:
{num_problem_nodes}")
    print(f"Number of customers used from orders_df:
{len(actual_orders_df)}")
    print(f"Setting Fleet Size to: {recommended_fleet_size}
(Theoretical Min: {theoretical_min_vehicles})")

    data['num_vehicles'] = recommended_fleet_size
    data['vehicle_capacities_weight'] = [TRUCK_W_CAP] *
data['num_vehicles']
```

```
    data['vehicle_capacities_volume'] = [TRUCK_V_CAP] *
data['num_vehicles']

    # --- 5. Other Data ---
    # Ensure service times are aligned with actual_orders_df
    data['service_times'] = [0] +
actual_orders_df['SERVICE_TIME'].astype(int).tolist()
    data['depot'] = 0
    data['penalty'] = 100000 # Keep high

    return data

#
------------------------------------------------------------------------
# --- 3. Initialize Solver Model and Constraints ---
#
------------------------------------------------------------------------

def print_solution(data, manager, routing, solution):
   """Prints the solution found by the solver."""
   total_distance = 0
   total_time_cost = 0 # Cost is now based on time, not distance
   total_time = 0
   time_dimension = routing.GetDimensionOrDie('Time')

   # --- Collect served nodes for later dropped node reporting ---
   served_nodes_set = set()

   # Calculate the total travel time for the objective reporting
   for vehicle_id in range(data['num_vehicles']):
       index = routing.Start(vehicle_id)
       if routing.IsEnd(solution.Value(routing.NextVar(index))):
           continue

       while not routing.IsEnd(index):
           previous_index = index
           current_node_index = manager.IndexToNode(index)
           if current_node_index != data['depot']: # Don't add depot
to served_nodes_set
               served_nodes_set.add(current_node_index)

           index = solution.Value(routing.NextVar(index))

           # The actual total travel time component of the objective
           from_node = manager.IndexToNode(previous_index)
           to_node = manager.IndexToNode(index)
           # The cost minimized is the travel time (t_ij)
           total_time_cost += data['time_matrix'][from_node][to_node]

   # Calculate objective based on MIP formulation: alpha * Yk +
```

```
sum(tij * xijk)
    num_used_vehicles = len([v for v in range(data['num_vehicles']) if
not routing.IsEnd(solution.Value(routing.NextVar(routing.Start(v))))])
    vehicle_penalty_component = num_used_vehicles *
ALPHA_VEHICLE_PENALTY

    # NOTE: The OR-Tools objective value here may include dropped node
penalties,
    # but the custom calculation below reflects the MIP goal:
    mip_objective = vehicle_penalty_component + total_time_cost

    print(f'OR-Tools Objective (Time Cost + Penalties):
{solution.ObjectiveValue()}')
    print(f'MIP Objective (Alpha*Vehicles + Total Time):
{mip_objective} ({num_used_vehicles} vehicles *
{ALPHA_VEHICLE_PENALTY} + {total_time_cost})')

    # Print routes (unchanged)
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        if routing.IsEnd(solution.Value(routing.NextVar(index))):
            continue

        plan_output = f'Route for vehicle {vehicle_id} (Depot: 0):'
        route_distance = 0

        while not routing.IsEnd(index):
            time_var = time_dimension.CumulVar(index)
            node_index = manager.IndexToNode(index)

            previous_index = index
            index = solution.Value(routing.NextVar(index))

            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id
            )

            plan_output += (
                f' {node_index} -> Time({solution.Min(time_var)})'
            )

        time_var = time_dimension.CumulVar(index)
        plan_output += (
            f' {manager.IndexToNode(index)} ->
Time({solution.Min(time_var)})'
        )
        plan_output += f'\n  Route Distance: {route_distance}'
        plan_output += f'\n  Route End Time: {solution.Min(time_var)}'

        print(plan_output)
```

```
        total_distance += route_distance
        total_time = max(total_time, solution.Min(time_var))

    print(f'\nTotal distance of all used routes: {total_distance}')
    print(f'Max route end time: {total_time}')
    print(f'Total travel time cost: {total_time_cost} (The minimized
sum component)')

    # --- Report Dropped Nodes (new approach) ---
    # All nodes from 1 to num_problem_nodes-1 are customer nodes
    all_customer_nodes = set(range(1, manager.GetNumberOfNodes()))
    dropped_nodes = list(all_customer_nodes - served_nodes_set)
    dropped_nodes.sort() # For consistent output

    if dropped_nodes:
        print(f'\n!!! WARNING: {len(dropped_nodes)} Nodes Were DROPPED
(Unserved) !!!')
        print(f'Dropped Nodes (NODE_ID): {dropped_nodes}')
    else:
        print('\nSUCCESS: All nodes were served.')


def solve_vrp():
    """Entry point for the VRP solver with MIP objective."""
    data = create_data_model_smart(df_orders, df_distance, df_time)

    manager = pywrapcp.RoutingIndexManager(
        len(data['distance_matrix']), data['num_vehicles'],
data['depot']
    )

    routing = pywrapcp.RoutingModel(manager)

    # --- A. Define Cost (Time - as per MIP Objective) ---
    def time_cost_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        # The cost minimized is the travel time (t_ij)
        return data['time_matrix'][from_node][to_node]

    transit_callback_index =
routing.RegisterTransitCallback(time_cost_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # A.1. Set Objective to Minimize Time Cost (t_ij) AND Penalize
Vehicles (alpha * y_k)

    # Fixed cost (alpha * y_k): Apply a large penalty for using each
vehicle
    for vehicle_id in range(data['num_vehicles']):
```

```
        routing.SetFixedCostOfVehicle(ALPHA_VEHICLE_PENALTY,
vehicle_id)

    # A.2. Add Dropped Node Penalty
    # Iterate over customer nodes, not including the depot (node 0).
    # The range should be from 1 to num_problem_nodes - 1 (inclusive of
the last customer).
    for node in range(1, len(data['distance_matrix'])):
        routing.AddDisjunction([manager.NodeToIndex(node)],
data['penalty'])

    # --- B. Add Capacity Constraints (Weight and Volume) ---

    def add_capacity_dimension(capacity_name, capacities, demands):
        def demand_callback(index):
            node = manager.IndexToNode(index)
            return demands[node]

        demand_callback_index =
routing.RegisterUnaryTransitCallback(demand_callback)

        # Capacity bounds are imposed at every node [cite: 87]
        routing.AddDimensionWithVehicleCapacity(
            demand_callback_index,
            0, # Slack
            capacities,
            True, # This dimension is cumulative
            capacity_name
        )

    add_capacity_dimension('WeightCapacity',
data['vehicle_capacities_weight'], data['weights'])
    add_capacity_dimension('VolumeCapacity',
data['vehicle_capacities_volume'], data['volumes'])


    # --- C. Add Time Window Constraints ---

    def time_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        travel_time = data['time_matrix'][from_node][to_node]
        service_time = data['service_times'][from_node]
        # Time propagation: T_jk >= T_ik + s_i + t_ij [cite: 99]
        return travel_time + service_time

    time_callback_index =
routing.RegisterTransitCallback(time_callback)

    # Time dimension: tracks T_ik (time vehicle k starts service at
```

```
node i) [cite: 45]
    routing.AddDimension(
        time_callback_index,
        0,                                # slack_max (0 is fine here)
        data['vehicle_max_travel_time'], # Planning horizon H [cite:
28]
        False,
        'Time'
    )
    time_dimension = routing.GetDimensionOrDie('Time')

    # Apply Time Windows [cite: 94]
    # The loop should iterate over the actual number of nodes in the
problem
    for node in range(len(data['distance_matrix'])):
        index = manager.NodeToIndex(node)
        start, end = data['time_windows'][node]
        time_dimension.CumulVar(index).SetRange(start, end)

    # Apply Route Duration Constraint (limited by H) [cite: 104]
    for i in range(data['num_vehicles']):
        time_dimension.SetSpanUpperBoundForVehicle(
            data['vehicle_max_travel_time'],
            i
        )

    # --- D. Set Search Parameters and Solve ---
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
    )
    search_parameters.time_limit.seconds = 120

    # Solve the problem
    solution = routing.SolveWithParameters(search_parameters)

    # --- E. Print Solution ---
    if solution:
        print_solution(data, manager, routing, solution)
    else:
        print('No solution found!')


#
------------------------------------------------------------------------
# --- EXECUTION ---
#
```

```
------------------------------------------------------------------------

print("Running Vehicle Routing Problem Solver (MIP Objective: Minimize
Vehicles, then Time)...")
solve_vrp()
```

```
Running Vehicle Routing Problem Solver (MIP Objective: Minimize
Vehicles, then Time)...
--- Initialization Report ---
Total Weight: 1303 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 524 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Number of nodes derived from distance matrix: 85
Number of customers used from orders_df: 84
Setting Fleet Size to: 6 (Theoretical Min: 1)
OR-Tools Objective (Time Cost + Penalties): 1882
MIP Objective (Alpha*Vehicles + Total Time): 1882 (5 vehicles * 200 +
882)
Route for vehicle 0 (Depot: 0): 0 -> Time(0) 51 -> Time(31) 52 ->
Time(69) 30 -> Time(86) 31 -> Time(99) 28 -> Time(115) 27 -> Time(128)
29 -> Time(137) 25 -> Time(154) 26 -> Time(175) 24 -> Time(190) 21 ->
Time(202) 65 -> Time(213) 22 -> Time(222) 45 -> Time(237) 84 ->
Time(244) 0 -> Time(275)
  Route Distance: 351
  Route End Time: 275
Route for vehicle 1 (Depot: 0): 0 -> Time(0) 70 -> Time(22) 71 ->
Time(34) 83 -> Time(60) 23 -> Time(94) 18 -> Time(118) 7 -> Time(125)
8 -> Time(132) 9 -> Time(139) 1 -> Time(158) 44 -> Time(167) 36 ->
Time(172) 6 -> Time(184) 5 -> Time(193) 78 -> Time(213) 77 ->
Time(222) 79 -> Time(231) 80 -> Time(242) 81 -> Time(251) 76 ->
Time(267) 73 -> Time(283) 72 -> Time(291) 74 -> Time(299) 75 ->
Time(312) 0 -> Time(354)
  Route Distance: 406
  Route End Time: 354
Route for vehicle 2 (Depot: 0): 0 -> Time(0) 68 -> Time(16) 66 ->
Time(28) 67 -> Time(37) 69 -> Time(47) 47 -> Time(69) 61 -> Time(99)
57 -> Time(123) 59 -> Time(146) 58 -> Time(155) 55 -> Time(165) 56 ->
Time(179) 60 -> Time(198) 63 -> Time(215) 62 -> Time(225) 64 ->
Time(235) 0 -> Time(273)
  Route Distance: 369
  Route End Time: 273
Route for vehicle 3 (Depot: 0): 0 -> Time(0) 19 -> Time(41) 20 ->
Time(52) 82 -> Time(66) 15 -> Time(75) 17 -> Time(81) 16 -> Time(93)
14 -> Time(112) 12 -> Time(129) 11 -> Time(148) 10 -> Time(164) 13 ->
Time(171) 3 -> Time(189) 2 -> Time(211) 4 -> Time(227) 0 -> Time(281)
  Route Distance: 385
  Route End Time: 281
Route for vehicle 5 (Depot: 0): 0 -> Time(0) 46 -> Time(24) 49 ->
Time(39) 48 -> Time(50) 50 -> Time(66) 54 -> Time(85) 53 -> Time(99)
43 -> Time(117) 41 -> Time(132) 42 -> Time(146) 34 -> Time(155) 32 ->
```

```
Time(175) 35 -> Time(183) 33 -> Time(192) 37 -> Time(208) 40 ->
Time(222) 38 -> Time(231) 39 -> Time(242) 0 -> Time(275)
  Route Distance: 371
  Route End Time: 275

Total distance of all used routes: 1882
Max route end time: 354
Total travel time cost: 882 (The minimized sum component)

SUCCESS: All nodes were served.
```

Task

Implement the Vehicle Routing Problem (VRP) solver using OR-Tools' MPSolver (`pywraplp`) by defining variables, the objective function to minimize vehicles and total travel time, and core constraints for time windows and capacities, replacing the existing CP-based solver logic.(using branch and bound instead)

```
from ortools.linear_solver import pywraplp
df_orders = pd.read_csv('/content/sample_data/orders_day9.csv')
df_distance = pd.read_csv('/content/distance_day9.csv', index_col=0)
df_time = pd.read_csv('/content/sample_data/time_day9_optimistic.csv',
index_col=0)
def solve_vrp_with_mpsolver():
    """Entry point for the VRP solver using MPSolver."""
    data = create_data_model_smart(df_orders, df_distance, df_time)

    # Create the MPSolver.
    # Try 'CBC' or 'SCIP' for the solver backend.
    # SCIP is generally more powerful but CBC is often sufficient for
VRPs.
    # Correct way to instantiate Solver:
    solver = pywraplp.Solver('VRP_MPSolver',
pywraplp.Solver.SCIP_MIXED_INTEGER_PROGRAMMING)

    if not solver:
        print('Could not create solver SCIP. Falling back to CBC.')
        solver = pywraplp.Solver('VRP_MPSolver',
pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING)
        if not solver:
            raise ValueError('Could not create any solver.')

    print(f'Using solver: {solver.SolverVersion()}')

    num_nodes = len(data['distance_matrix'])
    num_vehicles = data['num_vehicles']
    max_travel_time = data['vehicle_max_travel_time']
    max_weight_capacity = data['vehicle_capacities_weight'][0] #
```

```
Assuming all vehicles have same capacity
    max_volume_capacity = data['vehicle_capacities_volume'][0] #
Assuming all vehicles have same capacity

    # 1. Decision variable x_ijk: 1 if vehicle k travels from node i
to node j
    x = {}
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i == j: # Exclude self-loops
                continue
            for k in range(num_vehicles):
                x[(i, j, k)] = solver.BoolVar(f'x_{i}_{j}_{k}')

    # 2. Decision variable y_k: 1 if vehicle k is used
    y = {}
    for k in range(num_vehicles):
        y[k] = solver.BoolVar(f'y_{k}')

    # 3. Continuous variable arrival_time[i][k]: arrival time of
vehicle k at node i (T_ik)
    arrival_time = {}
    for i in range(num_nodes):
        for k in range(num_vehicles):
            # Lower bound 0, Upper bound vehicle_max_travel_time (H)
            arrival_time[(i, k)] = solver.NumVar(0, max_travel_time,
f'T_{i}_{k}')

    # 4. Continuous variables load_w[i][k] and load_v[i][k]: load of
vehicle k after visiting node i
    load_w = {}
    load_v = {}
    for i in range(num_nodes):
        for k in range(num_vehicles):
            # Lower bound 0, Upper bound max_weight_capacity
            load_w[(i, k)] = solver.NumVar(0, max_weight_capacity,
f'LW_{i}_{k}')
            # Lower bound 0, Upper bound max_volume_capacity
            load_v[(i, k)] = solver.NumVar(0, max_volume_capacity,
f'LV_{i}_{k}')

    # Define Objective Function
    objective = solver.Objective()

    # Minimize total number of vehicles used
    for k in range(num_vehicles):
        objective.SetCoefficient(y[k], ALPHA_VEHICLE_PENALTY)

    # Minimize total travel time
    for i in range(num_nodes):
```

```
        for j in range(num_nodes):
            if i == j:
                continue
            for k in range(num_vehicles):
                objective.SetCoefficient(x[(i, j, k)],
data['time_matrix'][i][j])

    objective.SetMinimization()

    # --- Define Routing Constraints --- (from previous steps)

    # Constraint 1: Each customer node (1 to num_nodes-1) is visited
exactly once.
    for j in range(1, num_nodes): # For each customer node
        solver.Add(solver.Sum(x[(i, j, k)] for i in range(num_nodes)
if i != j for k in range(num_vehicles)) == 1)

    # Constraint 2: Each vehicle starts its route from the depot (node
0) at most once.
    for k in range(num_vehicles):
        solver.Add(solver.Sum(x[(0, j, k)] for j in range(1,
num_nodes)) <= y[k])

    # Constraint 3: Flow conservation (if a vehicle enters a node, it
must leave that node).
    for i in range(1, num_nodes): # For each customer node
        for k in range(num_vehicles):
            solver.Add(solver.Sum(x[(j, i, k)] for j in
range(num_nodes) if j != i) ==
                       solver.Sum(x[(i, j_prime, k)] for j_prime in
range(num_nodes) if j_prime != i))

    # Constraint 4: Each vehicle returns to the depot (node 0).
    for k in range(num_vehicles):
        solver.Add(solver.Sum(x[(i, 0, k)] for i in range(1,
num_nodes)) == y[k])

    # Constraint 5: Link x_ijk variables to y_k variables (if any
vehicle k travels on any arc, then y_k must be 1).
    # If a vehicle starts from the depot, then it is used.
    for k in range(num_vehicles):
        for j in range(1, num_nodes):
            # If x_0jk is 1, then y_k must be 1.  This is equivalent
to x_0jk <= y_k
            solver.Add(x[(0, j, k)] <= y[k])

    # --- Define Capacity Constraints (Weight and Volume) --- (from
previous steps)

    M_w = max_weight_capacity # Big M for weight
```

```
    M_v = max_volume_capacity # Big M for volume

    # Constraint 1 & 2 (Implicit and Explicit): Ensure load at node i
accounts for demand at i.
    # This is covered by the flow constraints and variable definition.
No extra constraint needed here.

    # Constraint 3 & 4: Load update using Big-M formulation (for
equality now)
    # If x[(i, j, k)] = 1, then load_w[(j, k)] = load_w[(i, k)] +
data['weights'][j]
    for i in range(num_nodes):
        for j in range(1, num_nodes): # For each customer node j
            if i == j:
                continue
            for k in range(num_vehicles):
                # Weight capacity constraint (Lower bound for load
flow)
                solver.Add(load_w[(j, k)] >= load_w[(i, k)] +
data['weights'][j] - M_w * (1 - x[(i, j, k)]))
                # Weight capacity constraint (Upper bound for load
flow)
                solver.Add(load_w[(j, k)] <= load_w[(i, k)] +
data['weights'][j] + M_w * (1 - x[(i, j, k)]))

                # Volume capacity constraint (Lower bound for load
flow)
                solver.Add(load_v[(j, k)] >= load_v[(i, k)] +
data['volumes'][j] - M_v * (1 - x[(i, j, k)]))
                # Volume capacity constraint (Upper bound for load
flow)
                solver.Add(load_v[(j, k)] <= load_v[(i, k)] +
data['volumes'][j] + M_v * (1 - x[(i, j, k)]))

    # Constraint: Ensure load doesn't exceed vehicle capacity at any
node
    for i in range(num_nodes):
        for k in range(num_vehicles):
            solver.Add(load_w[(i, k)] <= max_weight_capacity)
            solver.Add(load_v[(i, k)] <= max_volume_capacity)

    # Constraint: Initial load for each vehicle k at depot (node 0) is
0
    for k in range(num_vehicles):
        solver.Add(load_w[(0, k)] == 0)
        solver.Add(load_v[(0, k)] == 0)

    # --- Define Time Window Constraints --- (from previous steps)

    # 1. Link arrival times (using Big-M)
```

```
    M_time = max_travel_time # Big M for time
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i == j:
                continue
            for k in range(num_vehicles):
                # arrival_time[(j, k)] >= arrival_time[(i, k)] +
service_time[i] + time_matrix[i][j] - M_time * (1 - x_ijk)
                solver.Add(
                    arrival_time[(j, k)] >= arrival_time[(i, k)]
                    + data['service_times'][i]
                    + data['time_matrix'][i][j]
                    - M_time * (1 - x[(i, j, k)])
                )

    # 2. Enforce node time windows
    for i in range(num_nodes):
        for k in range(num_vehicles):
            start_time, end_time = data['time_windows'][i]

            # arrival_time[(i, k)] >= start_time
            solver.Add(arrival_time[(i, k)] >= start_time)
            # arrival_time[(i, k)] <= end_time
            solver.Add(arrival_time[(i, k)] <= end_time)

    # 3. Initial depot time: For the depot (node 0) and every vehicle
k, arrival_time[(0, k)] is 0
    for k in range(num_vehicles):
        solver.Add(arrival_time[(data['depot'], k)] ==
data['time_windows'][data['depot']][0]) # This should be 0

    # --- Solve the Problem and Print the Solution ---
    print('\nSolving the problem...')
    status = solver.Solve()

    if status == pywraplp.Solver.OPTIMAL or status ==
pywraplp.Solver.FEASIBLE:
        print(f'Solution found! Status: {solver.StatusName(status)}')
        print(f'Total objective value: {solver.Objective().Value()}')

        total_travel_time_cost = 0
        num_used_vehicles = 0
        served_nodes = set()

        # Collect information for printing
        routes_output = []

        for k in range(num_vehicles):
            if y[k].solution_value() > 0.5: # Vehicle k is used
                num_used_vehicles += 1
```

```
                route_for_vehicle_k = []
                current_node = data['depot']
                total_route_distance = 0
                route_time_start = arrival_time[(data['depot'],
k)].solution_value()

                # Build the route
                # Find the first node from depot
                next_node = -1
                for j in range(1, num_nodes): # Iterate through
possible first customer nodes
                    if x[(data['depot'], j, k)].solution_value() >
0.5:
                        next_node = j
                        break

                if next_node != -1:
                    route_for_vehicle_k.append(data['depot'])
                    route_for_vehicle_k.append(next_node)
                    # Accumulate travel time cost for objective
calculation
                    total_travel_time_cost += data['time_matrix']
[data['depot']][next_node]
                    total_route_distance += data['distance_matrix']
[data['depot']][next_node]
                    served_nodes.add(next_node)
                    current_node = next_node

                    # Continue building the route until back to depot
                    while current_node != data['depot']:
                        found_next = False
                        for next_j in range(num_nodes):
                            if current_node == next_j: # Avoid self-
loop
                                continue
                            if x[(current_node, next_j,
k)].solution_value() > 0.5:
                                if next_j != data['depot']:
                                    route_for_vehicle_k.append(next_j)
                                    served_nodes.add(next_j)
                                total_travel_time_cost +=
data['time_matrix'][current_node][next_j]
                                total_route_distance +=
data['distance_matrix'][current_node][next_j]
                                current_node = next_j
                                found_next = True
                                break
                        if not found_next and current_node !=
data['depot']:
                             # This case should ideally not happen if
```

```
flow constraints are correct
                            # But as a safety measure, break to
prevent infinite loop for malformed route
                            print(f"Warning: Route for vehicle {k}
seems incomplete after node {current_node}")
                            break
                    route_for_vehicle_k.append(data['depot'])

                    # Generate the output string for the route
                    plan_output = f'Route for vehicle {k} (Depot:
{data["depot"]}): '
                    for node_idx in range(len(route_for_vehicle_k)):
                        node = route_for_vehicle_k[node_idx]
                        plan_output += f' {node} ->
Time({int(arrival_time[(node, k)].solution_value())})'
                        if node_idx < len(route_for_vehicle_k) - 1:
                            plan_output += ''

                    # Adjust the last arrow and depot time
                    plan_output = plan_output.replace(f' ->
Time({int(arrival_time[(data["depot"], k)].solution_value())})
{data["depot"]} -> Time({int(arrival_time[(data["depot"],
k)].solution_value())})', f' -> {data["depot"]} ->
Time({int(arrival_time[(data["depot"], k)].solution_value())})')

                    # Get the actual end time of the route at the
final depot visit
                    final_depot_arrival_time =
arrival_time[(data['depot'], k)].solution_value()
                    plan_output += f'\n  Route Distance:
{total_route_distance}'
                    plan_output += f'\n  Route End Time:
{int(final_depot_arrival_time)}'
                    routes_output.append(plan_output)
                else:
                    # Vehicle used but doesn't depart depot
                    pass # This vehicle is 'used' but inactive if y[k]
is 1 but x[0,j,k] are all 0

        # Print all collected routes
        for route_str in routes_output:
            print(route_str)

        # Calculate MIP objective (Alpha*Vehicles + Total Time)
        mip_objective = num_used_vehicles * ALPHA_VEHICLE_PENALTY +
total_travel_time_cost
        print(f'MIP Objective (Alpha*Vehicles + Total Time):
{mip_objective} ({num_used_vehicles} vehicles *
{ALPHA_VEHICLE_PENALTY} + {total_travel_time_cost})')
```

```
        # Report Dropped Nodes
        all_customer_nodes = set(range(1, num_nodes))
        dropped_nodes = list(all_customer_nodes - served_nodes)
        dropped_nodes.sort() # For consistent output

        if dropped_nodes:
            print(f'\n!!! WARNING: {len(dropped_nodes)} Nodes Were
DROPPED (Unserved) !!!')
            print(f'Dropped Nodes (NODE_ID): {dropped_nodes}')
        else:
            print('\nSUCCESS: All nodes were served.')

    elif status == pywraplp.Solver.INFEASIBLE:
        print('No solution found, problem is INFEASIBLE!')
    elif status == pywraplp.Solver.UNBOUNDED:
        print('Problem is UNBOUNDED!')
    else:
        print(f'Solver did not find a solution. Status:
{solver.StatusName(status)}')

# Call the solver function to execute
solve_vrp_with_mpsolver()
```

```
--- Initialization Report ---
Total Weight: 1155 | Max Truck W: 2000 -> Min Trucks: 1
Total Volume: 446 | Max Truck V: 1000 -> Min Trucks: 1
Latest Deadline found: 360
Number of nodes derived from distance matrix: 79
Number of customers used from orders_df: 78
Setting Fleet Size to: 6 (Theoretical Min: 1)
Using solver: SCIP 9.2.2 [LP solver: SoPlex 7.1.3]

Solving the problem...
```

References:

[1] Maersk. "Last-Mile Delivery: Key Challenges and How to Overcome Them." Maersk. Accessed [February 2, 2025]. https://libguides.uno.edu/how/notavail.

[2] Vrani, Anna, Savvas D. Apostolidis, Athanasios Ch. Kapoutsis, and Elias B. Kosmatopoulos. "Delivering Data: A Real-World Dataset for Last-Mile Delivery Optimization." *Data in Brief* 61 (2025): 111762. doi:10.1016/j.dib.2025.111762