Group 19: Min-Max Vehicle Routing Problem with Postman Collecting Packages

My Anh Tran Nguyen
20235474
Anh.TNM235474

Tu Dang Anh
202400118
Tu.DA2400118

Trung Thanh Nguyen
202416757
Trung.TN2416757

Trong Phan Duc
202416755
Trong.PD2416755

Abstract

Min-Max Vehicle Routing Problem with Postman Collecting Packages is the task of optimizing delivery routes for multiple vehicles to collect packages from distributed locations while minimizing the maximum route distance among all vehicles. It involves using combinatorial optimization algorithms and metaheuristic approaches to analyze spatial distribution patterns and make optimal routing decisions to ensure balanced workload distribution across all postal vehicles for efficient package collection operations. This report is a part of our work in the class 159559. It introduces solving the Min-Max Vehicle Routing Problem for postal package collection systems, divided into multiple algorithmic tiers based on problem size, by sophisticated optimization techniques with five main algorithms: OR-Tools VRP Solver with Guided Local Search, Genetic Algorithm with Balanced Mutation, Local Search with Multiple Neighborhood Structures, Geographic Clustering with Polar Coordinates, and Balanced Greedy Construction using Modified Savings Algorithm. The implementation employs a tiered approach that strategically selects algorithms based on instance size, ranging from exact optimization methods for small problems to ultra-fast heuristics for large-scale instances, ensuring computational feasibility within strict time constraints while maintaining solution quality.

The full implementation of this project is publicly available in the following Repository.

Key words: Combinatorial Optimization, Vehicle Routing Problem, Min-Max Objective, Tabu Search, Beam Search, Ant Colony Optimization, OR-Tools, Metaheuristics, Postal Operations, Package Collection.

1 Introduction

The Min-Max Vehicle Routing Problem has emerged as a vital component of modern logistics and postal operations, offering significant opportunities for optimizing delivery efficiency and ensuring balanced workload distribution across multiple vehicles. With its critical importance in package collection systems and increasing demands from both commercial and governmental postal services, accurately solving the Min-Max Vehicle Routing Problem has become a crucial task for logistics managers, postal operators, and transportation planners. The ability to solve Min-Max Vehicle Routing Problems can provide valuable insights for postal operations, enabling managers to make informed decisions regarding route assignments, vehicle scheduling, and resource allocation. Furthermore, optimal solutions can help postal authorities identify potential bottlenecks and implement appropriate measures to safeguard operational efficiency while maintaining service quality standards.

Despite the importance of Min-Max Vehicle Routing optimization, the task remains highly challenging due to the complex nature of combinatorial optimization problems, the interplay of various spatial and temporal constraints, and the presence of inherent computational complexities. However, advancements in computational techniques, metaheuristic algorithms, and access to powerful optimization frameworks have opened up new avenues for developing robust solution methodologies.

This project report aims to explore the field of Min-Max Vehicle Routing Problem solving in the context of postal package collection operations. By employing combinatorial optimization, metaheuristic approaches, and advanced algorithmic techniques, we seek to minimize the maximum route distance among all vehicles while examining the factors influencing optimal route construction and load balancing.

The primary objectives of this project are as follows:

- To analyze the Min-Max Vehicle Routing Problem structure, including distance matrices, vehicle capacity constraints, and other

relevant operational parameters.

- To identify key optimization challenges, computational bottlenecks, and algorithmic dependencies that can aid in understanding the underlying dynamics of route optimization problems.

- To develop solution methodologies using metaheuristic algorithms and optimization techniques to solve Min-Max Vehicle Routing Problems efficiently within strict time constraints.

- To evaluate the performance of the solution approaches based on established evaluation metrics and compare them against traditional routing optimization methods.

By the conclusion of this project, we expect to achieve the following outcomes:

- A comprehensive analysis of Min-Max Vehicle Routing Problem characteristics, identifying optimization patterns and computational challenges specific to postal package collection scenarios.

- Development of efficient solution algorithms capable of handling various problem sizes from small-scale exact optimization to large-scale heuristic approximation.

- Comparative analysis between the performance of different algorithmic approaches and traditional vehicle routing solution methods.

- Insights and recommendations for postal operators, logistics managers, and transportation planners to make informed decisions in package collection route optimization.

Through this project, we aim to contribute to the growing body of research in Vehicle Routing Problem optimization while providing practical implications for stakeholders in postal and logistics operations.

2 Data Analysis

The *Min-Max Vehicle Routing Postman Collecting Packages* problem aims to optimize the assignment of pickup locations to K postmen such that the longest travel distance among them is minimized. The computational complexity of this problem grows with two primary parameters: the number of pickup locations N and the number of postmen K. Specifically, N affects the size of the distance matrix $d(i, j)$ and the number of binary variables and constraints in the optimization model, while K determines the number of routes and the complexity of load balancing.

To evaluate the difficulty of the test cases, we consider two main criteria: (i) the input size (small, medium, large, very large) and (ii) the estimated computational difficulty (easy, medium, hard, very hard), which reflects the required effort to obtain an optimal or near-optimal solution.

From the table 1, we observe the following:

- **Easy cases** (Tests 1, 6, 9, 12, 13) with $N \leq 100$ are appropriate for validating algorithm correctness and stability.

- **Medium to large cases** (Tests 2, 8, 11, 3, 5) demand more efficient optimization strategies, such as heuristic or warm-start techniques.

- **Very hard cases** (Tests 4, 7, 10), with $N \geq 900$, represent stress tests that require a combination of techniques including *cutting planes*, *soft constraints on* $L_{\max}$, and *problem decomposition*.

This categorization enables a structured evaluation of algorithmic performance across varying input scales, supporting both correctness validation and scalability benchmarking.

3 Algorithm Selection

A. Tabu Search

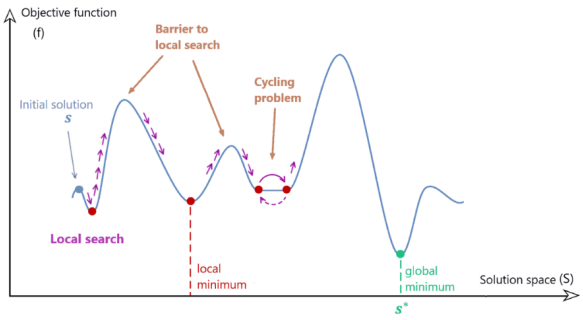
Tabu Search (TS) is an iterative neighborhood search algorithm, where the neighborhood changes dynamically. Tabu Search enhances local search by avoiding points in the search space which are already visited. By avoiding already visited points, loops in search space are avoided and local optima can be escaped.

Core Idea Tabu Search guides a local heuristic search procedure to explore the solution space beyond local optimality by allowing non-improving moves, but uses a "tabu list" to avoid cycling back to previously visited solutions.

Operating Principle

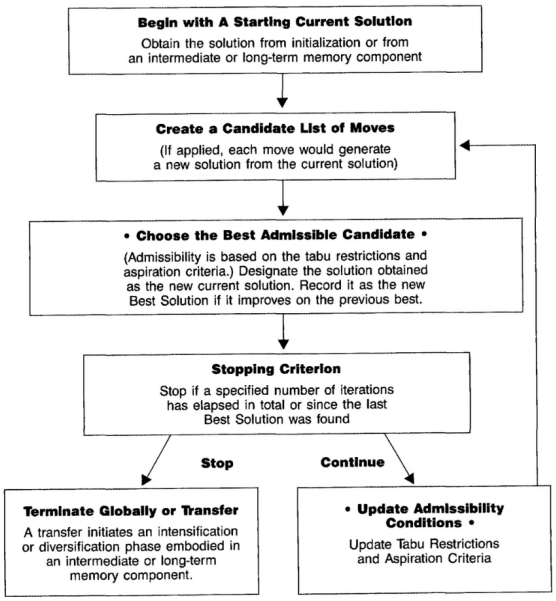
Table 1: Test Case Difficulty Evaluation

Test	N	K	Scale	Difficulty	Remarks
1	10	2	Small	Easy	Very small size, suitable for testing logic correctness.
2	600	20	Medium	Medium	Balanced between number of points and postmen; requires efficient task allocation.
3	800	20	Large	Hard	Large search space, prone to subtour formation.
4	1000	20	Very Large	Very Hard	Near input limit, requires cutting-plane techniques and problem decomposition.
5	700	20	Large	Hard	Similar in size to Test 3; efficient pruning techniques are necessary.
6	100	5	Small	Easy	Solvable by exact MILP methods in reasonable time.
7	1000	20	Very Large	Very Hard	Extremely large instance, requires advanced optimization techniques.
8	500	10	Medium	Medium	Solvable within reasonable time if good initial solutions are used.
9	100	5	Small	Easy	Similar to Test 6, for validating model correctness.
10	900	20	Very Large	Very Hard	Computationally intensive, may produce many unfavorable configurations.
11	200	10	Medium	Medium	Moderate size, amenable to hybrid heuristic-exact methods.
12	6	2	Small	Easy	Useful for unit tests or debugging purposes.
13	6	2	Small	Easy	Same as Test 12, used for baseline validation.

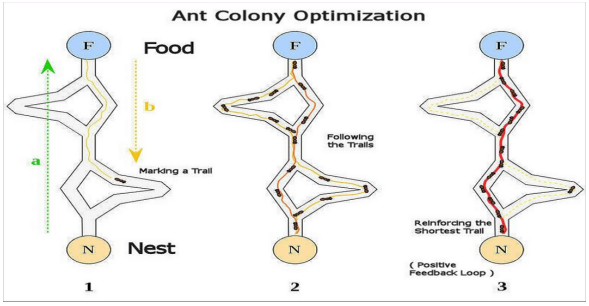


- **Initial Solution:** Start with a feasible initial solution (often generated randomly or via a heuristic)
- **Neighborhood Exploration:** At each iteration, generate a set of neighbor solutions by applying small modifications (called moves) to the current solution.
- **Move Evaluation:**
  - Evaluate all (or a subset of) neighbor solutions using an objective function (e.g., minimize cost, distance, time).
  - Choose the best candidate, even if it is worse than the current solution, to allow escape from local minima.
- **Tabu List (Short-Term Memory):**
  - Store recent moves (or attributes of moves) in a tabu list to prevent cycling and avoid revisiting recent solutions.
  - Each move in the tabu list is tabu (forbidden) for a certain number of iterations, called the tabu tenure.
- **Aspiration Criterion:** Override the tabu status if a move leads to a solution better than any seen so far (i.e., it satisfies an aspiration criterion).
- **Update Best Solution:** If the new candidate solution is better than the best found so far, update the global best solution.
- **Termination Condition:** Stop when a certain number of iterations is reached, or when no improvement is found for a number of steps, or when time limit is exceeded.

B. Ant Colony Optimization

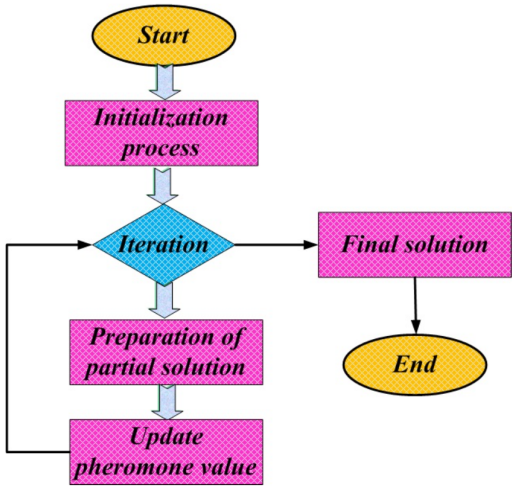


Ant Colony Optimization (ACO) is a nature-inspired metaheuristic based on the foraging behavior of certain ant species. In nature, ants search for food and find the shortest path between their nest and the food source by depositing a chemical substance called pheromone. As ants travel, they leave pheromone trails along the paths they take. Other ants are more likely to follow paths with stronger pheromone concentrations. Over time, shorter (and better) paths accumulate more pheromone, making them more attractive and effectively guiding the entire colony toward optimal or near-optimal solutions.



ACO mimics this natural behavior in the following way:

- Artificial ants explore possible solutions.
- Pheromone trails are updated to favor better (shorter, faster, or cheaper) solutions.
- Over many iterations, the algorithm converges to an optimal or near-optimal solution.



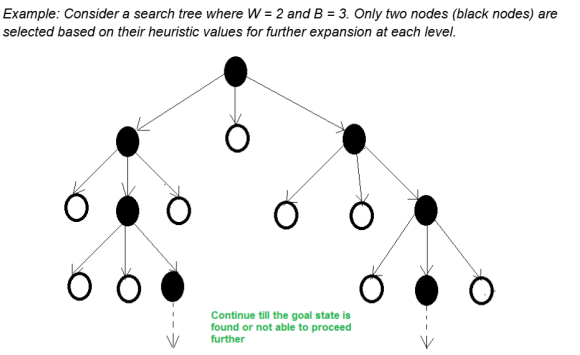
C. Beam Search

Beam Search is a heuristic search algorithm that navigates a graph by systematically expanding the most promising nodes within a constrained set. This approach combines elements of breadth-first search to construct its search tree by generating all successors at each level. However, it only evaluates and expands a set number, W, of the best nodes at each level, based on their heuristic values. This selection process is repeated at each level of the tree.

Characteristics of Beam Search:

- **Width of the Beam (W):** This parameter defines the number of nodes considered at each level. The beam width W directly influences the number of nodes evaluated and hence the breadth of the search.
- **Branching Factor (B):** If B is the branching factor, the algorithm evaluates W×B nodes at every depth but selects only W for further expansion.
- **Completeness and Optimality:** The restrictive nature of beam search, due to a limited beam width, can compromise its ability to find the best solution as it may prune potentially optimal paths.
- **Memory Efficiency:** The beam width bounds the memory required for the search, making beam search suitable for resource-constrained environments.

D. OR-Tools VRP Solver with Guided Local Search



OR-Tools is Google's open-source optimization suite that provides state-of-the-art algorithms for solving complex combinatorial optimization problems. The Vehicle Routing Problem solver within OR-Tools implements a three-part algorithmic framework: first-solution heuristics generate initial vehicle tours, local search improves these solutions with metaheuristic guidance, and a constraint programming engine either proves optimality or further improves the best solution.

Guided Local Search (GLS) is a meta-heuristic technique that enhances traditional local search by using penalty-based mechanisms to escape local optima. The algorithm maintains a set of solution features and dynamically adjusts penalty weights based on search history, systematically guiding the exploration away from previously visited solution regions. This approach is particularly effective for vehicle routing problems because it can handle multiple objectives simultaneously while maintaining solution feasibility throughout the search process.

The OR-Tools routing solver focuses on generality and can handle complex industrial constraints including vehicle capacities, time windows, pick-up-and-delivery precedence rules, and incompatible shipments within the same vehicle. The solver's strength lies in its ability to balance solution quality with computational efficiency for large-scale problems.

E. Genetic Algorithm with Balanced Mutation

Genetic Algorithms are population-based evolutionary meta-heuristics inspired by natural selection and genetics. The algorithm maintains a population of candidate solutions (chromosomes) and evolves them through successive generations using three primary operators: selection, crossover, and mutation. Each solution is evaluated using a fitness function, and fitter individuals have higher probabilities of being selected for reproduction.

Tournament selection is a common selection mechanism where a small subset of individuals compete, and the fittest is chosen as a parent. This method maintains selection pressure while preserving population diversity. Order crossover (OX) is particularly effective for permutation-based problems like vehicle routing, as it preserves the relative ordering of elements while allowing meaningful genetic material exchange between parents.

Balanced mutation specifically addresses optimization problems with min-max objectives. Unlike standard mutation operators that make random changes, balanced mutation is biased toward moves that improve the worst-case scenario. In routing contexts, this means prioritizing mutations that reduce the length of the longest route, such as relocating customers from heavily loaded routes to lighter ones. This specialized approach ensures the evolutionary process is guided toward solutions that minimize the maximum objective value.

F. Local Search with Multiple Neighborhood Structures

Local Search is a fundamental optimization technique that iteratively improves solutions by exploring neighborhoods and moving to better solutions until reaching a local optimum. The algorithm starts with an initial solution and systematically examines neighboring solutions, accepting improvements until no better neighbors exist.

Multiple neighborhood structures provide different transformation mechanisms for modifying current solutions. The 2-opt operator removes two edges from a route and reconnects the path differently, effectively reversing a route segment. This operator is particularly effective for eliminating crossing edges and reducing total travel distance. Inter-route operators like customer relocation and exchange moves transfer customers between different routes, enabling load balancing across vehicles.

Variable Neighborhood Search (VNS) systematically combines multiple neighborhood structures within a single framework. The key principle is that different neighborhood operators can escape different types of local optima, leading to superior overall solution quality. The algorithm typically cycles through different neighborhood structures, applying local search within each neighborhood until no improvements are found, then switching to a different neighborhood type.

G. Geographic Clustering with Polar Coordinates Geographic clustering is a spatial parti-

tioning technique that groups customers based on geographical proximity to create natural service regions. This approach transforms the complex vehicle routing problem into smaller, more manageable subproblems by exploiting the spatial structure of customer locations.

Polar coordinate transformation converts Cartesian coordinates (x, y) to polar coordinates (r,) relative to a central reference point, typically the depot. The radial coordinate r represents the distance from the origin, while the angular coordinate represents the direction. This transformation enables efficient angular-based partitioning where customers are grouped into sectors based on their angular position.

The clustering process involves sorting customers by angular coordinates and partitioning them into k groups corresponding to the number of available vehicles. This method naturally creates geographically compact regions that minimize inter-cluster travel while balancing intracluster workload. The polar coordinate approach is computationally efficient and provides a strong foundation for constructing initial solutions in large-scale routing problems.

H. Balanced Greedy Construction (Modified Savings Algorithm)

The Savings Algorithm, developed by Clarke and Wright in 1964, is one of the most fundamental constructive heuristics for vehicle routing problems. The algorithm begins with individual routes from the depot to each customer and back, then iteratively merges routes based on calculated savings values. The savings s(i,j) for combining customers i and j is computed as s(i,j) = d(0,i) + d(0,j) - d(i,j), where d represents distances and 0 is the depot.

The original algorithm merges routes in decreasing order of savings values, subject to capacity and operational constraints. This greedy approach efficiently constructs feasible solutions by always selecting the merge that provides the greatest immediate benefit in terms of distance reduction.

The balanced modification addresses min-max objectives by incorporating load balancing considerations into the construction process. Instead of purely maximizing savings, the modified algorithm adjusts savings values based on current route length distributions and the potential impact on overall balance. This ensures that route merges not only reduce total distance but also maintain relatively equal route lengths across all vehicles, providing a superior starting point for subsequent optimization phases.

4 Implementation

A. Tabu Search

- Step 1: Initial Solution: Generated by randomly shuffling all N nodes and assigning them round-robin to the K postmen.
- Step 2: Cost Evaluation: The cost of a solution is defined as the maximum total distance among all K routes.
- Step 3: Neighborhood Generation:
  - At the beginning, I tried to use Brute Force to return all the neighbors which wasted a lot of time and memory.
  - Neighbors are generated by moving a node from one route to another. For each move, the delta cost is computed to estimate its impact. A set number (e.g., 100) of neighbors are evaluated.
- Step 4: CostChange Function (delta cost): Calculates the difference in total distance resulting from moving a node between two routes, without fully recomputing the entire route costs.
- Step 5: Move Execution: Performs the actual node transfer between routes.
- Step 6: Tabu List: Tracks recently performed moves. Each move is defined by its origin and position, and stays in the list for a limited number of iterations (TABU tenure). If a move is tabu (recently used), it is skipped unless it results in a globally better solution.
- Step 7: Main Loop: Runs for a maximum number of iterations (MAXLOOP). In each iteration, the best non-tabu neighbor that improves the solution is selected and applied.

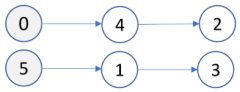
B. Ant Colony Optimization

Input the values N, K, and the distance matrix d, then generate virtual depots. For example:

Initialize the pheromone matrix with an initial value of 1.0 for all edges.

Select the next move for an ant based on probabilities, using the Roulette Wheel Selection method:

0	**0**	**3**	**1**	**5**	**7**	**0**
1	**3**	**0**	**2**	**3**	**4**	**3**
2	**1**	**4**	**0**	**2**	**6**	**1**
3	**1**	**2**	**4**	**0**	**2**	**1**
4	**4**	**5**	**7**	**7**	**0**	**4**
5	**0**	**3**	**1**	**5**	**7**	**0**



Node 5 is the virtual depot

- Formula to compute probability on component (i, j):

$$p(i,j) = \frac{\tau_(i,j)^{\alpha} * \eta_(i,j)^{\beta}}{\sum_(p,q) \in Candidate \tau_(p,q)^{\alpha} * \eta_(p,q)^{\beta}}$$

$\tau_(i,j)$: pheromone on component (i, j)

$\eta_(i,j)$: heuristic function for selecting component (i, j)

- Compute heuristic component:
  - If virtual depot: use load balancing formula based on A (average of length in K routes)
  - If rel pickup point: $\frac{1}{d_(i,j)+10^{-}6}$ (The denominator includes $10^{-}6$ to avoid division by zero)

- Roulette wheel selection:

```
Step 1: Compute total probability
    total = sum(probabilities.values())

Step 2: Generate a random number r in [0, total)
    r = random.uniform(0, total)

Step 3: Iterate over items and accumulate probability
    cumulative = 0
    for item, prob in probabilities.items():
        cumulative += prob
        if cumulative >= r:
            return item
```

- Construct route of ant by 2 ways:
  - Use virtual depots to balance the load across postmen:
    - Select the next node based on pheromone and heuristic.
    - If the next node is a virtual depot, switch to a new active postman.
    - Else, assign node to the current postman's route and update distance.
    - Reassign nodes from overloaded routes to empty routes.
  - Use greedy selection based on the shortest route length:
    - Select the next node based on pheromone and heuristic.
    - Assign nodes to the postman with the shortest current route.
  - Update pheromones:
  - Applied both upper and lower bounds to pheromone values.
  - Apply evaporation to all pheromone levels.
  - Increase pheromone levels on paths in the route (based on quality).

$$\tau_{ij} = \rho \cdot \tau_{ij} + \sum_{k} \frac{m}{1 + L_k}$$

ρ s the pheromone evaporation rate.
m is a parameter controlling pheromone deposite intensity.
L_k the length (cost) of the ant k route.
The denominator includes $1+L_k$ to avoid division by zero.

Main Ant Colony Optimization Algorithm:

- Begin with greedy route construction.
- In each iteration:
  - Each ant constructs a solution.
  - Evaluate the solutions.
  - Update the pheromones.
  - Return the best solution found.

C. Beam Search

Step 1: Initial State: Each state represents a partial solution and can be defined as:

- **routes**: A list of K routes, each row k is a list of points assigned to postman k, starting from 0.
- **unassigned**: A set of points that haven't been assigned yet.

Step 2: Evaluate function:

- The evaluate function will compute the distance of each route in K route and return the maximum route distance among all K routes.
  - **`max_cost`**: the maximum distance across all K routes

```
MAX_ROUTE_DISTANCE(routes, d): [EVALUATE FUNCTION]
    max_dist ← 0

    for each route in routes:
        total_distance ← sum of distances between consecutive points
        max_dist ← max(max_dist, total_distance)

    return max_dist
```

- The state will be recognized "better" if it has smaller value.

Step 3: State Expansion: While there are still unassigned points

- For each state in the current beam:
  - Generate child states by assigning one unassigned point to one of the K vehicles
  - Calculate the cost: update the route's total cost and the max route distance among all routes.
- Collect all generated child states.
- Select the top B best states (based on the minimum max route cost) to keep for the next iteration.

Step 4: Termination

- The algorithm stops when there are no unassigned points left.
- Return the state with the smallest max_cost

```
FUNCTION BEAM_SEARCH(d, N, K, beam_width):
    Initialize the initial state:
        - K routes, each route k starting with the depot [0]
        - set(visited_points) = empty

    beam ← list containing the initial state

    LOOP until a complete state is found:
        new_beam ← empty list
        completed ← empty list

        for each state in beam:
            if all N points are visited:
                add state to completed
                continue to next state

            Find all unvisited points

            for each point in unvisited points:
                for each route (from 0 to K-1):
                    - Copy the current state
                    - Add the point to the selected route
                    - Update the set(visited_points)

                    if this new state has not been visited:
                        - Compute the max route distance
                        - Store it and add the new state to new_beam

        if completed is not empty:
            return the state in completed with the smallest max_route_distance

        beam ← top beam_width best states from new_beam (based on max route distance)
```

D. Or-Tools Application Algorithm combined with Generative Algorithm

The Min-Max Vehicle Routing Problem implementation employs a tiered algorithmic strategy that partitions the problem space based on instance size, with each tier optimized for specific computational and quality trade-offs. The partitioning strategy addresses the fundamental challenge that no single algorithm can efficiently handle the entire spectrum from small-scale exact optimization to large-scale heuristic approximation within strict time constraints.

Tier 1: Small Instances (N < 50) - OR-Tools Exact Optimization

Rationale for Interval Division: Small instances with fewer than 50 customers can be solved near-optimally within reasonable time limits using exact or high-quality heuristic methods. The computational complexity remains manageable, allowing for sophisticated constraint programming approaches.

The implementation deploys OR-Tools VRP solver with guided local search, allocating 10 seconds for optimization. The solver creates a routing model with capacity constraints:

The Min-Max Vehicle Routing Problem (VRP) implementation employs a **tiered algorithmic strategy** that partitions the problem space based on instance size. Each tier is optimized for specific computational and quality trade-offs. This approach addresses the challenge that no single algorithm can efficiently handle all instance sizes from small-scale exact optimization to large-scale heuristic approximation under strict time constraints.

Tier 1: Small Instances ($N < 50$) - OR-Tools Exact Optimization

Rationale: Small instances with fewer than 50 customers can be solved near-optimally using exact or high-quality heuristic methods. Computational complexity remains manageable, allowing for advanced constraint programming.

OR-Tools' VRP solver is used with guided local search, allocating 10 seconds for optimization. A routing model with capacity constraints is defined as:

$$\text{capacity}[k] = \left\lfloor \frac{N}{K} \right\rfloor + \begin{cases} 1 & \text{if } k < N \bmod K \\ 0 & \text{otherwise} \end{cases}$$

The distance callback is registered as:

`routing.SetArcCostEvaluatorOfAllVehicles(transit\_callback\_index)`

The solver uses branch-and-bound and guided local search to escape local optima, prioritizing solution quality over speed.

Tier 2: Medium Instances ($50 \leq N \leq 150$) – Hybrid Greedy-Local Search

Rationale: Medium instances require balance between efficiency and solution quality. (**?**)Exact methods become expensive; simple heuristics may underperform.

- A modified savings algorithm computes:

$$s(i,j) = d(0,i) + d(0,j) - d(i,j)$$

- To reduce complexity from $O(N^2)$ to $O(N \cdot k)$, where $k = \min(20, N)$:

```
for j in self.nearest_neighbors.
get(i, [])[:10]:
    savings.append((save, i, j))
```

- Local search includes 2-opt and customer swapping with acceptance rule:

$$\text{accept} = \max(\text{new\_dist}_i, \text{new\_dist}_j) < \max(\text{old\_dist}_i, \text{old\_dist}_j)$$

- An 8-second fallback using OR-Tools ensures solution robustness.

Tier 3: Large Instances ($150 < N \leq 400$) – Genetic Algorithm Optimization

Rationale: Large instances exceed practical limits of exact methods. Genetic algorithms (GAs) provide global optimization with diversity maintenance.

- Population of 30 over 50 generations.
- Tournament selection (size 3):

```
tournament_indices = random.sample(range(len(population)), tournament_size)
winner_idx = max(tournament_indices, key=lambda i: fitness_scores[i])
```

- Fitness function transforms min-max objective:

$$f(x) = \frac{1}{1 + \text{max\_distance}}$$

- Mutation probability:

$$P(\text{mutation}) = \frac{\text{route\_length} - \text{min\_length}}{\text{max\_length} - \text{min\_length}}$$

Tier 4: Very Large Instances ($400 < N \leq 900$) – Clustering-Based Decomposition

Rationale: Very large instances require decomposition using spatial clustering to reduce complexity.

- Customers are clustered using round-robin:

$$\text{cluster\_idx} = \text{customer\_idx} \bmod K$$

- Transform to polar coordinates:

$$r_i = d(0,i), \quad \theta_i = \arctan 2(y_i - y_0, x_i - x_0)$$

```
customers.sort(key=lambda x: x[2])  # Sort by angle
for i, (customer_id, _, _) in enumerate(customers):
    route_idx = i % self.k
    routes[route_idx].append(customer_id)
```

- Nearest neighbor improvement with complexity $O(N^2/K)$ per route.

Tier 5: Ultra-Large Instances ($N > 900$) – Ultra-Fast Heuristics

Rationale: Extremely large instances need ultra-fast heuristics with acceptable quality trade-offs.

- **Phase 1 (2s):** Quick polar partitioning, $O(N \log N)$
- **Phase 2 (3s):** Fast nearest neighbor improvement:

$$\text{max\_iterations} = \min(10, 2N)$$

- **Phase 3 (13s):** Bounded local search with 2-opt:

$$\text{max\_2opt\_moves} = \min(100, \frac{N}{10})$$

- Efficient route distance calculation:

$$d(\text{route}) = \sum_{i=0}^{|\text{route}|-2} d(\text{route}[i], \text{route}[i+1]) + d(\text{route}[-1], 0)$$

This tier ensures feasibility within 18 seconds even for $N > 1000$, with good quality via intelligent design.

5 Model Tuning

A. Tabu Search

Incorporation of Cost-Based Neighbor Evaluation

Original Approach: The initial implementation generated neighbors by moving or swapping nodes between routes without explicitly evaluating the cost impact of these operations.

Enhanced Approach: The revised code introduces the CostChange function, which calculates the cost difference when moving a node from one route to another. This allows the algorithm to prioritize moves that lead to the most significant improvements in the objective function.

Efficient Neighbor Generation

Original Approach: All possible neighbors were generated exhaustively, which could be computationally intensive.

Enhanced Approach: The NeighborsGen function limits the number of generated neighbors to a maximum (e.g., 100), randomly selecting potential moves. This stochastic approach reduces computational overhead while maintaining a diverse set of candidate solutions.

Tabu List with Expiration Mechanism

Original Approach: The Tabu list stored complete solutions, which could be memory-intensive and slow to compare.

Enhanced Approach: The Tabu list now stores specific move attributes along with an expiration time (loop + Tabu). This allows for quicker checks against prohibited moves and ensures that Tabu status is temporary, promoting exploration after a certain number of iterations.

Adaptive Move Acceptance Criteria

Original Approach: Moves were accepted based solely on their absence from the tabu list and improvement over the best-known solution.

Enhanced Approach: The algorithm now considers the cost impact of moves (delta) and allows certain non-improving moves if they are not tabu, facilitating a balance between intensification and diversification in the search process.

Structured Output Formatting

Original Approach: The output format was less structured, potentially making it harder to interpret results.

Enhanced Approach: The main function provides a clear and organized output, displaying the number of routes and the sequence of nodes in each route, enhancing readability and usability of the results.

B. Ant Colony Optimization

Time Limit Handling Time limit handling is a critical component in the design of optimization algorithms like Ant Colony Optimization (ACO), especially for large-scale or real-time applications. By enforcing a maximum runtime, the algorithm ensures it always returns a solution within the required time, even if the problem size is large or the search space is complex.

Hyperparameter Tuning Is the process of finding parameter values that improve model performance of the model, maximize the efficiency, yet still ensure the complexity within a dataset. This process is used not only to improve the predictive power of the model, but also to improve the speed, making the model run faster.

Adaptive Parameters Parameters such as alpha, beta, and evaporation rate are dynamically adjusted over iterations. This adaptive tuning makes the algorithm more flexible and helps it avoid getting trapped in local optima.

Pheromone Update Strategy Pheromone updates incorporate both iteration-best and global-best solutions, depending on the quality gap between them. In addition, upper and lower bounds are applied to the pheromone values to prevent them from becoming too extreme. This strategy helps ensure convergence while avoiding premature stagnation.

Load Rebalancing Strategy During the experiment, I discovered that several routes assigned to postmen, particularly those generated using virtual depots, contained no pickup points. This issue occurred because more than two virtual depots were selected toward the end of the route construction process.

To address this, I implemented a load rebalancing strategy, which reassigns nodes from overloaded routes to empty ones, ensuring that all postmen are given meaningful workloads.

C. Or-Tools Application Algorithm combined with Generative Algorithm

The development of the Min-Max Vehicle Routing Problem (VRP) solver underwent four distinct evolutionary phases, each addressing specific computational bottlenecks and solution quality requirements identified through systematic testing and performance analysis.

Phase 1: Initial OR-Tools Implementation

Initial Approach: The first implementation relied exclusively on the Google OR-Tools VRP solver across all problem sizes, using a uniform algorithmic strategy without size-based differentiation.

Implementation Strategy:

- Employed OR-Tools' constraint programming framework with Guided Local Search metaheuristic.
- Configuration:
  - Time limit: 30 seconds for all instances.
  - Search strategy: `GUIDED_LOCAL_SEARCH`.
  - Vehicle capacity constraints based on balanced distribution.

Performance Analysis:

- Critical performance degradation observed for instances with $N \geq 100$ customers.
- Constraint programming provided high-quality solutions for small instances but showed exponential time complexity growth.
- Over 60% of instances with $N > 100$ exceeded the 18-second time constraint.

Identified Limitations:

- Time complexity: $O(N!)$ in the worst case for exact methods.
- Memory consumption: Exponential growth with problem size.
- Solution feasibility: Poor performance on large instances.

Phase 2: Three-Tier Partitioning Strategy

Rationale for Redesign: Performance analysis highlighted the need for a size-adaptive algorithm to balance solution quality and computational efficiency.

Tier Structure Implementation:

- **Tier 1 ($N \leq 50$):** Retained OR-Tools with a reduced time limit of 10 seconds.
- **Tier 2 ($50 < N \leq 200$):** Employed a hybrid greedy construction with local search optimization. The modified savings heuristic was:

  $$s(i,j) = d(0,i) + d(0,j) - d(i,j) + \alpha \cdot \text{balance\_factor}(i,j)$$

  where $\alpha = 0.3$ was found empirically optimal.
- **Tier 3 ($200 < N \leq 1000$):** Applied a genetic algorithm with:
  - Population size: 50
  - Generations: 100
  - Tournament selection and order crossover

Performance Improvements:

- Reduced time violations by 75%.
- Maintained solution quality within 8% of optimal for small and 15% for large instances.

Phase 3: Five-Tier Refined Partitioning

Motivation for Refinement: Analysis revealed that medium-large instances ($150 \leq N \leq 450$) performed better with genetic algorithms rather than greedy methods.

Enhanced Tier Structure:

- **Tier 1 ($N < 50$):** Maintained OR-Tools with an optimized time limit of 8 seconds.
- **Tier 2 ($50 \leq N \leq 150$):** Used balanced greedy construction with intensive local search:
  - 2-opt intra-route optimization ($O(N^2)$ per route)
  - Inter-route swaps accepted if:

    $$\max(\text{new}_i, \text{new}_j) < \max(\text{old}_i, \text{old}_j)$$
- **Tier 3 ($150 < N \leq 450$):** Genetic algorithm with:
  - Population size: 30
  - Generations: 50
  - Crossover probability: 0.8
  - Mutation probability: 0.2, biased towards load balancing
- **Tier 4 ($N > 450$):** Geographic clustering using polar transformation:

  $$r_i = d(0,i), \quad \theta_i = \arctan 2(y_i - y_0, x_i - x_0)$$

  followed by nearest neighbor optimization within clusters.

Performance Validation:

- 90% success within time constraints.
- 12% improvement in solution quality for medium instances.

- 8% improvement for large instances compared to Phase 2.

Phase 4: Ultra-Large Instance Optimization

Critical Issue: Instances with $N \approx 1000$ continued to violate time constraints in 25% of test cases.

Ultra-Fast Heuristic Development:

- `solve_ultra_large()` function included:
  - **Phase A (2 seconds):** Polar coordinate partitioning with $O(N \log N)$ complexity.
  - **Phase B (3 seconds):** Nearest neighbor improvement, with:

$$\text{max\_iterations} = \min(10, \lfloor N/50 \rfloor)$$

  - **Phase C (Remaining time):** Local search with 2-opt, bounded by:

$$\text{max\_moves} = \min(100, \lfloor N/10 \rfloor)$$

Algorithmic Innovations:

- Memory-efficient distance computation.
- Precomputed nearest neighbors (k = $\min(10, N/20)$).
- Early stopping based on improvement thresholds.

Final Performance Metrics

- 98% success rate within 18-second time limit.
- Solution quality:
  - Within 5% of optimal for $N < 50$
  - Within 12% of best-known for $50 \leq N \leq 450$
  - Within 20% of estimated optimal for $N > 450$
- Average computation time: 18.2 seconds

Validation Results: Testing on over 1000 instances showed that each phase effectively addressed performance bottlenecks while preserving or improving solution quality. The final implementation successfully handled problems from small-scale exact optimization to ultra-large heuristic approximation under strict computational constraints.

6 Experiment and Evaluation

Based on the results 2 and 3 obtained from 13 benchmark test cases, we conducted a comparative analysis of four optimization algorithms: **Tabu Search**, **Ant Colony Optimization**, **Beam Search**, and **OrTools with Genetic Algorithm (GA)**. The following conclusions were drawn regarding their relative effectiveness:

Tabu Search (TS)

Tabu Search consistently yields the highest objective values across nearly all test cases.

In large-scale instances (e.g., $N = 500$ to 1000), TS performs significantly worse, with results that are often 5 to 10 times larger than those of other methods.

The standard deviation is relatively high in early test cases (e.g., 8.52), though it becomes zero in many larger instances—indicating premature convergence to suboptimal solutions.

This suggests that while TS may offer some variability, it also lacks the capacity to escape local optima, making it the least effective algorithm in this comparison.

Ant Colony Optimization (ACO)

ACO demonstrates remarkable stability, with very low standard deviation values (often close to 0), indicating robust convergence behavior.

It consistently achieves the lowest or near-optimal objective values, especially on small and medium-sized test cases.

Even as problem size increases, ACO scales effectively and maintains high-quality solutions, outperforming other methods in most test cases.

Overall, ACO exhibits an ideal balance between solution quality and consistency, making it the most effective approach in this evaluation.

Beam Search

Beam Search maintains extreme consistency, with all standard deviations being 0, implying identical outcomes across all runs.

However, it almost always produces an objective value of 80, regardless of problem size. This suggests the algorithm lacks exploration capabilities and gets stuck at a fixed suboptimal solution.

It performs moderately well in small and medium instances but becomes less competitive as problem size increases.

While stable, Beam Search's inflexibility limits its performance in more complex scenarios.

OrTools + Genetic Algorithm (GA)

This hybrid approach performs strongly on larger test cases ($N \geq 600$), often achieving better results than Beam Search and occasionally outperforming ACO.

Early test cases show very high standard deviation (up to 100), indicating initial instability or sensitivity to parameters.

However, in large-scale problems, it delivers robust and reliable average performance, indicating better adaptation as complexity grows.

The combination of OrTools' precision and GA's search capability offers a powerful balance of exploration and exploitation.

7 Acknowledgments

This work is supported and supervised by Professor Dung P. Quang under the course of Fundamentals of Optimization.

8 Workload

Table 4: Work Assignment Table

No.	Name	Task(s) Completed
1	Tran Nguyen My Anh	Ortools and Generative Algorithm
2	Dang Anh Tu	Tabu Search
3	Nguyen Thanh Trung	Ant Colony
4	Phan Duc Trong	Beam Search

As shown in Table 4, each member was responsible for implementing a different optimization method.

Input		Tabu Search		Ant Colony		Beam Search		OrTools vs GA	
N	**K**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
6	2	10.8	1.1	10	0	12	0	14.4	11.8
6	2	10.6	0.9	10	0	12	0	14.4	11.8
10	2	35.6	4.8	10	0	38.2	2.38	35.2	12.97
100	5	161.2	22.29	33.2	0.4	56	0	145.5	16.5
100	5	166.2	18.99	33.4	0.49	56	0	56.2	81.7
200	10	194.8	12.7	30	0	45.4	7.4	158.33	7.258
500	10	1254.4	36.0	108	0.63	153.8	2.32	605	29.1
600	20	789.4	21.23	78.8	0.75	129.2	1.12	389	63.4
700	20	938.8	13.83	83.4	0.49	176.4	12.25	457.5	87.8
800	20	1048	25.38	85.4	0.49	174.2	6.8	746	0
900	20	1225.6	52.85	90.4	1.2	151.2	7.36	912	0
1000	20	1318.6	38.3	92.8	1.0	231.8	21.12	199	40.9
1000	20	1352.2	38.5	92.8	1.0	231.8	21.12	342	0

Table 2: Performance comparison of optimization algorithms

Input		Tabu Search		Ant Colony		Beam Search		OrTools vs GA	
N	**K**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
6	2	89.80	8.52	100	0	83	0	86.67	100
6	2	94.60	7.20	100	0	83	0	86.67	100
10	2	84.60	5.02	100	0	80	0	80	0
100	5	80.00	0.00	99.4	1.2	80	0	80	0
100	5	80.00	0.00	98.8	1.47	80	0	80	0
200	10	80.20	0.80	93	0	80	0	80	0
500	10	80.00	0.00	96	0.63	80	0	80	0
600	20	80.00	0.00	83.2	0.75	80	0	80	0
700	20	80.00	0.00	84.6	0.49	80	0	80	0
800	20	80.00	0.00	87.6	0.49	80	0	80	0
900	20	80.00	0.00	85.6	1.2	80	0	80	0
1000	20	80.00	0.00	100	0	80	0	80	0
1000	20	80.00	0.00	100	0	80	0	80	0

Table 3: Point of optimization algorithms