Frankfurt University
of Applied Sciences

– Faculty of Computer Science and Engineering –

Implementation and Evaluation of Approaches for Efficiently Solving Practical Vehicle Routing Problem by Using Machine Learning Algorithms

Thesis submitted in order to obtain the academic degree

Master of Science (M.Sc.)

submitted on 15. December 2023 on

Arnob Mahmud

Student ID: 1079386

First Supervisor : Prof. Dr. Doina Logofatu
Second Supervisor : Prof. Dr. Christina Anderson

Declaration

Hereby I assure that I have written the presented thesis independently and without any third party help and that I have not used any other tools or resources than the ones mentioned/referred to in the thesis.

Any parts of the thesis that have been taken from other published or yet unpublished works in terms of their wording or meaning are thoroughly marked with an indication of the source.

All figures in this thesis have been drawn by myself or are clearly attributed with a reference.

This work has not been published or presented to any other examination authority before. I am aware of the importance of the affidavit and the consequences under examination law as well as the criminal consequences of an incorrect or incomplete affidavit.

Frankfurt, 15. December 2023

Arnob Mahmud

Contents

Abstract

The research introduces various novel elements to manage the temporal dimension and proposes an effective Hybrid Genetic Search, Guided Local Search, Ant Colony Optimization, and Simulated Annealing methods of time-constrained vehicle routing problems. In terms of time window, new move evaluation approaches are proposed that allow the evaluation of movements from any classical neighborhood based on arc or node exchanges in amortized constant time, while also accounting for penalized infeasible solutions.

The most recent open-source Hybrid Genetic Search implementation for the Capacitated Vehicle Routing Problem (HGS-CVRP) now has time window support. We also added more construction heuristics, a Selective Route Exchange (SREX) crossover, and an enhanced local search process modeled after the SWAP* neighborhood. We employed several schedules to increase the neighborhood's size and population based on instance attributes.

The artificial ant colony hierarchy used in MACS-VRPTW is intended to progressively optimize the functions of multiple objectives. For example, the first colony's goal is to minimize the number of vehicles, while the second colony's goal is to minimize the distances traveled. Colonies work together by communicating information through pheromone updates, which we have implemented here.

We have also used Google OR-tools to develop the Guided Local Search algorithm and the Simulated Annealing algorithm in its raw form to observe how it operates and produces results.

For any combination of periodic, duration-constrained vehicle routing problems with time windows, the suggested algorithms surpass all existing state-of-the-art techniques to benchmark instances from the classical literature.

Keywords: Vehicle Routing Problems with Time Window (VRPTW), Hybrid Genetic Search (HGS), Guided Local Search (GLS), Ant Colony Optimization (ACO), Simulated Annealing (SA), MACS-VRPTW, Genetic Algorithm (GA), Exact, Heuristics, Metaheuristics, Machine Learning Algorithms, GRASP, Local Search (LS), Neighborhood Search, OR-Tools, VRP, VRPTW, Vehicle Routing Problems (VRP).

Chapter 1

Introduction

An essential component of a manufacturing organization is logistics. Sending goods from a business known as a depot to clients who order them is one of logistics' tasks. Although logistics is not a component that directly adds value to a product, poor logistics management can result in excessive expenses that reduce a company's capacity to compete. Therefore, it is essential to manage and carry out the shipping of goods from depots to clients in an effective and efficient manner. Effective in the sense that all client demands are completely met and the product is delivered to the consumer. Effectively indicates that the costs associated with the logistics process can be minimized. Choosing the most cost-effective shipping routes from the depot to customers and from one customer to another is one of the tasks that can accomplish this. The Vehicle Routing Problem (VRP) is this activity.

There are several types of combinatorial optimization problems, vehicle routing problem (VRP) is one class among them, as well as integer programming problems. The first edition was published by Dantzig and Ramser in 1959 [39], it was defined under the name Truck Dispatching Problem, following which it has been studied extensively. Although the problem may be simple to define, it may be challenging to resolve. The price of distribution, logistics, and transportation may all be significantly impacted. The objective of the task is to serve several customers with a fleet of vehicles at the lowest possible cost. The expense may include the time it takes to travel or serve consumers. By introducing different limitations, one can create new variations of the issue.

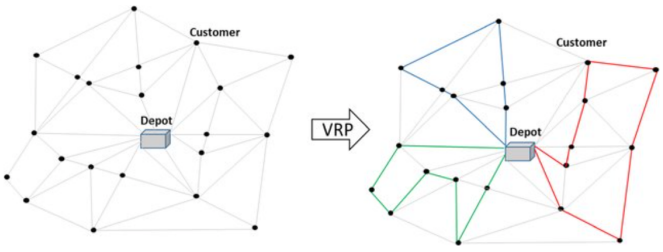


Figure 1.1: An example of vehicle routing problem (VRP) [5].

Vehicle Routing Problem (VRP) and its variations fall under the category of NP-hard problems [1] [230] [90] [160]. At the moment of the potential computations, the VRP in this situation cannot be solved precisely. The NP-Hard case can be resolved by combining the heuristic method that estimates the answer with an acceptable completion time with the optimal solution strategy. Therefore, heuristic or metaheuristic methods can be used to solve VRP problems [237].

Regarding in Chapter 2, we are going to mathematically describe the combinatorial optimization problems, VRP definition, formulation and classification of various type of constrains. In Chapter 3, we will discuss about the thesis goals, research questionnaires and scenarios. In Chapter 4, various evolutionary algorithms have been briefly discussed for solving VRP. In Chapter 5, the implementation and evaluation of the various VRPTW methods have been addressed theoretically. In Chapter 6, we will represent the results, in Chapter 7, It will be about the summary, and all of the model's programming formats have been shown in the Chapter Appendix A.

Chapter 2

Background

2.1 Combinatorial Optimization Problems Formulation

An objective function, a collection of constraints, and a set of choice variables are all components of optimization problems. This is how it is explained: Find values for the decision variables that satisfy the set of constraints to reduce or maximize the objective function.
Assuming that X is a feasible set, S is a solution space, and f is an objective function [199]. The formulation of the optimization problem is as follows:

$$min\ \{f(x) \mid x \in X, X \subseteq S\} \tag{2.1.1}$$

It is referred to be a reasonable solution if $x \in X$. It is not a feasible approach in any other case. Optimization models can often be divided into several categories. We take into consideration the categorization based on variable type: continuous optimization, and discrete optimization issues. Combinatorial Optimization Problems (COP) result if the cardinality of the solution space is finite. It is commonly known that they are simple to express but difficult to solve [94]. COPs can also be expressed as programming problems in mathematics.

Mathematical Programming Problems are described as follows:

$$minf(x) \tag{2.1.2}$$

subject to

$$g_i(x) \leq 0, i = 1, \ldots, m \tag{2.1.3}$$

$$x_j \geq 0, j = 1, \ldots, n \tag{2.1.4}$$

Where $f(x) : R^n \to R$ and X belongs to R^n or S that satisfies $\mid S \mid < \infty$.

A mathematical model known as a $linear\ program$ (LP) seeks a set of non-negative values for variables that maximize or minimize a linear objective function while satisfying a number of linear constraints. A mixed integer linear program (MIP) is an LP with some integer variables. It is referred to as an integer linear program (IP) if all the variables are integers. In other words, we get a linear program (LP) if there are no integer variables, and a pure (IP) if there are no continuous variables. When both integer and continuous variables are present, mixed IP (MIP) is the result [69]. In general, it is more difficult to solve IP than LP.

This is how mixed integer linear program (MIP) is created:

$$max/min \qquad cx + dy \tag{2.1.5}$$

subject to

$$Ax + Dy \leq b \tag{2.1.6}$$

$$x \geq 0, y \geq 0 \tag{2.1.7}$$

$$x\ integer, \tag{2.1.8}$$

where $c = (c_1, \ldots, c_n)$, $d = (d_1, \ldots, d_n)$, $A = (a_ij)_{m \times n}$, $D = (d_ij)_{m \times n}$, $b = (b_1, \ldots, b_m)^T$, integer variables $x = (x_1, \ldots, x_n)^T$ and continuous variables $y = (y_1, \ldots, y_n)^T$.

One of the most well-known models in combinatorial optimization is the integer program, which is defined as follows [69]:

$$max/min \qquad cx \tag{2.1.9}$$

subject to

$$Ax \leq b \tag{2.1.10}$$

$$x \geq 0 \tag{2.1.11}$$

$$x\ integer, \tag{2.1.12}$$

Finding the maximum or least value of the objective function while taking into consideration a number of linear constraints is the definition of a linear programming model in terms of mathematics. Find the values of the n choice variables x_i to maximize or reduce the objective function z, or put it another way. In the following format, where c_i, a_{ij}, and b_i are constants:

$$max/min \qquad z = \sum_{i=1}^{n} c_i x_i \tag{2.1.13}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad \forall\ i = 1, \ldots, m \tag{2.1.14}$$

$$x_i \geq 0 \qquad \forall\ i = 1, \ldots, n \tag{2.1.15}$$

The linear programming issue has feasible solutions when the decision variables satisfy the linear constraints. The optimal solution is a feasible solution that optimizes the goal function, while the feasible region comprises all feasible solutions.

Vehicle routing problem (VRP), which is regarded as a pure integer linear program (IP), is an illustration of a real-world combinatorial optimization problem. The goal of this thesis is to identify the best or most accurate solution to a certain class of VRPs.

2.2 Vehicle Routing Problem

The Vehicle Routing Problem consists of a set of customers with known demands. There is often only one depot (multiple depots may appear for different type of constraints). Each vehicle has a specific amount of cargo space and a set maximum distance it can travel. Every vehicle departs from a depot, delivers the requested products, and then drives back to the depot. Every customer is given a single vehicle route (multiple route may appear for different type of constraints). Any route's overall demand cannot be greater than the vehicle's carrying capacity.

The goal of the vehicle routing problem is to deliver goods to a group of consumers whose wants are known using vehicle routes that are as minimal as possible that start and end at a depot.

2.2.1 Vehicle Routing Problem Formulation

Since we are aware that the VRP is a combinatorial optimization issue, we use graph theory to model it. We will simply simulate the traditional VRP as shown by Tan et al. [206].

Mathematical framework:

- Let $G = (V, A)$ be a graph, where $V = \{v_0, v_1, v_2, \ldots, v_N\}$, where v_0 is the depot and $\{v_0, v_1, v_2, \ldots, v_N\}$ is the node set indicating clients to be served.
- Each client has a demand D_i.
- $A = \{(v_i, v_j) : v_i, v_j \in V\}$ denote as the arc set (subscript indicates sequence) connecting nodes i and j with a distance d_{ij}.
- Let $M_m = \{m_1, m_2, \ldots, m_m\}$ denote as the vehicle set, where each vehicle in the set has a maximum load capacity cap_m which means the total load of vehicle m cannot exceed the maximum load capacity cap_m.

The goal of the VRP:

- Finding the optimal vehicle routes so that each customer is only visited once by one vehicle and that each vehicle begins and ends its route at the depot.

The classical VRP assumptions that were used in this instance:

- There is no demand at the depot.
- Only one vehicle provides service to each customer location.
- The requirement of every client is one and the same.
- Each vehicle must not carry more weight than it is capable of carrying.
- At the depot, each vehicle begins and ends its route.
- Information on customer demand, customer distribution distances, and delivery prices is available.

Notations used in this formulation:

- V: Node set, in which v_0 is the depot and $\{v_1, v_2, \ldots, v_N\}$ are customers
- i, j: Subscripts of the customer nodes, where $i, j = 1, 2, \ldots, N$
- $A = \{(v_i, v_j) : v_i, v_j \in V\}$ is arcs set connecting nodes i and j
- M_m: Set of vehicles with type m

- D_i: Demand of customer i
- d_{ij}: Distance between node i and j
- veh_m: Maximum number of each type of vehicle available
- cap_m: Maximum load capacity of type m vehicles
- fc_m: Fixed cost of vehicle type m
- vc_m: Variable cost of m type vehicle
- Dm_{ij}, m: Amount transported from i to j using vehicle type m
- X_{ij}, m: Value of one if a vehicle of type m moves from node i to node j, If not, a value of zero

Fitness or objective function:

- Total cost = Fixed cost + Variable cost

$$Minimize \sum_{m=1}^{M} \sum_{i=0}^{N} \sum_{j=0}^{N} X_{ij,m} \times fc_m + \sum_{m=1}^{M} \sum_{i=0}^{N} \sum_{j=0}^{N} d_{ij} \times D_{ij,m} \times vc_m \tag{2.2.1}$$

Constraint 1 (Routing):

- Every vehicle needs to go back to the depot, where the subscript is 0.

$$\sum_{i=1}^{N} X_{i0,m} = 1 \, \forall \, m \in M_m \tag{2.2.2}$$

Constraint 2 (Routing):

- In a route, each node can only be visited once.

$$\sum_{m=1}^{M} \sum_{i=1}^{N} \sum_{j=1}^{N} X_{ij,m} = 1 \, \forall \, (i,j) \in A \, \forall \, m \in M_m \tag{2.2.3}$$

Constraint 3 (Routing):

- In order to maintain route continuity, a vehicle must leave a node after arriving at it.

$$\sum_{m=1}^{M} \sum_{i=1}^{N} X_{ip,m} = \sum_{m=1}^{M} \sum_{i=1}^{N} X_{pi,m} \, \forall \, p \in V \tag{2.2.4}$$

Constraint 4 (Demand):

- They set limitations on the levels of demand.

$$\sum_{i=1}^{N} \sum_{j=1}^{N} X_{ij,m} \times D_j = Dm_{ij,m} \, \forall \, (i,j) \in A \, \forall \, m \in M_m \tag{2.2.5}$$

Constraint 5 (Capacities):

- They place limitations on the capacities.

$$\sum_{i=1}^{N} Dm_{0i,m} \leq cap_m \, \forall \, m \in M_m \tag{2.2.6}$$

Constraint 6 (Demand and Capacities):

- There are almost as many vehicles as are readily available.

$$\sum_{m=1}^{M} X_{ij,m} \leq veh_m \, \forall \, m \in M_m \tag{2.2.7}$$

Other VRP variant modeling necessitates some adjustments to the limitations or fitness function. When solving a multi-objective VRP problem, we simultaneously optimize a lot of different variables.

After learning how to formulate VRP, we will describe how the number of VRP variations has increased from Dantzig et al. [39] to numerous VRP variations and what makes each one unique.

2.2.2 Vehicle Routing Problem Classification

Numerous researchers have investigated different VRP models, primarily motivated by the substantial potential for applications as well as the limits of these models.

Applications in the real world include mail delivery, solid waste collection, street cleaning, commodity distribution, telecommunications network design, transportation networks, school bus routing, dial-a-ride services, transportation of people with disabilities, and scheduling of sales representatives and maintenance crews.

There are numerous classification principles for VRPs, including those that are based on restrictions (constrained and unconstrained) and data (static and dynamic).

2.2.2.1 Unconstrained Vehicle Routing Problems

The unconstrained (VRP) is described as creating a set of tours at the lowest possible cost to service a group of consumers (or cities) using a number of vehicles that meet the requirements listed below [199]:

- Customer Constraints: Because of customer restrictions, Only one vehicle visits or serves each customer. If the distance matrix fulfills the triangle inequality, then the possibility that some consumers will be visited more than once is accepted.
- Vehicle Constraints: Due to vehicle restrictions, all trips begin and conclude at the depot.

Let $G = (V, A)$ be a full graph, where V is a collection of vertices defined as follows:

- $V = 0 \cup N$ wherein N is the number of customers (or cities), 0 designates the depot, and A is the set of arcs.
- Each arc (i, j) refers to a non-negative value c_{ij}, which is a distance (travel expense or journey time) between vertex i and vertex j.
- C is asymmetric if the distance from vertex i to vertex j differs from the distance from vertex j to vertex i. If not, C is symmetric and a collection of arcs (A) is swapped out for a collection of edges (E).
- The number of automobiles that can be fixed or given away is assumed to be m.

The decision binary variable x_{ij} is referred as follows:

$$x_{ij} = \begin{cases} 1 & \text{if the optimal solution, where i}\neq\text{j is represented by the arc (i, j)}\in\text{A;} \\ 0 & \text{Otherwise.} \end{cases} \tag{2.2.8}$$

Unconstrained VRP is formulated as follows [169]:

$$Min \qquad \sum_{i \neq j} C_{ij} X_{ij} \tag{2.2.9}$$

subject to

$$\sum_{j \in V} x_{ij} = 1 \qquad \forall i \in V \setminus \{0\} \tag{2.2.10}$$

$$\sum_{i \in V} x_{ij} = 1 \qquad \forall j \in V \setminus \{0\} \tag{2.2.11}$$

$$\sum_{i \in N} x_{i0} = m \tag{2.2.12}$$

$$\sum_{j \in N} x_{0j} = m \tag{2.2.13}$$

$$\sum_{i,j \in S} x_{ij} \leq \mid S \mid - r(S) \qquad \forall S \subseteq V \setminus \{0\},\ S \neq \emptyset \tag{2.2.14}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V \tag{2.2.15}$$

Constraints (2.2.10) and (2.2.11) are the in-degree as well as out-degree for each vertex, respectively, ensuring that each vertex is only visited once. Constraints (2.2.12) and (2.2.13) are the in-degree as well as out-degree of the depot, respectively. Constraints (2.2.14) and (2.2.15) are the integrality constraint and subtour elimination, respectively; where $r(S)$ is the bare minimum of vehicles needed to go between each vertex in S.

2.2.2.2 Constrained Vehicle Routing Problems

Constrained VRP is described as the creation of a set of tours with the lowest possible cost to service a group of customers utilizing a fleet of cars while satisfying the constraints of the customers, the vehicles, and other constraints. There are several kinds of limitations, and thus, various kinds of constrained VRPs [199]:

- capacitated vehicle routing problem (CVRP).
- distance-constrained vehicle routing problem (DVRP).
- distance-constrained capacitated VRP (DCVRP).
- vehicle routing problem with time windows (VRPTW).

- vehicle routing problem with backhauls (VRPB).
- vehicle routing problem with pickup and delivery (VRPPD)

Capacitated Vehicle Routing Problem (CVRP)

It is believed that CVRP is the most straightforward and extensively researched variant of VRP.

- Let $G = (V, A)$ be a fully directed graph, $V = \{0\} \cup N$ given that N is the set of customers, m identical vehicles have capacity Q, 0 is the central depot, and A is the collection of arcs. The travel expense (or distance) between vertex i and vertex j is denoted by the symbol c_{ij}. Vertex i's demand is d_i, where $Q \geq d_i \geq 0$ for each of $i = 1, \ldots, n$ and $d_0 = 0$.

The CVRP consists of a central depot, m identical trucks, each with a capacity of Q, a set of clients N, each with a demand d_i. The goal of CVRP is to service all clients with the least amount of total distance (or expense), provided that:

- Customer constraints: Customers are limited to being visited (served) only once per vehicle.
- Transportation restrictions: All tours begin and end at the depot.
- Capacity constraints: Each vehicle's load is never greater than or equal to its capacity.

Distance-constrained Vehicle Routing Problem (DVRP)

A distance-constrained vehicle routing problem (DVRP), which is likewise NP-hard, results if the capacity restriction is swapped out for the distance constraint. The following is its definition [169]: To connect the depot to n clients using m cars, determine the optimal set of trips with the shortest distance traveled. Such that:

- Customer constraints: Customers are restricted in that each is only visited once.
- Vehicle restrictions: Each vehicle begins and ends its journey at the depot.
- Distance restrictions: It requires that each vehicle in the solution travels a total distance that is less than or equal to the distance that can be traveled.

If the distance from vertex i to vertex j differs from the distance from vertex j to vertex i, the DVRP is asymmetric. If not, the symmetric DVRP is established. It is referred to as Distance-constrained CVRP (DCVRP) when the VRP problem contains both capacity and distance limits.

The CVRP (DVRP, or DCVRP) will be comparable to the TSP if we just have one vehicle with unlimited capacity or no distance restriction. In this situation, the best course of action would be to find a single trip that accommodated all consumers for the least amount of cost or distance [169].

Vehicle Routing Problem with Time Windows (VRPTW)

With a time window constraint, VRPTW is seen as an extension of CVRP and thus an NP-hard issue. To service a group of consumers within a time limit, it is described as optimizing a set of least-cost tours utilizing a group of vehicles that match the following criteria:

- Customer constraints: One vehicle only visits or serves each customer once. The triangle inequality can be satisfied if the distance matrix, in which case a customer could be crossed more than once.
- Vehicle constraints: All trips begin and conclude at the depot.
- Capacity constraints: Each vehicle's load must be less than or equal to its maximum capacity.
- Service time constraints: Each client must be attended to within a set window of time $[a_i, b_i]$. In other words, serving consumers before or after a time frame interval is unacceptable, and if a vehicle arrives before the start time, it must wait [170].

Let $G = (V, A)$ be a fully directed graph with $V = \{0, n+1\} \cup N$ and $N = \{1, \ldots, n\}$ being a set of customers; $0;\ n+1$ being the first and last depots; A being the set of arcs, and k being identical vehicles. Each arc $(i, j) \in A$ travels for time t_{ij}. Each vertex i has a service time s_i and a time frame $[a_i, b_i]$, that it must be served within the time window $[a_i, b_i]$.

The decision variable x_{ijk} is utilized in the formulation of VRPTW and is defined as follows:

$$x_{ijk} = \begin{cases} 1 & \text{if (i,j)} \in \text{A is used by vehicle k} \in \text{K;} \\ 0 & \text{Otherwise.} \end{cases} \tag{2.2.16}$$

- The variable of time when a vehicle $k \in K$ is serving a vertex $i \in V$, w_{ik} indicates the beginning of service.
- The start depot and end depot time windows are $[a_0, b_0] = [a_{n+1}, b_{n+1}] = [E, L]$ where E, L are the earliest departure from and the latest arrival from the start depot, respectively.
- These vertices have zero demand (0) and zero service time $(n+1)$, respectively.
- Let $\Delta^-(i)$ represent the set of vertices directly before i such that $(j, i) \in A$, and let $\Delta^+(i)$ represent the set of vertices immediately after i such that $(i, j) \in A$.

The following is the multicommodity network flow formulation to VRPTW using the same notation as in [170]:

$$min \quad \sum_{k \in K} \sum_{(i,j) \in A} C_{ij} X_{ijk} \tag{2.2.17}$$

subject to

$$\sum_{k \in K} \sum_{j \in \Delta^+(i)} x_{ijk} = 1 \qquad \forall\, i \in N \tag{2.2.18}$$

$$\sum_{j \in \Delta^+(0)} x_{0jk} = 1 \qquad \forall\, k \in K \tag{2.2.19}$$

$$\sum_{i \in \Delta^-(j)} x_{ijk} - \sum_{i \in \Delta^+(j)} x_{ijk} = 0 \qquad \forall\, k \in K,\ j \in N \tag{2.2.20}$$

$$\sum_{i \in \Delta^-(n+1)} x_{i,n+1,k} = 1 \qquad \forall\, k \in K \tag{2.2.21}$$

$$x_{ijk}(w_{ik} + s_i + t_{ij} - w_{jk}) \leq 0 \qquad \forall\, k \in K,\ (i, j) \in A \tag{2.2.22}$$

$$a_i \sum_{j \in \Delta^+(i)} x_{ijk} \leq w_{ik} \leq b_i \sum_{j \in \Delta^+(i)} x_{ijk} \qquad \forall\, k \in K,\ i \in N \tag{2.2.23}$$

$$E \leq w_{ik} \leq L \qquad \forall\, k \in K,\ i \in \{0, n+1\} \tag{2.2.24}$$

$$\sum_{i \in N} d_i \sum_{j \in \Delta^+(i)} x_{ijk} \leq C \qquad \forall\, k \in K \tag{2.2.25}$$

$$x_{ijk} \geq 0 \qquad \forall\, k \in\ K,\ (i,j) \in\ A \tag{2.2.26}$$

$$x_{ijk} \leq \{0,1\} \qquad \forall\, k \in\ K,\ (i,j) \in\ A \tag{2.2.27}$$

In the above nonlinear formulation, the constraint (2.2.18) shows that only one vehicle visits each vertex; the constraints (2.2.19 - 2.2.21) represent the flow along the path taken by vehicle k; the constraints (2.2.22 - 2.2.24) are time restrictions; the constraint (2.2.25) represents as capacity constraint and (2.2.26) represents as non-negativity constraint; and the final constraint (2.2.27) is that related to integrality constraint.

Vehicle Routing Problem with Backhauls (VRPB)

Another addition to CVRP is VRPB, which is NP-hard as well. The group of customers must be divided into two subgroups in order to establish VRPB: the first group consists of linehaul customers, who need the product delivered. Backhaul customers are those who need the merchandise picked up; they make up the other group of customers. Customers from both sets must serve themselves first if the tour includes both sets of customers. It should be noted that some formulas do not permit trips with only backhaul consumers [171].

The following are the restrictions for this issue [171]:

- Customer constraints: One vehicle visits (serves) each customer once. A client could cross more than once if the distance matrix meets the triangle inequality.
- Vehicle constraints: The depot is where all trips begin and terminate.
- Capacity constraints: Each vehicle's load is always less than or equal to C.
- In any trip, linehaul customers must be attended to before backhaul clients.

It is okay to have a tour with a single linehaul customer in the following formulation, which is also appropriate for asymmetric VRPB, however it is not allowed to have a tour with backhaul customers only.

- Let $G = (V, A)$ be a full directed graph; where $V = \{0\} \cup L \cup B$ in which 0 stands for the depot; $L = \{1, 2, \ldots, n\}$ represents a subset of linehaul clients and $B = \{n+1, \ldots, n+m\}$ represents a subset of backhaul clients; d_j represents the demand of every single client for delivery or collection; K represents the number of vehicles, every single vehicle has capacity C; and c_{ij} represents the cost of arc (i, j).
- Suppose $L_0 = L \cup \{0\}$ and $B_0 = B \cup \{0\}$ If $\bar{V} = V$, and $\bar{A} = A_1 \cup A_2 \cup A_3$, then let $\bar{G} = (\bar{V}, \bar{A})$ be a full directed graph. Arcs between depot and linehaul customers to linehaul clients are represented by $A_1 = \{(i,j) \in A : i \in L_0, j \in L\}$; and $A_2 = \{(i,j) \in A : i \in B, j \in B_0\}$ depicts the arcs between the depot and the backhaul consumers; and arcs from linehaul customers to backhaul clients and depot are shown as $A_3 = \{(i,j) \in A : i \in L, j \in B_0\}$.
- Let all customer subsets in L and B be represented by ζ and β. Also let $F = \zeta \cup \beta$; Let $r(S)$ be the least number of vehicles needed to serve all clients in S where $S \in F$. Let $\Delta^+(i)$ denotes the set of vertices which come directly after i such a way that $(i,j) \in A$ and $\Delta^-(i)$ denotes the set of vertices which come directly before i such a way that $(j,i) \in A$.

The following is how the decision variable x_{ij} is used and defined:

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is in the optimal solution;} \\ 0 & \text{Otherwise.} \end{cases} \tag{2.2.28}$$

The following is the formulation for VRPB [171]:

$$min \quad \sum_{(i,j)\in\bar{A}} c_{ij}x_{ij} \tag{2.2.29}$$

subject to

$$\sum_{i\in\Delta^-(j)} x_{ij} = 1 \qquad \forall\, j \in \bar{V} \setminus \{0\} \tag{2.2.30}$$

$$\sum_{j\in\Delta^+(i)} x_{ij} = 1 \qquad \forall\, i \in \bar{V} \setminus \{0\} \tag{2.2.31}$$

$$\sum_{i\in\Delta^-(0)} x_{i0} = K \tag{2.2.32}$$

$$\sum_{j\in\Delta^+(0)} x_{0j} = K \tag{2.2.33}$$

$$\sum_{j\in S} \sum_{i\in\Delta^-(j)\setminus S} x_{ij} \geq r(S) \qquad \forall\, S \in F \tag{2.2.34}$$

$$\sum_{i\in S} \sum_{j\in\Delta^+(i)\setminus S} x_{ij} \geq r(S) \qquad \forall\, S \in F \tag{2.2.35}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, (i,j) \in \bar{A} \tag{2.2.36}$$

The in-degree and out-degree for every client are the constraints (2.2.30, 2.2.31), meaning that each customer is only visited once; In-degree and out-degree constraints for the depot are represented by constraints (2.2.32, 2.2.33); connectivity and capacity constraints are represented by constraints (2.2.34, 2.2.35); and the integrality constraint is represented by constraint (2.2.36).

Vehicle Routing Problem with Pickup and Delivery (VRPPD)

Each customer i makes two demands in the simplest form of VRPPD; d_i must be delivered, and p_i must be picked up. Additionally, we must add two new variables for each client i; where O_i, which represents the vertex at which the source of the shipment occurs, and D_i which represents the client's location at which the destination of the pick-up exists. The basic VRPPD is subject to the following constraints [172], assuming that delivery occurs before pickup at each customer:

- Customer constraints: One vehicle visits (serves) each customer once. The triangle inequality can be satisfied if the distance matrix, in which case a customer could be visited more than once.
- Vehicle constraints: The depot is where all trips begin and terminate.
- Capacity constraints: Each vehicle's load needs to be constantly met and must be less than or equal to Q.
- If both O_i and D_i are not present at the depot, each client i must be served before customer D_i and after customer O_i during the same tour.

It is obvious that clients who must demand only to be picked up must be served before other clients, including those who must demand only to be delivered. Three variables are used in the VRPPD formulation:

1. The variable of time T_{ik} uses vehicle k to determine when the service for customer i begins.
2. The load element When the service for the customer i is complete, L_{ik} assesses the load of the vehicle k.
3. The following is a definition of the decision variable x_{ijk}:

$$x_{ijk} = \begin{cases} 1 & \text{if } (i,j) \in A_k \text{ is used by vehicle } k \in K; \\ 0 & \text{Otherwise.} \end{cases} \tag{2.2.37}$$

- Let $N = P \cup D$; where P be the set of pick-up vertices and D the set of delivery vertices, where $P = \{1, \ldots, n\}$ and $D = \{n+1, \ldots, 2n\}$ respectively. Let $l_i = d_i$ and $l_{n+i} = -d_i$ and assume that vertex i needs d_i to pick up and deliver to vertex $n+i$.
- Let K represent the collection of vehicles, each serving a set of vertices $N_k = P_k \cup D_k$ and having a capacity of C_k, since N_k, P_k and D_k are subsets of N, P and D. There is a network for each vehicle $G_k = (V_k, A_k)$ where the source and destination depots for each vehicle k is represented by a set of vertices called $V_k = N_k \cup \{o(k), d(k)\}$. Vehicle k has the travel time t_{ijk} and cost of c_{ijk} between two vertices, respectively.

The following is the formulation of VRPPD [172]:

$$min \quad \sum_{k \in K} \sum_{(i,j) \in A_k} c_{ijk} x_{ijk} \tag{2.2.38}$$

subject to

$$\sum_{k \in K} \sum_{j \in N_k \cup \{d(k)\}} x_{ijk} = 1 \qquad \forall\, i \in P \tag{2.2.39}$$

$$\sum_{j \in N_k} x_{ijk} - \sum_{j \in N_k} x_{j,n+i,k} = 0 \qquad \forall\, k \in K, i \in P_k \tag{2.2.40}$$

$$\sum_{j \in P_k \cup \{d(k)\}} x_{o(k),j,k} = 1 \qquad \forall\, k \in K \tag{2.2.41}$$

$$\sum_{i \in N_k \cup \{o(k)\}} x_{ijk} - \sum_{i \in N_k \cup \{d(k)\}} x_{jik} = 0 \qquad \forall\, k \in K, j \in N_k \tag{2.2.42}$$

$$\sum_{i \in D_k \cup \{o(k)\}} x_{i,d(k),k} = 1 \qquad \forall\, k \in K \tag{2.2.43}$$

$$x_{ijk}(T_{ik} + s_i + t_{ijk} - T_{jk}) \leq 0 \qquad \forall\, k \in K, (i,j) \in A_k \tag{2.2.44}$$

$$a_i \leq T_{ik} \leq b_i \qquad \forall\, k \in K, i \in V_k \tag{2.2.45}$$

$$T_{ik} + t_{i,n+i,k} \leq T_{n+i,k} \qquad \forall\, k \in\ K, i \in\ P_k \tag{2.2.46}$$

$$x_{ijk}(L_{ik} + l_j - L_{jk}) = 0 \qquad \forall\, k \in\ K, (i,j) \in\ A_k \tag{2.2.47}$$

$$l_i \leq L_{ik} \leq C_k \qquad \forall\, k \in\ K, i \in\ P_k \tag{2.2.48}$$

$$0 \leq L_{n+i,k} \leq C_k - l_i \qquad \forall\, k \in\ K, n+i \in\ D_k \tag{2.2.49}$$

$$L_{o(k),k} = 0 \qquad \forall\, k \in\ K \tag{2.2.50}$$

$$x_{ijk} \geq 0 \qquad \forall\, k \in\ K, (i,j) \in\ A_k \tag{2.2.51}$$

$$x_{ijk} \in \{0,1\} \qquad \forall\, k \in\ K, (i,j) \in\ A_k \tag{2.2.52}$$

Constraints (2.2.39, 2.2.40) represent each vertex requirement (pick up or delivery) must be fulfilled once by the same vehicle. constraints (2.2.41 - 2.2.43) represent each vehicle must begin at its origin depot $o(k)$ and end at its destination depot $d(k)$. The compatibility of the needs between tours and schedules is due to nonlinear constraints (2.2.44). The time window constraint is represented by constraint (2.2.45), where $[a_i, b_i]$ denotes the vertex i's time window interval. constraint (2.2.46) represent each vehicle must visit the pickup vertex before the delivery vertex. The appropriateness of the requirements between tours and vehicle loads is due to nonlinear constraints (2.2.47). The vehicle capacity interval at the pickup vertex and delivery vertex for each vehicle is represented by constraints (2.2.48, 2.2.49). For each vehicle, the initial load is represented by constraint (2.2.50). Constraint (2.2.51) is binary constraint and constraint (2.2.52) is non negativity constraint.

There are frequently side constraints because of additional limits in various real-world vehicle routing difficulties. The earliest and most-researched variations are:

- **Multiple Depot Vehicle Routing Problem (MDVRP)** The vendor supplies consumers from a number of depots.
- **Vehicle Routing Problem with Pick-up and Delivering (VRPPD)** Some goods may be returned by customers to the depot.
- **Split Delivery Vehicle Routing Problem (SDVRP)** Different vehicles might be used to service the customers.
- **Stochastic Vehicle Routing Problem (SVRP)** There are random demands, service times, and/or travel times.
- **Satellite Facilities Vehicle Routing Problem (VRPSF)** Vehicle refueling takes place at satellite facilities along a route.
- **Vehicle Routing Problem with Backhauls (VRPB)** A VRP where consumers can request or return certain goods.
- **Dynamic** - Some customers are added in-the-moment who are unknown in advance.

- **Heterogeneous** - Different vehicle types and capacities are available.
- **Periodic** - We need to cut and solve the problem in D days. We have a list of customers to serve and window of opportunity.
- **Capacitated Vehicle Routing Problem (CVRP)** is a capacity constraint-extended version of the VRP. Every truck must have a consistent capacity, and it must meet the demands that are set for each customer.
- **Vehicle Routing Problem with Time Windows (VRPTW)** A CVRP variation with added time restrictions is called VRPTW. We refer to the intervals $[a, b]$ that are allotted to the depot and each client as a period of time window and service time. The VRPTW has two sub-variants: one with soft time restrictions and the other with hard time windows. In the hard time window variant, vehicles must leave the depot after it opens, must serve customers during their time windows (they may arrive earlier and wait for the depot to open, but clients must be served before the depot closes), and must arrive back at the depot before it closes. Vehicles are allowed to come a little bit later in the version with soft time windows, but at a cost.

More variants: the vehicles' capacity [76] (CVRP or Capacitated VRP, HFVRP or Heterogeneous Fleet VRP), and the necessity of returning to the depot [54]. (OVRP or Open VRP), VRP dimensions [53], (2D-VRP or 3D-VRP), MEVRP or Multi-Echelon VRP), and VRP loading capacity [207] Priority of customer trust (TTVRP or Truck and Trailer VRP) [45] Consistent VRP (ConsVRP), VRP (DVRP or Dynamic VRP) Time Changes, Environmental and Pollution Regulation [28] (EVRP or Electric VRP, HVRP or Hybrid VRP, PRP or Pollution Routing Problem, GVRP or Green VRP, TDVRP or Time-Dependent VRP), Travel Time [58], and Routes across a planning horizon [210]) (RVRP or rich VRP), several VRP variations and limitations simultaneously [188], and so forth.

2.2.3 Taxonomy of Vehicle Routing Problem

The taxonomy that is being applied is a modified version of the one that Eksioglu et al. (2009) presented [26]. The five main features (research type, problem physical characteristics, scenario characteristics, information characteristics, along with data characteristics) are distinguished, and each has its own specific categories and subcategories [114]. It is displayed here in Table [2.2.3]:

1. Type of study
   - 1.1. Theory
   - 1.2. Applied methods
     - 1.2.1. Exact methods
     - 1.2.2. Classical heuristics
     - 1.2.3. Meta heuristics
     - 1.2.4. Simulation
     - 1.2.5. Real-time solution methods
   - 1.3. Implementation documented
   - 1.4. Survey, review or meta-research
2. Scenario Characteristics
   - 2.1. Number of stops on route
     - 2.1.1. Known (deterministic)
     - 2.1.2. Partially known, partially probabilistic
   - 2.2. Load splitting constraint
     - 2.2.1. Splitting allowed
     - 2.2.2. Splitting not allowed
   - 2.3. Customer service demand quantity
     - 2.3.1. Deterministic
     - 2.3.2. Stochastic
     - 2.3.3. Unknown
   - 2.4. Request times of new customers
     - 2.4.1. Deterministic
     - 2.4.2. Stochastic
     - 2.4.3. Unknown
   - 2.5. Onsite service/waiting times
     - 2.5.1. Deterministic
     - 2.5.2. Dependent
     - 2.5.3. Stochastic
     - 2.5.4. Unknown
   - 2.6. Time window structure
     - 2.6.1. Soft time windows
     - 2.6.2. Strict time windows
     - 2.6.3. Mix of both
   - 2.7. Time horizon
     - 2.7.1. Single period
     - 2.7.2. Multi-period

- 2.8. Backhauls
  - 2.8.1. Nodes request simultaneous pickups and deliveries
  - 2.8.2. Nodes request either linehaul or backhaul service, but not both
- 2.9. Node/Arc covering constraints
  - 2.9.1. Precedence and coupling constraints
  - 2.9.2. Subset covering constraints
  - 2.9.3. Recourse allowed

3. Problem Physical Characteristics
   - 3.1. Transportation network design
     - 3.1.1. Directed network
     - 3.1.2. Undirected network
   - 3.2. Location of addresses (customers)
     - 3.2.1. Customer on nodes
     - 3.2.2. Arc routing instances
   - 3.3. Number of points of origin
     - 3.3.1. Single origin
     - 3.3.2. Multiple origin
   - 3.4. Number of points of loading/unloading facilities (depot)
     - 3.4.1. Single depot
     - 3.4.2. Multiple depots
   - 3.5. Time window type
     - 3.5.1. Restriction on customers
     - 3.5.2. Restriction on depot/hubs
     - 3.5.3. Restriction on drivers/vehicle
   - 3.6. Number of vehicles
     - 3.6.1. Single vehicle
     - 3.6.2. Limited number of vehicles
     - 3.6.3. Unlimited number of vehicles
   - 3.7. Capacity consideration
     - 3.7.1. Capacitated vehicles
     - 3.7.2. Uncapacitated vehicles
   - 3.8. Vehicle homogeneity (Capacity)
     - 3.8.1. Similar vehicles
     - 3.8.2. Load-specific vehicles
     - 3.8.3. Heterogeneous vehicles
     - 3.8.4. Customer-specific vehicles
   - 3.9. Travel time
     - 3.9.1. Deterministic
     - 3.9.2. Function dependent (a function of current time)
     - 3.9.3. Stochastic
     - 3.9.4. Unknown
   - 3.10. Objective
     - 3.10.1. Travel time dependent
     - 3.10.2. Distance dependent
     - 3.10.3. Vehicle dependent
     - 3.10.4. Function of lateness
     - 3.10.5. Implied hazard/risk related
     - 3.10.6. Other

4. Information Characteristics
   - 4.1. Evolution of information
     - 4.1.1. Static
     - 4.1.2. Partially dynamic
   - 4.2. Quality of information
     - 4.2.1. Known (Deterministic)
     - 4.2.2. Stochastic
     - 4.2.3. Forecast
     - 4.2.4. Unknown (Real-time)
   - 4.3. Availability of information
     - 4.3.1. Local
     - 4.3.2. Global
   - 4.4. Processing of information
     - 4.4.1. Centralized
     - 4.4.2. Decentralized

5. Data Characteristics
   - 5.1. Data used
     - 5.1.1. Real-world data
     - 5.1.2. Synthetic data
     - 5.1.3. Both real and synthetic data
   - 5.2. No data used

Regarding applied approaches (category 1.2), we suggest making a distinction between traditional heuristics and meta-heuristics. In order to obtain better (optimal) solutions, Laporte (2009) [116] specifies classical heuristics as heuristics that prevent the intermediate solution from degrading. They frequently become stuck in local optima as a result. For instance, improvement heuristics like the λ-opt mechanism (S. Lin, 1965) [119] and construction heuristics like the savings algorithm (Clarke & Wright, 1964) [64]. On the other hand, meta-heuristics have tools to prevent becoming caught in local optima, like Simulated Annealing (Kirkpatrick, Gelatt, & Vecchi, 1983) [196] and Tabu Search (Glover, 1986) [77].

In (category 2.5), "onsite service or waiting times," it is stated how long a vehicle must wait at a customer before beginning the service or how long it takes to complete the service. This is especially important when discussing time windows. As the service time can depend on a variety of factors, including the number of people in the vehicle (Pureza, Morabito, & Reimann, 2012) [231] or the delivery quantity (Salani & Vacca, 2011) [137], we adapted the taxonomy provided by Eksioglu et al. (2009) by combining the two subcategories "time-dependent" and "vehicle type dependent" into one subcategory "dependent" in this category [26].

The time window type (category 3.5) divides articles into categories based on whether parties (clients, depots, or drivers) are subject to the window's restrictions. A road time window restriction was also included in Eksioglu et al. (2009)'s study [26]. This is not included in this study because it is uncommon. We also decided to eliminate the category "geographical location" from the taxonomy of problem physical characteristics because it is frequently ambiguous which type of instances (randomly dispersed customers, clustered customers, or a combination of these) are used to test proposed solution methods.

The quantity of vehicles that are readily available can also be used to classify VRP articles. According to the number of vehicles to be used, Eksioglu et al. (2009) [26] take into consideration three categories: exactly n vehicles, up to n vehicles, and an infinite number of vehicles. Except in the case of a single vehicle, instances in which exactly n cars should be employed are quite rare (in a multi-vehicle setting, often the option exists to not use certain vehicles when this is advantageous). As a result, we modified (category 3.6.1) to only include publications $(n = 1)$ on single-vehicle problems. Multiple vehicles may have either a constrained or an unrestricted number of slots available (categories 3.6.2 and 3.6.3, respectively). Note that, the single vehicle case is referred to as the Traveling Salesman Problem (TSP) (Applegate, Bixby, Chvatal, & Cook, 2011 [41]; Gutin & Punnen, 2007 [82]; Lawler, Lenstra, Rinnooy Kan, & Shmoys, 1985 [49]).

Finally, articles are grouped into categories in (category 3.10) based on the objective function that is chosen (either focused on journey time, distance, number of vehicles, expenses associated with delays, costs associated with risks or hazards, any other objective type, or any combination of these). The subcategory "other" was added to the original classification since some authors take into account goals that are particular to the topic being studied [114].

Chapter 3

Goals

Given a graph with a set of requests on vertices, where the vehicles are sent to fulfil these requests. This problem has been considered in various variations, the majority of which result in NP-complete issues. As a result, we must employ a variety of techniques to monitor the problem's time complexity. We will mainly focus on the Vehicle Routing Problem with Time Windows (VRPTW) variant of the issue.

The following are the objectives of the thesis:

- Find and analyze relevant publications using classical heuristics, meta-heuristics, and exact approaches.
- Implement some of these techniques, then use a test data-set to run benchmarks and compare to each other.
- Try to develop a new solving method for VRPTW or adapt existing methods from other VRP implementations.

3.1 Background and Context

In brief, a set of consumers with known demands constitutes the vehicle routing problem. There is often only one depot (multiple may also appear). Each vehicle has a specific amount of storage capacity and a set maximum distance it can travel. Every vehicle departs from a depot, delivers the requested products, and then drives back to the depot. Every customer is given a single vehicle route (multiple may also be considered if the requirements is satisfied). Any route's overall demand cannot be greater than the vehicle's carrying capacity.

The goal of the vehicle routing problem is to deliver goods to a group of consumers whose wants are known using vehicle routes that are as minimum-cost vehicle routes as possible that start and end at a depot.

3.2 Problem Statements

The issue may include optimizing package delivery for an e-commerce company where each product has a time window in which it must be delivered and delivery trucks have limited capacity. The goal is to reduce overall vehicle mileage traveled while ensuring that all packages get to their destination within their specific time frames and vehicle capacity is not exceeded.

Even with only a few hundred customer nodes, this problem is computationally tough to solve to optimality.

This problem has been identified as the Travelling Salesman Problem (TSP). For minor TSP problems, it may be feasible to intuitively determine the best path to take. As the size of a problem instance grows, one's intuition is likely to fail to identify the optimum course of action. When the quantity of deliveries to be made is large enough to necessitate the use of more than one truck, the problem becomes more complex.

It is necessary to take into account the issue of dividing the delivery burden between the vehicles. The next step needed is to partition the collection of deliveries so that each delivery vehicle can only deliver to a subset of them. The following inquiries must be addressed in relation to this enlarged VRP problem:

- How should the deliveries be distributed among the fleet's vehicles?
- What sequence should the deliveries made to the various vehicles be in?
- Which paths should each truck follow to get from one delivery location to the next?
- How will it address and resolve the issue if the products are not delivered to the customer on time?
- How will the vehicle respond in the event of a roadblock that prevents it from going on that route?

It is unlikely that one could choose the most effective delivery division, delivery schedule, and routes by relying solely on intuition. This contributes to the prevalence of inefficient vehicle fleets.

The outrageous cost of the software required to address issues like this is another concern. The ability to increase fleet efficiency may be made available to managers of more vehicle fleets by a less expensive software solution.

3.3 Research Questions

Possible research inquiries that might be investigated in relation to this issue include:

- How can we create effective metaheuristics or heuristics that can deal with large-scale VRP instances with time window constraints?
- How can we ensure on-time delivery by including real-time information, such as traffic jams or unforeseen delays, into the VRP optimization algorithm?
- Can we create unique algorithms that consider the dynamic nature of the issue and modify routes as needed to guarantee that time windows are met even in the face of unforeseen circumstances?
- How can we choose the best vehicles for each delivery task to reduce travel time and maximize the utilization of our whole fleet?
- How would we determine which exact methods, metaheuristics, or heuristics algorithms provide better solutions?

By addressing these questions, researchers may be able to cut costs, lower carbon footprints, and increase customer happiness by assisting logistics and transportation companies in improving their efficiency and effectiveness.

3.4 Relevance and Importance of the Research

The vehicle routing problem is a well-known NP-Hard integer programming problem, which means that the computational work needed to solve it grows exponentially with the size of the problem.

Since they may be discovered quickly and are accurate enough for the task, approximate solutions are frequently preferred for such issues. This activity is typically completed using a variety of heuristics, metaheuristics, and exact procedures, all of which depend on knowledge about the nature of the problem.

With a fleet of vehicles, a number of clients, and a depot, the issue can be summarised as follows: each vehicle leaves from the depot, travels to its initial client list, and then returns to the depot. Final requirements include serving every client exactly once, having a minimal goal function, and having a workable solution given the constraints. The purpose is to minimize an objective function, and it is categorized as an integer programming and combinatorial optimization issue. Depending on the precise problem definition, this function can change, but it often minimizes the sum of route lengths.

Since the issue falls within the category of NP-hard problems, specific solutions are not relevant for significant real-world issues with numerous clients. A portion of the research is still focused on creating new exact algorithms that can solve more complex problems, while another portion is exploring heuristic and metaheuristic algorithms that produce good results quickly.

We examine a few of these heuristic and metaheuristic algorithms and put them to use solving the Vehicle Routing Problem with Time Window (Capacitated) form of the problem and compare the results with each one.

3.5 Scenarios

The table 3.1 below provides an example of a dataset (instance) for a VRP with time window constraints:

Customer	X-coordinate	Y-coordinate	Service Time (minutes)	Time Window (start)	Time window (end)	Demand
Depot	0	0	0	-	-	0
A	4	5	15	7:00	8:00	3
B	2	0	10	7:00	8:30	3
C	1	3	5	7:30	8:30	1
D	5	5	20	7:30	9:00	4
E	3	4	15	8:00	9:00	2
F	2	3	15	8:30	9:30	1

Table 3.1: An example of a dataset for a VRP with time window constraints.

This data set depicts a VRP with a depot and six customers $(A–F)$. Each customer's and the depot's locations in a two-dimensional space are shown in the $X-coordinate$ and $Y-coordinate$ columns, respectively. The $demand$ column shows how much merchandise needs to be supplied to each customer, and the $service\ time$ column shows how long it takes to serve each customer.

The $time\ window$ $(start)$ and window (end) columns indicate the window of time during which the vehicle may arrive at each customer's location. In this illustration, the vehicle may show up to customers A and B between $7:00$ and $8:00$, respectively. There are no time window constraints on the depot or customer C.

The goal of the VRP is to determine the best delivery route for the vehicle that reduces the overall distance traveled while taking the time frame and capacity restrictions into consideration. This problem can be solved and near-optimal solutions found using metaheuristics like Simulated Annealing (SA), Genetic Algorithms (GA), and Tabu Search (TS), or classical heuristics like Clarke and Wright Savings Algorithms, Route First Cluster Second Algorithms, or exact methods like Branch and Bound Algorithms.

Practical Considerations

- Problem instance size: Customer and vehicle counts are the size of the problem instances.
- Problem type: Single depot, several depots, heterogeneous or homogeneous vehicle capacity — these are the different problem types.
- Restrictiveness: allowing or restricting time limits.

We will utilize the Solomon (1987) [212] datasets as a benchmark.A sample of Solomon RC101.txt [212] dataset is shown here in Table 3.2 in .txt format. Moreover, Figure 3.1 provides a visual depiction of the Solomon RC101.txt dataset. A sample of cost of the routes are shown in Table 3.3, and a graphical representation of the optimal routing option is displayed in Figure 3.2.

RC101
VEHICLE NUMBER 25
CAPACITY 200

CUSTOMER CUST NO.	XCOORD.	YCOORD.	DEMAND	READY TIME	DUE DATE	SERVICE TIME
0	40	50	0	0	240	0
1	25	85	20	145	175	10
2	22	75	30	50	80	10
3	22	85	10	109	139	10
⋮	⋮	⋮	⋮	⋮	⋮	⋮
99	26	35	15	77	107	10
100	31	67	3	180	210	10

Table 3.2: Solomon RC101 dataset in .txt format.

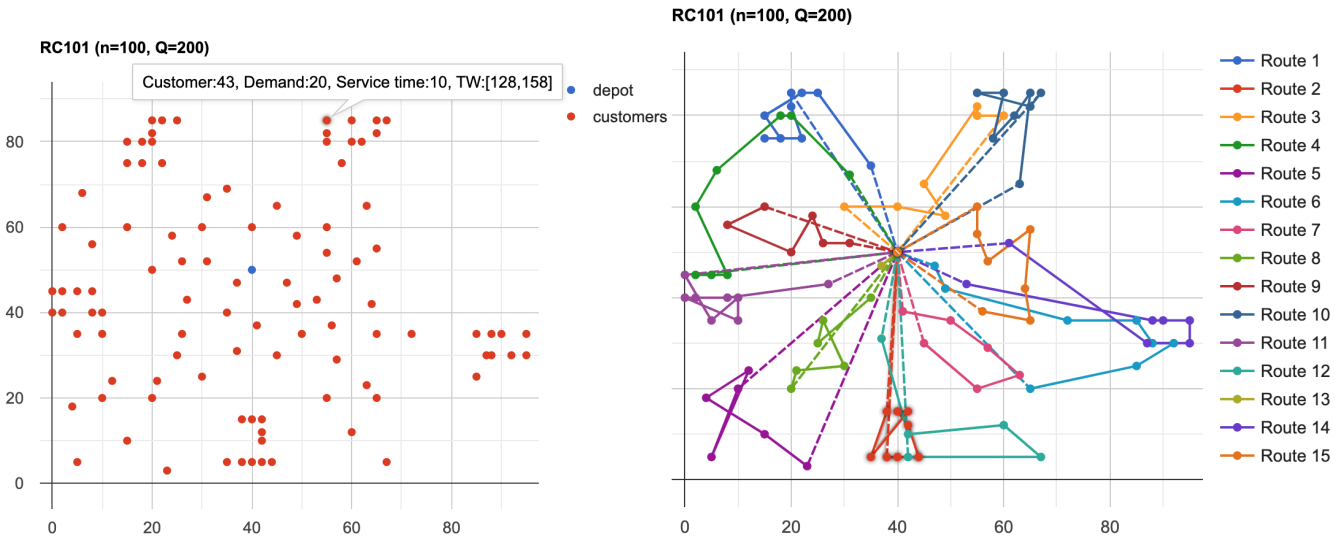


Figure 3.1: Solomon RC101 Instance dataset as a visual representation.

Figure 3.2: The optimal routing solution for the Solomon RC101 dataset in graphic form.

Route #1: 5 45 2 7 6 8 3 1 70
Route #2: 23 21 18 49 22 20 25 24
Route #3: 39 42 44 61 81 68 55
Route #4: 47 14 12 73 79 46 4 100
⋮
Route #14: 92 31 29 27 26 32 93
Route #15: 95 62 67 71 94 96 54
Cost: 1619,8

Table 3.3: The optimal routing solution and the calculated cost for the Solomon RC101 dataset.

3.6 Thesis Contribution

Using a library called PyVRP, an open/source solver called OR-Tools, and the raw implementation of two metaheuristic algorithms (a total of four separate models I have implemented here), I have used numerous classic heuristics and metaheuristic algorithms in my thesis. The following are the things I addressed in my thesis:

- I have modified Thibaut Vidal's HGS-CVRP code (`https://github.com/vidalt/HGS-CVRP`) to accommodate time windows and included construction heuristics, an enhanced crossover process, and a more intense local search by using the library PyVRP (`https://github.com/PyVRP/PyVRP`).
- Instead of using a C++ vector of vectors, we constructed the distance matrix as a flat array, which is more efficient and represented the routes into a graph.
- I have tried to eliminate pointless route-duration analyses.
- In order to improve the results of this model (`https://github.com/RJPenic/CVRPTW`), I have expanded the simulated annealing algorithm by incorporating a greedy neighbor search technique, calibrating the parameters, and plotting the routes into a graph.
- While implementing multiple ant colony systems on VRPTW (many processes simultaneously) on this model (`https://github.com/jonzhaocn/VRPTW-ACO-python`), which was difficult to execute, I was able to run and test multiple threads. I then adjusted the parameters to find better solutions and plotted the routes into a graph.
- I have utilized the parameter options of Google OR-Tools (`https://developers.google.com/optimization/routing/routing_options`by adjusting several algorithms on this source file (`https://github.com/AlexKressner/Vehicle_Routing_Problem`). In order to create the model, I have used a variety of algorithms. I have also used datasets to determine the routes and costs and have plotted them on a graph.
- Through the application of several algorithms and methodologies and their comparative analysis, I have attempted to attain the desired outcomes.
- I have combined all of the models I used from the source file into a single Python script and ran it as a whole. In addition, I have included a part in the script where we can simply edit the parameters without having to open each separate file.
- Each model's outputs, such as routes and costs, are plotted together and compared with the most well-known solutions as a gap percentage, algorithm run-time, etc. The results are all displayed in a table after running the code.
- Overall, by adjusting the parameters and choosing an initialization strategy other than a random seed, I have experimented with numerous approaches to decrease the CPU run time.

Chapter 4

Algorithms, Methods and Strategies

Any VRP is solved using one of three types of algorithms:

- Exact algorithms that seek the best answer. Among these methods are branch and bound, cutting planes, branch and cut, column generation, cut and solve, branch-and-cut-and-price, branch-and-price, and dynamic programming.
- Classical heuristics that seek for a good, feasible solution without ensuring optimality. These methods include savings algorithms, route first-cluster second, improvement heuristics, two-phase methods, and constructive heuristics.
- Metaheuristics which is the framework for building heuristics, are divided into groups as population-based (nature-inspired), hybrid, and local search-based metaheuristics.

We refer [124] [115] [68] [66] [166] for surveys of VRPs solution methods.

4.1 Exact Algorithms

Exact algorithms search throughout the entire state space in search of the best solution. The problem is NP-hard, hence accurate methods cannot solve large instances (Lenstra and Kan [98]). (Fukasawa, Longo, Lysgaard, de Aragao, Reis, Uchoa, and Werneck [191]) are able to solve literary examples with up to 135 customers. They require the use of specialist solvers and mixed-integer linear model construction since they are quite complex. The amount of CPU time needed to solve the problem grows exponentially as the problem size grows polynomially [86].

4.1.1 Branch and Bound (B&B)

Since its invention in the 1960s, B&B has been utilized as an exact method to solve the majority of VRPs. It is typically used to address discrete optimization problems that are defined as integer programs.
While the B&B algorithm is similar to the brute force technique in that it only explores a subset of solutions, the brute force algorithm is designed to investigate every solution. Make a tree out of the collection of answers. Every path from the root to the leaf is explored by the algorithm, but anytime it should branch, the estimated lower and upper bounds are verified. If the current solution does not fit, it is discarded, and the algorithm then analyzes the next path.
The following elements of B&B are crucial:

- The upper bound's (UB) and lower bound's (LB) quality is as follows: The value of the incumbent, or most practical, solution, is known as UB. The value of the function that represents the objective to the present node is LB, and if the current node is enlarged further, no successor node with a value less than LB can be reached. Cutting works better when a strong UB is located early in the search tree.
- The search strategy specifies how to branch to the next node. There are three fundamental tactics [144]:

1. $Breadth\ first\ search$: Extend the tree of searches by one level, next look at every node in that level before expanding by another level until the solution is found. This process is known as a breadth-first search. Since there are exponentially more nodes in the search tree at each level, it is impossible to utilize this technique to tackle huge issues.
2. $Depth\ first\ search$: Extend the search tree by selecting the most recently created node, keep going until you find a solution or a node without children, and then go back and examine the most recently generated node. Although it needs a polynomial amount of memory, the search tree and solution times are both very long.
   Since their lower bound values are higher than the values of the upper bounds, finding a good upper bound will be helpful in this method to reduce the size of the search tree by understanding many nodes.
   The disadvantage of this approach is that it may result in excessive CPU time and useless computations if the incumbent value is not near to the value of the ideal solution. Even when a heuristic is utilized to locate an initial answer, this search technique performs poorly in practice, according to the finding in [99].
3. $Hybrid\ search$: Consider a minimization issue. In order to reduce the total amount of nodes in the search tree that must be investigated before the ideal solution is discovered, the hybrid search method selects a node with the smallest lower bound to branch [86]. This approach focuses on finding evidence of optimality, which states that the current solution is the best option available [99]. It is referred to as $best\ first\ search\ strategy$.
   This approach is the fastest but it uses exponentially more memory. Because it branches to fewer sub problems, it is thought to be more effective than breadth first search [194]. However, it is less impacted by setting an initial upper bound than the depth first search.

The root node of the search tree serves as a placeholder for the original problem. It will be increased by include new nodes throughout the search. Every freshly formed node in the search tree is given a number, and all other nodes indicate sub problems. According to the search strategy, a node from a list of active (unexplored) nodes is selected to grow at each iteration. Every new node is examined. If it generates a workable solution, the upper bound's value is changed. If not, the node generates an impractical solution. Once more, it will be understood whether its value exceeds that of the current occupant. If not, it will be put on a list of active nodes that the search will expand upon.

Due to their greater limitations, the newly created nodes get squeezed more than the parent nodes. Each node in the search tree solves a relaxation of the original problem. In order to reduce the amount of created nodes in the search tree, fathoming is a key component of the B&B tree.

Any branch's search will come to an end if either a feasible solution is found or the goal function's value is worse than the UB value (best feasible solution thus far) [69]. The search in the entire B&B tree comes to an end when there are no longer any unexplored nodes, and the value of the current upper bound represents the ideal solution, if one exists.

We must choose, when B&B technique is utilized to solve a problem [47]:

- What restrictions can be loosened? (to make the problem easier to solve).
- What branching rules should be employed? (A rule that divides the possible set into smaller subgroups).
- What lower bounding technique should be applied? (Finding the value of the objective function for the relaxation problem at each sub problem is what this method is for).
- Which search technique should be applied? (It is standard practice to select the following sub problems to be addressed).
- What method of upper bounds will be applied?
- How to imagine (fathom)? when should you stop?

Heuristics are typically used to arrive at the initial (good) feasible solution. In order to fathom nodes and reduce the size of the search tree, the value picked up is used as the initial UB. The upper bound $UB = \infty$ if no known heuristic solution exists.

In general, relaxation refers to the removal of some or all restrictions. Relaxation can be divided into two categories: simple combinatorial relaxation and sophisticated (complex) relaxation, like Lagrangian relaxation [168]. By removing

some constraints and including them in the goal function, Lagrangian relaxation is defined. While it offers tight lower bounds, it also requires additional computing time. It is advised to pick challenging constraints (complicating constraints) to relax and add to the goal function in order to have a model that is simpler to solve [69].

B&B effectiveness is determined by the method used to decide which node in the search tree to branch to next [222]. It also takes into account the effectiveness of the upper bounds that are employed to reduce the size of the search tree [47]. therefore, a good upper bound is necessary to maintain the B&B tree's small size. You must use a heuristic or metaheuristic at some nodes in the search tree in order to obtain a good upper bound [86]. Therefore, the quick convergence of the lower and upper bounds is the foundation for B&B's effectiveness [99].

The B&B algorithm can be improved in two ways [79]:

- Tighten the bounds: Increase the lower bound or obtain a good upper bound using a heuristic.
- Improve the branching rules: Apply different branching rules to enhance the algorithm's branching rules.

Be aware that while a workable solution is frequently discovered early in B&B search tree, proving optimality requires more CPU time. B&B can also be utilized as a heuristic to generate workable solutions in a specific amount of time [51].

Survey of Branch and Bound

One of the most effective techniques for solving MIPs is the B&B algorithm. It is founded on the branching and choosing the node from the search tree principles.
There are numerous varieties of branching, including:

- Most impractical branching: It is inadequate because its outcomes resemble those of random branching.
- Pseudocost branching: Although weak at first, pseudocost branching is effective.
- Strong branching: It is efficient when measured in terms of the number of nodes in the search tree, but wasteful when measured in terms of time. $Full\ strong\ branching$ is employed when all available candidates are used.
- Reliability branching: A generalization of pseudocost and strong branching, reliability branching outperforms hybrid strong/pseudocost branching in terms of performance [222].

Strong branching is comparable to our branching principles, which are based on tolerances. Strong branching, as recommended in CPLEX 7.5, evaluates the candidates (fractional variables) to determine the additional cost of assigning an integer value to a fractional variable [222]. Before deciding which sub problem to focus on further, tolerance assesses the additional cost of eliminating the in-feasibility of the existing solution (destroying an infeasible tour by reducing one arc).

4.1.2 Cutting Planes

This method is employed when there are several linear IP constraints or when adding some strong valid inequalities makes the linear relaxation stronger [43]. It produces Gomory Cuts [69]. Through the deletion of a portion of the solution space, Gomory Cuts are utilized to tighten the relaxed problem. The additional cuts will have no impact on the initial issue, but they will have an impact on the relaxed issue by raising the likelihood that a solution will be found. The following should be done to apply the method: The best solution is the value of the incumbent, which should be stopped after adding cuts and solving the relaxation issue until the answer at the present relaxation problem equals the incumbent (current upper bound) [80].

4.1.3 Branch and Cut (B&C)

It combines the cutting plane method and the branch and bound algorithm. Cutting planes can be used to make global cuts or local cuts at individual nodes in the search tree. As a result, the size of the tree will be decreased [149] [69]. By eliminating a group of solutions for the relaxed subproblem, B&C adds a cutting plane to the tight relaxed subproblem. For the unrelaxed subproblem, the eliminated solutions are impractical [113].

B&C has an advantageous impact on increasing the size of solvable cases by shrinking the search tree, but it has a negative effect on lengthening the time spent at each node [113]. The lower bound picked up from the enumeration tree is superior to the bound obtained from the branch and bound tree in branch and cut, where enumeration benefits from cutting plane. On the other hand, cutting plane benefits from enumeration, and when employed with branching, the separation method can be more active [43].

4.1.4 Other Approaches

The literature also includes exact methods like:

- **Cut and Solve:** Instead of the search tree, Cut and Solve employs a path tree. A relaxed problem and a spare problem are two simple subproblems that are solved at each node, and a constraint is added to a relaxed problem. Because there is only one branch, this method has the advantage that the search will not select the incorrect branch, and the memory requirements are manageable [203].
- **Column Generation:** In order to avoid increasing the number of constraints, this method increases the number of variables in the problem [69]. According to [86], it is regarded as a dual-cutting plane. For instance, the original linear program is separated into a price subproblem and a linear limited master problem [31]. On column generation, we cite [87] [100] [120] for more information.
- **Branch-and-Cut-and-Price:** It is described as a fusion of the B&B, cutting plane, and column generation techniques that yield significant algorithms [190] [86].
- **Branch-and-Price:** It is described as a B&B and column generation hybrid. Each node in the search tree produces a new column [86].
- **Dynamic Programming:** Bellman proposed dynamic programming [17]. It is a process known as $multistage\ programming$ that is used to address optimization issues. The four components of dynamic programming are stages, states, decisions, and policies [51].

4.2 Classic Heuristics

For several reasons, not all problems can be handled by exact methods. Finding the best answer will be challenging, for instance, if a problem has a big number of restrictions or a vast search space since there will be a large number of plausible alternatives [144]. Since VRP is an NP-hard issue, as we have already discussed, it can occasionally be challenging to discover the best solution within a reasonable amount of time, even for small instances. As a result, rather than focusing on obtaining the ideal answer, we must accept the best viable solution that is discovered in a reasonable amount of processing time [73]. As a result, we must employ alternative techniques like heuristics or metaheuristics.

Heuristics were created and developed between 1960 and 1990 [67] and have been investigated by numerous academics [65] [49] [166]. It is described as a method for locating approximations, specifically a process that yields a good workable answer (not an ideal one) [63]. This yields a roughly correct answer but offers no assurance of optimality. Local search is one heuristic that can locate almost ideal answers in a reasonable amount of time. The process starts with an initial solution looking for a superior alternative in its immediate vicinity. Local search will keep presenting the current solution as locally optimal until a better alternative is found, at which point it will stop. Empirical findings highlight the fact that local search can find workable answers in reasonable CPU times [50].

The quality of the solution is determined by calculating the ratio between the value of the final solution obtained by the heuristic and the optimal solution or best-known feasible solution obtained from the literature [50]. The effectiveness of heuristics is measured by the running time (CPU) and the quality of the solution [50].

[97] state that the four criteria of accuracy, speed, simplicity, and flexibility can be used to compare heuristics. They list the most popular VRP heuristics, including: Because of its simplicity and speed, Clarke and Wright's algorithm [65] is one of the most well-known algorithms, Sweep algorithm (less simple, more speedy) [20], Fisher and Jaikumar's algorithm [134]. Because it is straightforward, quick, and easy, Greedy algorithm is popular, however, it falls short [144].

Each heuristic is created for a distinct problem, which is the drawback of heuristics [144]. The nearest neighbor heuristic, insertion heuristic, and tour improvement heuristic are a few of the traveling salesman problem heuristics that can be applied to solving VRP with simple modifications. There are numerous additional heuristics in addition to these [115].

In [73], they discuss heuristics that have two stages: the construction stage, which constructs an initial solution, and the local search stage, which searches the immediate area for the original answer. According to [75], the three categories of classical heuristics are as follows: improvement heuristics, two-phase methods, and constructive heuristics.

4.2.1 Constructive Heuristics

These heuristics concentrate on developing a workable solution with a primary focus on cost rather than improvement [75]. To create a workable solution, two basic methods are used. Insertion heuristics is the first method, which builds a solution by inserting one client at a time. This class includes algorithms that use construction to obtain workable solutions, such as greedy algorithms [144]. Savings heuristics, which employs the concept of merging the routes where each consumer is supplied, is the second strategy. Clark and Wright Savings Algorithm [65] is one illustration.

4.2.1.1 Savings Algorithm

Many methods use the Savings algorithm (Clarke and Wright [193]) published in 1964 to compute the basic solutions. A few parameterized variations have also existed; these are listed in Corominas, Garca-Villoria, and Pastor [192].

This algorithm was selected because it is deterministic and generates a reasonable initial answer. Additionally, computational complexity is quite low; $O(N^2)$ is required for saving, and $O(NlogN)$ is required for sorting (depending on the sorting algorithm used, $O(NlogN)$ is the computational complexity of an average quick sort run), giving us a total of $O(N^2)$. The lack of randomization, which is critical for exploring a significant percentage of the state space, especially in population-based approaches like the genetic algorithm, may be a drawback of this algorithm as an initialization method.

In order to improve the metaheuristic algorithms, we built a version with a lambda parameter ($d[i][j]$ on line 5 of Algorithm 1 would change to $lambda.d[i][j]$, where $lambda$ is a random real number in the range $(0, 2)$). It just aids in obtaining various basic solutions. (Most metaheuristics already include a random component; we don't need to find them a random basic solution.)

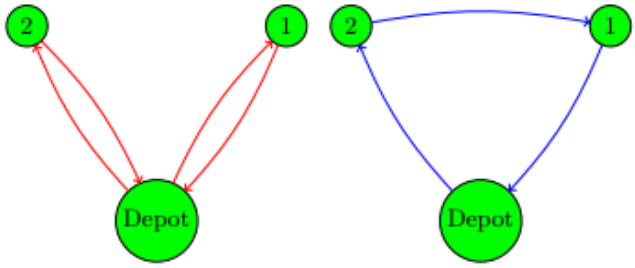


Figure 4.1: Savings concept (before & after savings merge) [136].

We begin by demonstrating the fundamental savings idea (Figure 4.1), which states that by merging two routes ($R1$ and $R2$) into one route ($R3$), we can save money. Specify $C(R1)$ as the price of $R1$ and $C(R2)$ as the price of $R2$. $R3$ is calculated to cost as follows: $c_{a,b}$ is the cost of traversing $e_{a,b}$, and $R1 + R2 - c_{i,0} - c_{0,j} + c_{i,j}$. The only difference between the existing solution and the solution after the procedure is the saving cost. On line 5, it is calculated. We sort the savings $V_{i,j}$, for $i, j \in \{1, \ldots, K\}$ in descending order and after calculating all possible combinations of savings $V_{i,j}$, so that we always start with one $V_{i,j}$ that has two customers i, j.

We decide whether or not we can connect them and how to connect them if they are not on the same route. Only if they are at the start or end of a route (($i_p = 0$ or $i_s = 0$) and ($j_p = 0$ or $j_s = 0$)) then we can connect them. There are just two possibilities. They are either both at the route's end ($i_p = 0$ and $j_p = 0$) or one is at the beginning while the other is at the end (($i_p = 0$ and $j_s = 0$) or ($j_p = 0$ and $i_s = 0$)). We can now only deal with one saving where $j_p = 0$ and $i_s =$

Algorithm 1 Savings algorithm pseudocode

```
1:  procedure AlgorithmSavings
2:      Initialize each customer in his own route
3:      for i = 1, ..., K do
4:          for j = 1, ..., K do
5:              s[it].cost ← d[i][0] + d[0][j] − d[i][j]
6:              s[it].i ← i
7:              s[it].j ← j
8:              it ← it + 1
9:              Sort s in descending order
10:             for all x in s do
11:                 a ← x.i
12:                 b ← x.j
13:                 if ConstraintsViolated(a, b) then
14:                     continue
15:                 if a.next = depot AND b.next = depot then
16:                     reverseRoute (b)
17:                     if a.next = depot AND b.prev = depot then
18:                         connectRoutes (a, b)
```

0 because, on lines 3–4, we allowed the savings $V_{i,j}$ and $V_{j,i}$ (Algorithm 1, line 17). If j_s and i_s are both equal to zero, the reverse route technique is first applied to the entire route, which includes node j. The routes are finally connected if $j_p = 0$ and $i_s = 0$ by deleting $e_{0,j}$ and $e_{i,0}$ and adding $e_{i,j}$.

The number of routes that are included in the final solution is reduced as the algorithm moves on to further pairs of nodes. The number of paths in the final solution is typically small or very near to the bare minimum.

4.2.2 Two Phase Methods

There are two models that exist in this two-phase method:

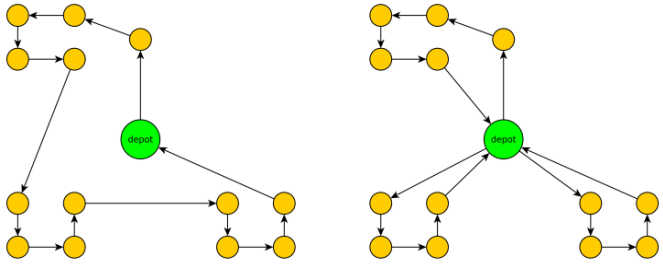


Figure 4.2: Route construction & clusterization [136].

- **Route First-Cluster Second:** We first build a massive traveling salesman problem (TSP) tour, and then we divide the tour into workable vehicle routes. The shortest path problem on an acyclic graph is solved to complete the cutting phase. Having a graph that is made up of the TSP path and adding workable edges, where edge $(i; j)$ represents the cost of having nodes along that route. (In brief, create a single path for all vertices, then divide it into workable routes) [15].
- **Cluster First-Route Second:** First, we cluster the customers such that they are close to one another and that their combined weights do not exceed the vehicle's carrying capability. By resolving a $generalized\ assignment\ problem$ (GAP), this is attained. Following that, each cluster represents a route, thus we use TSP to solve it. There

are several ways to create clusters, including the intriguing Sweep Algorithm, which rotates a ray centered at the depot using the smallest angle from prior customers until the clusters are too large for a vehicle. e.g Fisher and Jaikumar algorithm [134] (In brief, create a route for each subset of customers after grouping the customers into groups).

When compared to constructive heuristics, this sort of heuristics offers good quality workable solutions [75].

4.2.3 Improvement Heuristics (Local Search)

In local search, improvement heuristics employ exchange and relocation operators. They are referred to as neighborhoods.

- **Local Search:** Improving objective function locally and iteratively.
- **Neighbourhoods:** Neighbourhood is a solution that can be derived by applying a local operator (exchange, swap, etc.) to the present solution.

Any first viable option is used as the starting point for the search, and improvements are made in the neighborhood of the initial solution. A route's arc or vertex can be exchanged, or the routes themselves can be changed, as part of the improvement [75]. Later in this chapter, we have gone into further detail regarding it.

4.3 Metaheuristics

Metaheuristics are thought to be an effective strategy for solving challenging combinatorial optimization problems and produce excellent outcomes [145]. Heuristics that make use of certain metaheuristic rules make up this system.

Metaheuristics are intended to be used to efficiently and effectively explore the search space [34]. The structure of memory, complex neighborhood search rules, and solution recombination are all combined in metaheuristic methods. By combining these techniques, metaheuristics strive to increase the quality of results relative to classical heuristics while increasing computation time [67]. Additionally, metaheuristics employ a variety of strategies to leave local optima, including iterative local search, multi-start local search, GRASP, neighborhood switching (VNS), and moving to non-improving neighbors (TS, SA).

In contrast to heuristics, metaheuristics are independent of the problem. They are more adaptable and can be utilized as black-boxes because they do not capitalize on the particular of the situation. They can leave the local optimum and discover a better solution because they are not as greedy (they do not always take the best answer; they may even take a worse solution than the found optimum).

According to local search, the size of the search space and the effectiveness of the search are related [144]:

- The local optimum will be found and the search will end fast if the search space is kept narrow. The quality of the ideal answer in this situation is poor. To raise the quality of the solution, it is advised to expand the search area.
- If the search space is enormous, we would run out of time, memory, or for any other reason before finishing the search.

Due to these factors, the search space shouldn't be chosen at random [144]. The following principles can be used to compare metaheuristics [177]:

1. Simplicity: Simple implementation, explanation, or analysis.
2. Coherence: The heuristics' stages for a particular situation should make sense and logically follow the metaheuristic's rule.
3. Effectiveness: In finding ideal or almost ideal conditions.

4. Efficiency: In finding a suitable solution in a fair amount of CPU time.
5. Robustness: Robustness across the board, not only in certain circumstances.
6. User-friendliness: It is preferable for the metaheuristics to be simple to comprehend and simple to use, hence having a limited number of parameters (few) is necessary.
7. Generality: They must therefore be appropriate for a variety of issues.
8. Interactivity: By letting the user adjust the procedure, interaction.
9. Multiplicity: The capacity to offer a few close-to-optimal options.

There are other classification methods, such as memory utilization (TS, GA) versus no-memory (SA), nature-inspired (ACO) vs. non-nature-inspired (TS) metaheuristics, deterministic vs. stochastic, constructive vs. iterative, and so forth [34] [107].

In this thesis, metaheuristics are categorized as follows according to how many solutions they are using at any one time:

1. Local Search Based Metaheuristics: In this group of metaheuristics, one solution is used to obtain another. For example, simulated annealing (SA), tabu search (TS), hill climbing (HC), late acceptance hill climbing (LAHC), step counting hill climbing (SCHC), deterministic annealing (DA), neural networks (NN), variable neighborhood search (VNS), guided local search (GLS), iterated local search (ILS) and greedy randomized adaptive search procedure (GRASP).
2. Population Based Metaheuristics (nature inspired): This employs multiple solutions to discover a brand-new one. For example, genetic algorithm (GA), evolutionary algorithm (EA), ant colony optimization (ACO), scatter search (SS), path relinking (PR) and particle swarm optimization (PSO).
3. Hybrid Metaheuristics: Hybrid metaheuristics, also referred to as artificial intelligence (AI) and operation research (OR) metaheuristics, combine metaheuristics and optimization techniques to create effective outcomes [34].

Maximum CPU usage, a maximum number of iterations, or a maximum number of iterations without improvement can all be halting conditions for the majority of them [34]. To fully comprehend how metaheuristics may function, $exploitation$ and $exploration$, as well as their distinctions, must be defined first.

Exploitation and Exploration

$Exploitation$ involves searching a certain area of the search space. We want to enhance a promising solution S, sometimes known as the $candidate$ solution in metaheuristics because it competes with the best solution. Our main search area in $exploitation$ is close to $candidate$ solution S.

$Exploration$ is the antithesis of $exploitation$. $Exploration$ involves looking at a bigger area of the search space. To try to improve it, we expect to uncover another potential alternative, S.

We seek the ideal balance between $exploitation$ and $exploration$ while using metaheuristic methods to solve optimization problems. Our algorithm would probably become trapped in the local optimum if we had massive $exploitation$ and minimal $exploration$. However, if we had massive $exploration$ and little $exploitation$, our algorithm would likely never find either the local or global optimum because it would still be searching the entire state space.

4.3.1 Local Search Based Metaheuristics

4.3.1.1 Multi Start Local Search

The multi-start method is intended to diversify the search and avoid local optima [135]. The process consists of two steps: the first stage produces a solution, and the second stage enhances it. The best local optimum is the outcome of the Multi-start method [187].

One of these criteria can be used to categorize multi-start methods: degree of rebuild (built from scratch or fix some components in the previously created solutions), memory (use memory or memoryless), or randomization (systematic or randomized) [187].

The following are the main steps of the multi-start algorithm [135]:

- first iteration $i = 1$;
- while the stopping condition is not met, proceed
  - construct solution s_i;
  - improve the solution s_i to get $\bar{s_i}$;
  - update the best solution found so far;
  - $i = i + 1$;

4.3.1.2 Simulated Annealing (SA)

The physical annealing process used on metals is what inspired simulated annealing. The basic idea behind the annealing process is to heat metal to a high temperature and then progressively cool it in accordance with a timetable. When a metal is heated to a high temperature, its atoms lose their crystalline lattice and are randomly distributed throughout the metal. As the metal cools, the atoms rearrange themselves into a pattern that resembles the global energy minimum of a perfect crystal.

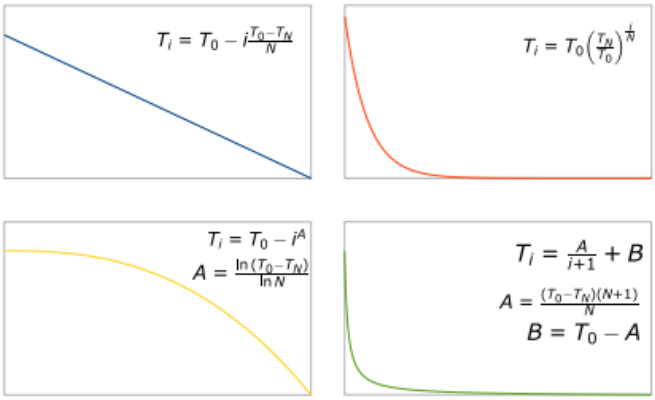


Figure 4.3: Cooling schedules example [233]. T_i is the temperature for cycle i, where i increases from 1 to N.In which the initial temperature T_0 and final temperature T_N, are determined by the user, as N.

The simulated annealing method operates on a similar concept. It was created and first presented by S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi in 1983 [201]. The method begins with a high temperature (large $exploration$) to identify the most promising solution over the whole state space, then as time passes, the temperature gradually decreases (large $exploitation$) to locally enhance the solution that was identified. A cooling schedule or strategy is the method used to reduce temperature. Because there is no one cooling schedule that works effectively for all issues, choosing one is a challenging aspect of the algorithm.

The drawback of simulated annealing (and cooling schedules in general) is that the beginning parameters for correct temperature calculating must be set. These initial parameters include the initial temperature, final temperature, and an estimate of the total number of iterations, which are typically unknown and it takes long time to calculate the optimal route solutions. Figure 4.3 shows an example of a few cooling schedules.

Accepting moves that aren't always better than the present value of the objective function, offers a strategy to avoid the local optimum and find the global optimum [7]. Four key elements are required for SA: a brief problem description, a neighborhood function, a transition technique, and a cooling schedule (static and dynamic schedules) [48].

Let T be a parameter used to decide acceptance of the $candidate$ solution in the event that its value is not better than the value of the existing solution, and let z and z_i be the objective functions of the current $candidate$ solution and the current solution, respectively [63].

Starting at a plausible initial solution chosen at random, SA proceeds to the next solution by applying $move\ selection\ rules$. The move choice guideline is:

- if the following solution's value is superior to the present one, accept moving on;
- alternatively, if no improved solution is discovered close to the existing one:
  - only move to the closest neighbor if the random number is lower than the acceptance probability.

$$Prob\{acceptance\} = e^{-x} \qquad where \qquad x = \frac{z_i - z}{T} \tag{4.3.1}$$

  - If not, stick with the existing resolution (don't change).

The value of T lowers as the number of iterations increases for each value of T utilized during the search. The process of stopping to obtain the best solution so far, at any iteration, as the final solution occurs when the number of iterations is assigned with the minimum value of T (or if the move selection rule can't accept any near neighbor) [63]. Three things set SA apart from local search [144]:

1. The way SA stops: When the stopping conditions are met, SA comes to a complete stop.
2. The way SA moves: The method SA moves is that it accepts solutions based on the parameter T in addition to moving to better solutions.
3. Throughout the search, T's value is often updated: This has an impact on SA's output.

SA employs the probabilistic rule: Which states to accept a solution if it is superior to the one currently being used in the neighborhood. In that case, there are two choices: either accept this solution or start looking around the same neighborhood again for another solution [144]. Deterministic annealing (DA) is the result of applying deterministic ideas to SA [148].

The maximum number of iterations generally or the maximum number of iterations if the present value of the goal function does not improve might serve as stopping conditions [147]. The final SA issue is that because SA relies on probabilistic rules, executing the same instance again under the same circumstances yields different results [144].

By Alfa et al. in 1991, SA submitted a CVRP application [12]. SA's performance was poor. The reason for this is that an initial solution was found using a route first-cluster second technique.
Osman's SA algorithm from 1993 [165], it performed well, but this method was not regarded as being robust (not a strong algorithm). Additionally, SA is a well-liked strategy because it is straightforward and quick to utilize [7].

4.3.1.3 Tabu Search (TS)

By using short-term memory, TS can be considered as an expansion of a classical local search [146]. One of the most well-liked and effective metaheuristics for VRP, the TS approach was first proposed by Glover in 1986 [78] [104]. Starting with an initial solution and accepting progression to non-improving moves if local search (LS) obtains a local optima is the primary concept of TS [70].

The search space, the neighborhood structure, the short-term tabu lists, and aspiration criteria are the fundamental elements of any TS. If there is no likelihood of cycling, tabus might be disregarded according to the aspiration criterion. In other words, if it yields solutions that are superior to the existing solution, it accepts movement to tabu moves [146]. After locating a local optimum via a $local\ search\ procedure$, TS goes to any location within the neighborhood. Apply the local search procedure once more to identify a new local optimum if a better solution is discovered. Apply another move in that case [63].

When an algorithm is running, a tabu list is updated. Tabu lists will record each move in $tabu\ moves$ in order to prevent repetition of the same local optimum [61]. The tabu list should be effectively utilized for a tabu search to make the most use of memory. The tabu list should be managed according to three principles [2]:

- Size of the tabu list (short term forbidding strategy): A short tabu list will be meaningless because cycling may have happened, whereas a long tabu list will broaden the search and increase the number of solutions explored. The most helpful method is to vary the size of the tabu list because, unfortunately, determining the size of a tabu list is difficult [2].
- Intensification (Intermediate term forbidding strategy): The term "intensification" is defined as "searching in the promising neighborhood," which refers to the area of the neighborhood that has particularly good solutions [63]. The tabu list's size needs to be reduced for a few iterations in order to sharpen the search [2].
- Diversification (long term forbidding strategy): Denotes looking in new areas (neighborhoods that have not yet been investigated) [63]. Repeatedly randomly perform restart in order to broaden the search [2].

Different stopping conditions can be employed, for example a maximum number of iterations in generally, a maximum number of iterations if the value of the goal function does not improve, or a maximum CPU time [63].

Algorithm 2 Tabu Search algorithm pseudocode

```
1: procedure AlgorithmTabuSearch
2:     tabuList ← initTabuList()
3:     create an initial solution bestSolution (usually random)
4:     while Stopping criteria not met do
5:         temporarySolution ← getFeasibleNeighbour (bestSolution)
6:         if getPrice(temporarySolution) ≤ getPrice(bestSolution) AND notIn(tabuList, temporarySolution) then
7:             bestSolution ← temporarySolution
8:             addToTabuList(tabuList, temporarySolution)
9:             deleteExpired(tabuList)
10:            return bestSolution
```

By using the most recent search history as a parameter to guide the search, TS attempts to improve the best solution discovered thus far. This is how it's done: Accept the best neighboring solution and decide whether its value is superior to the present neighboring solution, but it should not be in the tabu list [144].

Osman's tabu search algorithm [165] is an illustration of a TS algorithm, where the initial version of the algorithm, known as $best - admissible$, is employed. It looks around the entire neighborhood and decides on the best move. The second version, known as $first - best\ admissible$, chooses the first move that is acceptable. Both versions produce good results according to computational results [147].

4.3.1.4 Hill Climbing (HC)

The simplest metaheuristic technique for local search is hill climbing. It starts with the initial solution S_0, which is typically created at random. The approach explores a neighborhood of the solution and chooses the best-discovered solution as the following point to explore. The simplicity of this approach is a benefit, although hill climbing frequently becomes stuck in a local optimum that may be quite remote from the global optimum. There are a number of ways to deal with getting trapped in the local optimum, such as random restarting the procedure if it becomes stopped or using more complex approaches for hill climbing [136].

Algorithm 3 Hill Climbing algorithm pseudocode

```
1: procedure AlgorithmHillClimbing
2:     create an initial solution bestSolution (usually random)
3:     while Termination condition not met do
4:         temporarySolution ← getFeasibleNeighbour (bestSolution)
5:         if getPrice(temporarySolution) ≤ getPrice(bestSolution) then
6:             bestSolution ← temporarySolution
7:             return bestSolution
```

4.3.1.5 Late Acceptance Hill Climbing (LAHC)

The hill climbing technique has recently been improved using late acceptance hill climbing (LAHC). Yuri Bykov created it and presented it in 2008 [52]. Since N is an input parameter for the method and can be based on issue requirements, LAHC is an iterative search algorithm that can accept a worse $candidate$ solution if it is equal to or better than the $candidate$ solution discovered N iterations earlier. The lack of a cooling schedule is a key benefit of the LAHC strategy. As a result, it is far more durable than $cooling - schedule - based$ methods (see simulated annealing (SA) 4.3.1.2 for more information). Additionally, LAHC is a practical and efficient search technique.

Algorithm 4 Late Acceptance Hill Climbing algorithm pseudocode

```
1: procedure AlgorithmLateAcceptanceHillClimbing
2:     create a candidate solution bestSolution (usually random)
3:     lastCosts[ ] ← getPrice(bestSolution)
4:     i ← 0
5:     while Termination condition not met do
6:         temporarySolution ← getFeasibleNeighbour (bestSolution)
7:         if getPrice(temporarySolution) ≤ lastCosts[i mod pL] OR getPrice(temporarySolution) ≤
   getPrice(bestSolution) then
8:             bestSolution ← temporarySolution
9:             lastCosts[i mod pL] ← getPrice(temporarySolution)
10:            i ← i + 1
11:            return bestSolution
```

4.3.1.6 Step Counting Hill Climbing (SCHC)

A step counter Bykov and Petrovic created and presented step counting hill climbing (SCHC), a novel local search heuristic, in 2013 [243]. It is comparable to LAHC, but it significantly enhances the hill climbing technique by allowing escape from a local optimum. The cost of the current solution establishes an upper acceptance bound for the following pL steps, where pL is a single input parameter that can either be taken from the issue instance or specified by the user. There are several ways to count the steps as well, which opens up a wide range of variations for this approach. The advantages of this approach are that SCHC is possibly even more effective and simpler than LAHC.

Algorithm 5 Step Counting Hill Climbing algorithm pseudocode

```
1: procedure AlgorithmStepCountingHillClimbing
2:     create a candidate solution bestSolution (usually random)
3:     lastCost ← getPrice(bestSolution)
4:     i ← 0
5:     while Termination condition not met do
6:         temporarySolution ← getFeasibleNeighbour (bestSolution)
7:         if getPrice(temporarySolution) ≤ getPrice(bestSolution) OR getPrice(temporarySolution) ≤ lastCost
   then
8:             bestSolution ← temporarySolution
9:             i ← i + 1
10:            if i ≥ pL then
11:                lastCost ← getPrice(bestSolution)
12:                i ← 0
13:                return bestSolution
```

4.3.1.7 Greedy Randomised Adaptive Search Procedure (GRASP)

A randomized greedy construction is used as a heuristic by GRASP, a multi-start procedure, to discover a solution [139]. It combines local search with constructive heuristics [34]. To put it another way, GRASP is a repeated technique with two parts in each iteration: the creation phase, which is an iterative greedy and adaptive process, and the improvement phase, which is a local search procedure [94].

While the second phase refines the first phase's solution, the first phase builds the initial solution. Until the halting criteria are met, both stages are repeated. The maximum number of iterations may be used as a halting condition [94]. Keep in mind that while more iterations result in longer CPU times, they also yield better results [140].

The following is a summary of the GRASP algorithm's primary steps [140]:

- Initialize: $max - iterations$, $seed$;
- for $k = 1$ to $max - iterations$ do - build a solution ;
  - if (solution infeasible) then (repair to get feasible solution);
  - local search (to find the local optimum);
  - update solution (if necessary);
- end;
- return $best - solution$.

The assessment of every other potential ingredient is the basis for choosing the subsequent element to be incorporated into the solution at each iteration of the construction phase. The best items are included in a $list\ of\ restricted\ candidates$ (RCL) list of restricted candidates (RCL) that is produced by the evaluation. One component is chosen at random from RCL and added to the unfinished solution. Continue updating RCL after reevaluating the remaining components until $RCL = \emptyset$ [140]. Use the repair process to make the solution possible if it turns out to be unfeasible [140].

Numerous factors, such as the neighborhood structure, neighborhood search strategy, and starting solution, might influence the local search procedure's speed and efficacy [140]. The final aspect states that starting with a high-quality solution that is acquired during the constructive phase is beneficial for local search [140].

There are two different neighborhood search strategies that can be used [140]:

- The $best\ improving\ strategy$: The optimal approach for improvement is to keep searching until all neighbors have been looked into and the best neighbor has been found. It might take a lot of time.
- The $first\ improving\ strategy$: The neighborhood is searched until the first neighbor with a value better than the current solution is located. The first neighbor discovered is returned.

Generally speaking, the best improvement can be more time-consuming but more successful than the first improvement. According to empirical research, the opposite is true in some circumstances (where the initial solution is not selected at random [182]). In general, GRASP is a fairly quick and straightforward metaheuristic [34].

4.3.1.8 Neural Networks (NN)

A net with weighted connections connecting its nodes is called a neural network (NN). Depending on the kind of connection, certain nodes' output can be used as input by other nodes in the network [29]. In order to do intricate computations rapidly, NN attempts to imitate the functioning of the human brain [144]. Neurons $v_i \in [0, 1]$, where $i = 1, 2, \ldots, N$ is an index to each neuron, are the fundamental components of NN. θ is the threshold, and w_{ij} is the weight, which can be either positive or negative, from neuron v_i to neuron v_j.

$$v_i = g(\sum_{j=1}^{N} w_{ij} v_i - \theta_i) \tag{4.3.2}$$

Where temperature is represented by the function $g(x) = \frac{1}{2}(1 + tanh(\frac{x}{c}))$ and $c > 0$ [29].

There are two different kinds of structures: $feedforward$, which transmits signals from input to output neurons, and $feedback$, which transmits signals both ways [29]. Additionally, there are two categories of neural networks (NN): hybrid and pure neural approaches [29].

NN offers numerous benefits, such as the ability to quickly identify a local optimal using training techniques and, in certain cases, the ability to use evolutionary techniques to go from a local optimal—if there are multiple local optimal solutions—to the global optimal [144]. However, when the network gets big, one drawback of NN is that it becomes less flexible [144].

4.3.1.9 Variable Neighbourhood Search (VNS)

Mladenović made this suggestion in 1997 [153]. It transitions from local to global optima using several neighborhoods [181]. Assume that our neighborhoods are nested. In order to get a random solution, we first search in the first neighborhood in quest of a local optima. We shift to this solution by treating it as a new local optimum if it outperforms the existing ones. If not, we proceed to the next neighborhood and so forth. When we have gone through all of the neighborhoods or discover a solution that outperforms the local optima, we go back to the first neighborhood.

Below are the primary steps in basic VNS [179] [12]:

- Initialization: Select the neighborhood N_k's structure where $k = \{1, \ldots, k_{max}\}$, identify x as the initial solution and decide on the termination condition;
- While termination condition is not satisfied, Do:

  for $k = 1$ to k_{max} do

  - Shaking: Find random solution x' from the k^{th} neighborhood of x;
  - Local search: Find the local optimum solution x'' with x' as initial solution;
  - Move to x'' if it is better than x and continue the search with $k = 1$, otherwise $k = k + 1$.

The maximum number of iterations overall, the maximum number of iterations without improving the existing solution, or the CPU time can all be used as the termination condition [179]. To avoid the local optimum, the solution x' is selected at random in the basic VNS steps [179].

The neighborhood can be searched using the following three approaches [177]:

- (i) Deterministic: We obtain variable neighborhood descent (VND) when the neighborhood's change is deterministic, or when finding the best neighbor is the main goal.
- (ii) Stochastic: We obtain reduced VNS (RVNS) when the random point is selected from the neighborhood, which is helpful for very large examples [179].
- (iii) Combination of deterministic and stochastic: Basic VNS (BVNS) is produced when a combination is used.

Skewed VNS (SVNS) and variable neighborhood decomposition search (VNDS) are two VNS extensions [179] [177]. Based on straightforward principles, VNS is regarded as strong, user-friendly, effective, and extremely efficient [179]. A general overview of such neighborhood search techniques is provided in Section 4.4.

4.3.1.10 Guided Local Search (GLS)

On its own, local search (LS) is a very fast way to identify workable solutions, but it may become trapped at local optima. Because of this, GLS is suggested to assist LS in escaping local optima and utilizing $penalties$ to arrive at the global optimal solution [37].

The following are the GLS algorithm's primary steps [37] [12]:

- Initialization:
  - apply construction method to find an initial solution;
  - set all penalties to 0;
  - define the augmented objective function;
- Repeat until stopping conditions are satisfied, Do:
  - apply improvement methods: such as simple local search, variable neighborhood search;
  - if (local optima is found) then (modify augmented objective function by increasing the penalties of one or more of the elements that appear in the local optima);
- Return best solution found.

GLS is a strategy that operates on a penalty base and is layered upon local search. The way it penalizes certain properties of a solution along the search path—we presume $minimization$—is what gives it its uniqueness and efficiency.

Augmented Objective Function

The following indicator function is denoted by I_i in the augmented objective function:

$$l_i(x) = \begin{cases} 1 & \text{if the solution, where x has feature i;} \\ 0 & \text{Otherwise.} \end{cases} \tag{4.3.3}$$

Some characteristics of a local optimum are penalized by the GLS metaheuristic. Let f represent the initial objective function and p_i be a penalty associated with a feature i. The following augmented objective function g is employed by the GLS metaheuristic:

$$g(x) = f(x) + \lambda \sum_i (l_i(x).p_i) \tag{4.3.4}$$

The intention is to use this new augmented objective function to allow the local search solve problems. The $penalty\ factor$, λ, can be used to fine-tune the search to locate entirely distinct solutions (a high λ value, $diversification$) or comparable solutions (a low λ value, $intensification$).

Penalties and their Modifications

Typically, penalties begin at 0 and increase by 1 for each local optimum. Because a feature is only penalized if its $utility$ is high enough, the GLS is novel and efficient. The goal is to penalize expensive features, but not too harshly if they appear frequently. The following is the definition of the $utility$ function for a feature i in a solution x:

$$u_i(x) = l_i(x) \frac{c_i(x)}{1 + p_i} \tag{4.3.5}$$

Where the cost related to feature i in solution x is indicated by $c_i()$. A feature's utility for a solution x is 0 $(I_i(x) = 0)$ if it is absent from the solution. If this feature i is frequently punished, the utility tends to vanish and is instead proportionate to the cost $c_i(x)$ of this feature in the solution x. A consistent feature in local optima may be a component of a workable solution.

Implementation

The implementation is both general for all routing problems and customized for routing problems. The fact that a solution traverses or does not cross an arc (i, j) is its selected feature. We will thus discuss a (i, j) arc feature (as well as c_{ij}, u_{ij}, and p_{ij}).

The cost of traversing an arc (i, j) in a particular solution will be indicated by the symbol d_{ij}. In our scenario, we have $c_{ij} = d_{ij}$, which is the cost of the desired function for that arc. If we employ different kinds of vehicles, this cost may vary accordingly. The augmented objective function we have is as follows:

$$g(x) = \sum_{(i,j)} d_{ij}(x) + \lambda \sum_{(i,j)} (I_{ij}(x)\ .\ p_{ij}\ .\ c_{ij}(x)) \tag{4.3.6}$$

The $gflags$ command line flag `routing_guided_local_search_lambda_coefficient` within the $Routing\ Library$ determines the penalty factor λ, which is set at 0,1 by default. For additional information to comprehend the GLS in its entirety, check [84].

In our thesis model, a method called local search [126] uses a single search path—looking around to enhance the initial solution and come up with a better one. The structure of the solution point is the same as that of the GA chromosomal representation, as explained in Section 4.3.2.1. The two steps that make up the local search algorithm's process are as follows:

1. Let s be the initial solution.
2. Find $S = searchNeighborhood(s, k)$, the set of neighboring solutions of s, where k denotes the size of S. The function $searchNeighborhood(s, k)$ returns the solutions of the k neighbors.
3. Repair each instance of a solution s' in S that differs from the specified parameters.
4. Return $s^* = \text{argmax}(s'.fitness)$, where $s' \in S$.

The amount of moves or CPU time can be used as stopping conditions. Penalties for certain features from the local optimal solution will be raised once the local optima has been identified. Furthermore, this leads to search diversity in addition to the escape from the local optima. Accordingly, features with high costs incur greater penalties than ones with low costs [37]. GLS is an easy-to-use, effective metaheuristic. It also looks for the locations that show promise [37].

4.3.2 Population Based Metaheuristics (Nature Inspired)

4.3.2.1 Genetic Algorithms (GA)

Genetic algorithm (GA) is used to solve issues for which there is no practical exact algorithm. J. H. Holland first presented the genetic algorithm in 1975 [89], created to address problems where the answer could be expressed as a binary vector and as its name implies, it draws inspiration from genetic development. The $population$ is the collection of solutions that the algorithm maintains. Since the $population$'s solution aims to be the best option, it is referred to as a $candidate$. We require a function that yields solution-quality information before we can apply a genetic algorithm. The function is known as a $fitness$ function in GA language, and it can be the $total\ travel\ distance$ in the case of VRPs.

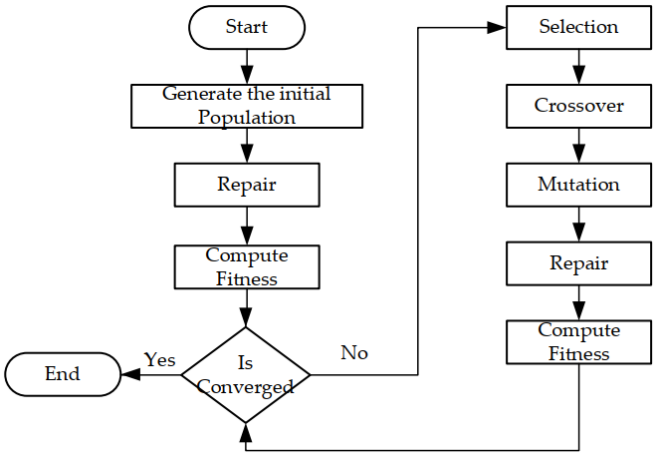


Figure 4.4: The flow of the GA scheme [189].

We randomly construct the route for every initial population solution in order to initialize the first population. After obtaining the list of allocated consumers for each solution using an initial route, we arrange the customers in ascending

order based on their demand time [63]. Once the route has been modified, the trip is constructed with the assumption that shippers will only need to return to the depot when their work is completed or the trip's capacity is exceeded. After that, they have to go back to the depot [147].

$Initialization$, $crossover$, $selection$, and $mutation$ are the four stages of general GA [189]. However, there exist GA variants in which either selection or mutation is absent [32]. There are two strategies: the first employs mutation after crossover, and the second employs either crossover or mutation (but not both) [189]. The following are the genetic algorithm's primary steps [42] and in Figure 4.4 has shown the flow of GA scheme:

1. Initialize a population;
2. Apply an evaluation function to every individual;
3. Select fitness individuals to create a new generation through mutation or crossover;
4. Apply the evaluation function again to retain the good elements and remove the bad ones;
5. check to see if stopping conditions are met before stopping. If not, proceed to Step 3.

The following shows the primary GA operators [142] [224]:

- **Initialization:** The process of initialization involves starting a population created from candidates. Either a random or heuristic generator can be used to create the candidates. According to the proposed optimization model, the

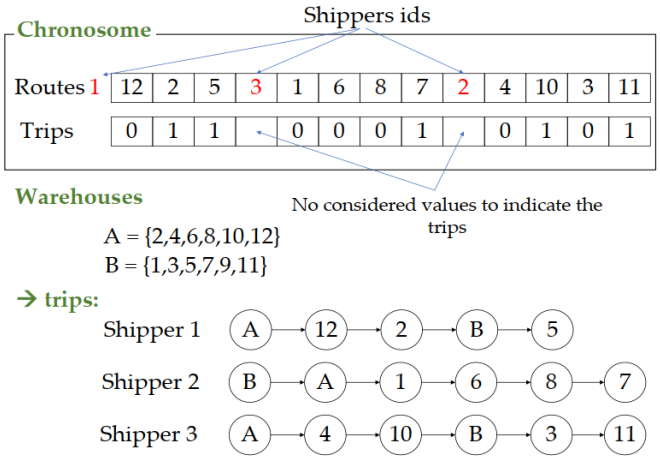


Figure 4.5: Chromosome representation [224].

individual's structure is equal to the decision variable O. The set of π individuals represents the population P that we generate. The chromosome is represented for programming simplicity by two arrays, $routes$ and $trips$, each with a size of $(C + K - 1)$. By recording the identities of C customers and $(K - 1)$ shippers and placing them in a random sequence, $routes$ reflect the pathways taken by the shippers. The $customer\ IDs$ are represented by positive integers, and the $shipper\ IDs$ are represented by negative integers. The shippers' journeys are identified using $trips$.

In the case of 12 customers with IDs ranging from 1 to 12, three shippers with IDs ranging from 1 to 3, and two warehouses A and B, the chromosomal representation is displayed in Figure 4.5. The first three $routes$ items in the figure are 12, 2, and 5. This indicates that these clients are to be delivered to by the shipper with ID 1. There are only binary items in $trips$ that correspond to these. When the value is 0, the shipper can go with their journey without any interruptions, but when it is 1, they must go back to the associated warehouse in order to load the package before proceeding to the next client. Only $K - 1$ values are saved in $trips$; so, shipper ID 1 does not need to be stored in $routes$, boosting the random process's ease.

- **Fitness function:** For the individual, we employed the compromise $Euclidean\ distance-based$ function as follows:

$$p.fitness = \left(\sum_{i=1}^{4} w_i \left| \frac{F_i - z_i^*}{z_i^{worst} - z_i^*} \right|^2 \right)^{\frac{1}{2}} \quad \forall\, p \in P \tag{4.3.7}$$

For situations involving single-objective optimization, every suggested algorithm produces the best outcomes. In this process, z_i^* and z_i^{worst} can be regarded as pre-computed.

- **Selection:** The procedure of selecting candidates for the upcoming surgeries involves choosing them from the current population. Typically, fitness function is used to choose candidates in order to maintain a robust population. Tournament selection is an example of selection; it chooses x candidates at random and then selects the best $candidate$ from the group of selected candidates. In order to retain φ elite individuals in the next generation, our approach excluded them from the $crossover$ and $mutation$ phases.

- **Crossover:** The process of combining candidates to create a new candidate is called $crossover$. The population often retains well-solved portions of a solution because of this process. One type of crossover is called one-point crossover; it starts with two binary vector candidates, $a, b \in \{0, 1\}$, and splits them into four binary vectors by randomly generated point i, $(a_1, a_2, \ldots, a_{i-1})$, $(a_i, a_{i+1}, \ldots, a_n)$, $(b_1, b_2, \ldots, b_{i-1})$, and $(b_i, b_{i+1}, \ldots, b_n)$. It then combines the split parts to create solutions, such as $(a_1, a_2, \ldots, a_{i-1}, b_i, b_{i+1}, \ldots, b_n)$, $(b_1, b_2, \ldots, b_{i-1}, a_i, a_{i+1}, \ldots, a_n)$.

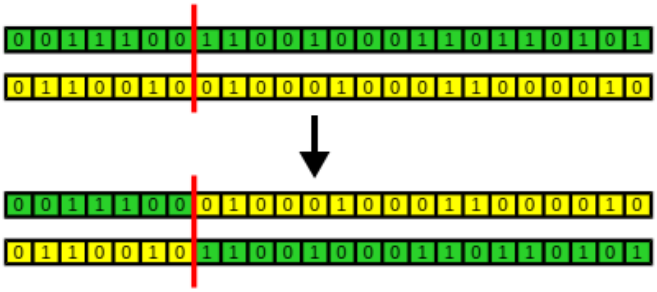


Figure 4.6: Crossover operation example [136].

Crossover produces a new solution while keeping its parent's positive aspects. Our choice for a crossover rate was μ. For the remaining individuals of the next generation, there are five steps involved in implementing the crossover (see Figure 4.7). These steps are as follows:

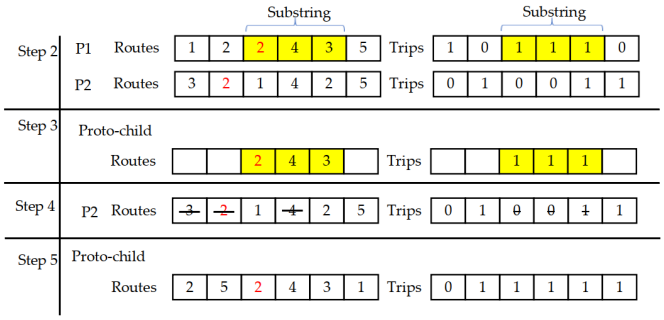


Figure 4.7: Step 2 to Step 5 of the Crossover Phase [224].

Step 1: Choose two individuals at random to be the parents, indicated by the numbers p_1 and p_2.

Step 2: For both $routes$ and $trips$, choose a substring at random from a parent.

Step 3: Phase the substring into the appropriate position to create a proto-child.

Step 4: Remove every element that already exists in the remaining parents' proto-child. As a result, an array with the components the proto-child needs is created.

Step 5: For $routes$, place the items of the resultant array from left to right into the unfix position of the proto-child. For $trips$, place the items of the resultant array into the proto-child's unfix position in the corresponding position.

- **Mutation:** The process of mutation involves changing one candidate, usually in a small way. A $bit - flip$ is an example of a mutation; it requires one candidate to have a binary vector of size $a \in \{0, 1\}$. It selects at random (with probability p) whether or not to negate each bit in a. In our method, a solution is modified to create a new solution in order to broaden the algorithm's search space. We choose a mutation rate of w. The following are the two phases to implementing the mutation for the remaining members in the next generation:

  Step 1: Choose a substring at random from the individual.

  Step 2: To make a new route with $routes$, shuffle the element in the substring. To make a new trip with $trips$, flip each element in the substring.

Algorithm 6 Bit-flip Operation for Binary Vector pseudocode

```
1: procedure AlgorithmBitFlipOperationForBinaryVector
2:     function BitFlip(solution, p)
3:         a ← Copy(solution)
4:         for all i in a do
5:             if Random(p) then
6:                 Flip(a[i])
7:                 return a
```

- **Replacement:** This operator selects the newly created individual who will be included in the next generation. Numerous tactics exist, such as generational replacement, in which every new individual becomes the next generation.
- **Repair:** In this stage, we correct the solutions that don't follow the established parameters. The following are the main guidelines that govern the repair process:

  - The shippers' capacity-related journeys are managed by the $trips$ array. The $trips$ array needs to be adjusted in order for the appropriate shipper to return for supply after the present customer if the client's weight has already exceeded theirs throughout the journey.

  - In order to uphold the idea of minimizing the number of $trips$, we first check $trips$ to see if any $trips[i] = 1$ may be eliminated without going against the capacity limit. If this is feasible, we remove them.

  - The corresponding element value in the $trips$ array at the conclusion of each journey must be 1.

  - $trips[i] = 1 \; \forall i = 1 \ldots \; (C + K - 1) \wedge routes[i] < 0$

The researcher must decide on the population's size and the method for selecting participants before implementing GAs [189]. The population's size is determined by the necessary degree of efficacy and efficiency [189]. Stopping conditions may include time, the number of fitness assessments, etc.

GAs are regarded as a significant class of Evolutionary Algorithms (EAs), and the term EA is now used by scholars to refer to the last 30 years of advancement [189]. Evolutionary methods don't have to start from scratch because they can adapt to population changes [144]. Individuals may be chosen in a deterministic or stochastic manner. When you have limited time for implementation, the deterministic approach is more suitable as it is faster than the stochastic approach [144]. High fitness individuals are more likely to be selected as parents or as members of the next generation [34].

4.3.2.2 Scatter Search (SS)

Combining solutions to create new ones is the definition of SS [62]. It requires a collection of points known as a $reference\ set$ ($Ref\ Set$) [141], which includes useful solutions [62]. In order to create new points, SS mixes the reference points [62].

Algorithm 7 Genetic Algorithm pseudocode

```
1: procedure AlgorithmGenetic
2:     function GeneticAlgorithm(problem, populationSize)
3:         population ← initPopulation(populationSize)
4:         while terminationConditionNotMet() do
5:             newPopulation ← createEmptyPopulation(populationSize)
6:             bestCandidate ← getBestCandidate(population)
7:             AddToPopulation(newPopulation, bestCandidate)

8:             for i ∈ {2, ..., populationSize} do
9:                 tempSolution ← Selection(population)
10:                tempSolution ← getFeasibleNeighbour(tempSolution)
11:                AddToPopulation(newPopulation, tempSolution)
12:                population ← newPopulation
13:                return getBestCandidate(population)
```

The following are the primary steps in the SS method [62] [12]:

- Create p solutions and a $reference\ set$ ($Ref\ Set$) with b distinct solutions;
- Sort the solution in the $reference\ set$ ($Ref\ Set$) from best to worst value according to the objective function value;
- Set $newSolutions =$ true;
- While ($newSolutions$) Do
  - Generate $newSubsets$ from $Ref\ Set$, each one of them contains m solutions;
  - Set $newSolutions =$ false;
  - While ($newSubset \neq \emptyset$) Do
    - ⋆ Select the next subset S;
    - ⋆ Using the procedure of solution combination on S to obtain one or more new solutions x;
    - ⋆ Add x to $Ref\ Set$, eliminate the worst value, rearrange $Ref\ Set$, and set $newSolutions =$ true if the value of the new solution is better than the worst value in the $Ref\ Set$;
    - ⋆ Delete S from $newSubsets$;
  - End While
- End While

A subset generation technique, an improvement method, a reference set update method, a diversification generation method, and a solution combination method are the five approaches that make up SS [62]. SS employs deterministic techniques to produce novel solutions and efficiently employs techniques to search in neighborhoods of intensification and diversification [141].

4.3.2.3 Particle Swarm Optimization (PSO)

PSO, a population-based stochastic optimization technique, was first created by Eberhart and Kennedy in 1995 with the goal of mimicking the social behavior of fish schools and bird flocks [216]. The best way to visualize this is as a flock of birds looking for food [102]. Some will follow the first bird that finds food, while others may look for other food sources. The birds don't know where the food is at first, but they always know how far away it is. The bird closest to the meal will lead the others. A bird will use its collective knowledge and past experiences to find food while it is hunting [204].

There are a number of particles in the PSO algorithm. A population of random $candidate$ solutions initializes each particle's potential solution [4]. Every particle has a $position$ and a $velocity$. The particles traverse the solution space repeatedly and at random speeds. Every particle in the N-dimensional space travels through the solution space at a starting velocity. The particles' velocity is then calculated to determine where they should be in the swarm. Every particle will recall its previous best position. The best move made by all particles over the entire iteration is referred to as the global best G_{best}, while the best location of the present particle is P_{best}. As stated by Santosa [25], the search space's i^{th} particle's $velocity$ is updated as

$$V_i(t) = V_i(t-1) + c_1 r_1 (P_{best(i)} - X_i(t-1)) + c_2 r_2 (G_{best(i)} - X_i(t-1)) \tag{4.3.8}$$

Where, X = Particle position, V = Particle velocity, i = Particle index, $t = t^{th}$ iteration, $P_{best(i)}$ = Local best from i^{th} particle, G_{best} = Global best from all swarm, c_1, c_2 = Learning factor, r_1, r_2 = Random number

Where t denotes the current iteration and $P_{best(i)}$ denotes the local best of particle i. Additionally, $G_{best(i)}$ indicates the global best position of particle i. The particle's social and individual achievement are developed using coefficients c_1 and c_2. The random numbers r_1 and r_2 are then defined inside the interval $[0, 1]$. Using equation (4.3.8), one can determine the particle's present velocity by taking into account its past velocity. Equation (4.3.9) then governs the motion of the particles as they approach a new location. The $local$ best solution takes the place of the $global$ best solution if the (P_{best}) value is less than the global best value (G_{best}).

$$X_i(t) = V_i(t) + X_i(t-1) \tag{4.3.9}$$

Choosing the appropriate parameters for an algorithm is one of the things that influences how well it performs in solving optimization issues. According to Eberhart & Kennedy [240], the particle will surpass the search space's bound if c_1 is comparatively larger than the social component c_2. In contrast, particles will locate an early position before reaching the local optimal if c_2 is bigger than c_1. Ratnaweera [4] suggested the time-varying acceleration coefficient as a method for figuring out parameters in the PSO since $exploitation$ and $exploration$ are expected to be balanced to optimize multi-objective optimization.

Consequently, the inertia coefficient (ω) and the time-variant acceleration coefficients (c_1 and c_2) are utilized in this study for the cognitive and social acceleration throughout iteration. Increasing the global search at the earliest optimization time and motivating the particles to meet the global optima at the end of the search are the goals of enhancing this parameter. Equations (4.3.10), (4.3.11), and (4.3.12) provide the mathematical formulation of this idea, respectively [14].

$$\omega_{curr\_iter} = (\omega_{max} - \omega_{min}) * \frac{max\_iter - curr\_iter}{max\_iter} + \omega_{min} \tag{4.3.10}$$

$$c_1 = (c_{1max} - c_{1min}) * \frac{curr\_iter}{max\_iter} + c_{1min} \tag{4.3.11}$$

$$c_2 = (c_{2max} - c_{2min}) * \frac{curr\_iter}{max\_iter} + c_{2min} \tag{4.3.12}$$

Where $curr\_iter$ is the number of iterations that are currently allowed, max_iter is the maximum number of allowed iterations, and c_{1max}, c_{1min}, c_{2max}, and c_{2min} are constants.

PSO is a population-based metaheuristic technique that updates each person's position and velocity based on swarm behavior. The following is a list of steps for utilizing PSO to solve VRP [215]:

1. The number of particles, initial routes, c_1, c_2, maximum number of iterations, initial position, and initial velocity should all be determined for the first initialization.
2. Calculate each particle's total distance.
3. Utilize the particle in G_{best} that has the least cumulative distance. Use the initial total distance as P_{best} for each particle.

4. Until the halting requirement is met, keep doing the following actions. The maximum number of iterations serves as the research's stopping criterion.

   a. Employ P_{best} and G_{best} to utilize Equation (4.3.8) to update each particle's velocity. Next, use Equation (4.3.9) to update each particle's position with the new velocity.

   b. Calculate each particle's total distance.

   c. Select the particle that has the least cumulative distance and assign it to G_{best}. Consider the initial total distance for each particle as P_{best}.

   d. Verify the end criterion. Proceed to step "a" once more if the stopping criterion is not met. Otherwise, stop.

4.3.2.4 Ant Colony Optimisation (ACO)

ACO is a subset of swarm intelligence that imitates how ants use the shortest paths to carry food from the source to the colony, or nest [106] [131]. It solves combinatorial optimization problems with dummy ants rather than actual ants.

Ants explore the area surrounding the nest in a random manner at first until they come across food. Prior to transporting the food to the nest, they assess its quality and quantity [34]. Real ants establish their own routes and communicate with one another via pheromones. Since pheromones are chemicals, the pheromone on a given path will rise with each use by an actual ant. Consequently, there will be a greater likelihood of additional ants selecting this path [106]. Pheromone production is dependent on food quality and quantity, therefore it will assist other ants in determining the quickest path to the food source [34].

Route construction, trail update, and route improvement techniques are included in the ACO heuristic in [106]. The following are the ACO algorithm's primary steps [132]:

- Initialization;
- While (Stopping criteria is not satisfied), Do:
  - Construct ant solution;
  - Apply local search (optional step);
  - Update pheromones;

When the arc arises in a good solution, the pheromone in ACO correlates to a value associated with an arc (or edge) and this value grows [104]. The shortest path will remain after food transportation, whereas the longer routes will be forgotten [106].

4.3.2.5 Path Relinking (PR)

Path Relinking (PR) is regarded as an addition to Scatter Search (SS) and is intended to include the search for diversity and intensification [62] [141]. Rather than merging the chosen answers to create new ones, PR creates new paths between them [141]. It is applied to each local solution to enhance it using greedy randomized adaptive search procedure (GRASP) as an intensification method [140].

4.3.3 Hybrid Metaheuristics

Combining one metaheuristic with additional metaheuristics, or with portions of other metaheuristics, or with operational research methodologies is how it is defined [34]. In terms of cutting CPU time and enhancing solution quality, hybrid optimizers work better [86]. Pure metaheuristics and hybrid metaheuristics both benefit from each other [85].

The literature has several categories for hybrid metaheuristics. In [85], Raidl et al. categorize hybrid metaheuristics according to the following rules:

1. Type of algorithm: It could be a blend of elements from several metaheuristic strategies or a metaheuristic combined with generic methods from artificial intelligence and operational research.

2. Level of hybridization: Pairings at high and low levels.
3. Order of the implementation: The implementation order is batch execution when the method is run in a parallel, entangled, or orderly fashion.
4. Control strategy: Collaborative (each one is not part of the other although they exchange information) or integrative (one of them is a component of the other algorithm).

Hybrid metaheuristics are divided into two categories [34]: integrative combinations and collaborative combinations, which exchange information throughout sequential or parallel runs. There are two ways in which the hybridization of Branch and Bound (B&B) and metaheuristic can occur:

- Branch and Bound (B&B) inside metaheuristics like ant colony optimisation (ACO) or greedy randomised adaptive search procedure (GRASP) to increase the metaheuristics' effectiveness.
- Using metaheuristic within B&B to shorten the search tree and cut down on CPU time [35].

We can employ precise techniques that make use of heuristics or metaheuristics. For instance, in [235], they employ reactive tabu search (TS) in conjunction with branch and bound (B&B), and they assert that this cooperation results in extremely efficient or reasonable CPU time.

In order to find better results, hybrid algorithms combine several heuristic or meta-heuristic approaches. A hybrid algorithm might, for instance, start with a solution produced by a constructive heuristic and refine it using a metaheuristic. As an alternative, a hybrid algorithm might be explored by combining metaheuristic techniques with neighborhood search.

Using both individual algorithms and their synergy to create a more successful hybrid system is the goal of hybrid metaheuristics [85].

4.4 Neighbourhood Search Methods

Neighbourhood search methods tweak an existing solution slightly in order to explore the solution space. A neighbourhood search algorithm might change the sequence of consumers in a route, add or remove customers from a route, or swap customers between routes in the context of VRPTW in order to explore the space of neighboring solutions. While making sure that the time window restrictions are met, these changes can be implemented [109] [183].

4.4.1 Variable Neighbourhood Search Basic Schemes (VNS)

Since $min\{f(x) \mid x \in X, X \subseteq S\}$ may be solved (see Combinatorial Optimization Problems 2.1 Chapter 2), let x represent the solution. The neighborhood $N_k(x)$, or the k^{th} neighborhood of x, is denoted by $N_k(x)$ and contains all vectors that could be derived from x when a modification k is made to x. Denote the set of solutions that belong to neighborhood $N_k(x), k = 1, \ldots, k_{max}$ [178].

If there is no solution better than x that belongs to $N_k(x)$, that is, $f(x) < f(x'), \forall x' \in N_k(x)$, then x is a $local\ minimum$ with respect to N_k. The best possible solution, $f(x_{opt}) < f(x), \forall x \in X$, where $X \subseteq S$, is the optimal solution, also known as the global minimum, or x_{opt}.

VNS has a straightforward core (Algorithm 8) where two solutions' qualities are compared. One is referred to be the incumbent (x), and another x' is a member of the k^{th} neighborhood around x. In the event where $f(x') < f(x)$, the new value (as a new incumbent) is updated and $k = 1$ is set to begin from the first neighborhood once more. If not, $(k = k + 1)$ is used to select the next neighborhood in the search. Distinct varieties of VNS are produced by VNS using distinct neighborhood structures in three different ways: deterministic, stochastic, and a combination of both. This section contains descriptions of each one's basic data and algorithms.

Algorithm 8 Neighborhood Change algorithm pseudocode

```
1: procedure AlgorithmNeighborhoodChange
2:     function NeighborhoodChange(x, x', k)
3:         if f(x') < f(x) then
4:             x ← x'; k ← 1    //Make a move
5:         else
6:             k ← k + 1    //Next neighborhood
```

4.4.1.1 Variable Neighbourhood Desert (VND)

Local Search (LS)

The literature examines two local search strategies: first improvement and best improvement.

- Best Improvement: The finest improvement begins with an initial solution, looks throughout the neighborhood, and then settles on the best value—which is regarded as the local minimum—found there. Since the entire neighborhood must be checked, this procedure takes a while. Algorithm 9 provides the best improvement algorithm.

Algorithm 9 Best Improvement algorithm pseudocode

```
1: procedure AlgorithmBestImprovement
2:     function BestImprovement(x)
3:         repeat
4:             x' ← x
5:             x ←arg min_{y∈N(x)} f(y)
6:         until (f(x) ≥ f(x'))
```

- First Improvement: It begins with a preliminary fix and advances as soon as a more effective one is discovered. After that, the investigation is restarted. It will come to an end if a better solution cannot be discovered in the current neighborhood. Algorithm 10 contains the first improvement algorithm.

Algorithm 10 First Improvement algorithm pseudocode

```
1: procedure AlgorithmFirstImprovement
2:     function FirstImprovement(x)
3:         repeat
4:             x' ← x;    i ← 0
5:             repeat
6:                 i ← i + 1
7:                 x ←arg min{f(x), f(x_i)}, x_i ∈ N(x)
8:             until (f(x) < f(x_i) or i =| N(x) |)
9:         until (f(x) ≥ f(x'))
```

Variable Neighbourhood Desert (VND)

The l_{max} neighborhoods N_l, where $l = 1, \ldots, l_{max}$ are deterministically altered using VND heuristics. After investigating the neighborhood $N_l(x)$, where x is an incumbent solution, it determines whether to proceed to $x' \in N_l(x)$. Algorithm 11 provides the stages for the fundamental (sequential) best improvement VND. Sequential (or basic), nested, and mixed nested VNDs are the three different forms [180] [176] [155]. Below is a list of essential details:

- Sequential VND: It operates in this manner:

  - Specify the l_{max} neighborhood structure as an order that will be applied to the search.

Algorithm 11 Variable Neighbourhood Desert (VND) algorithm pseudocode

```
1: procedure AlgorithmVND
2:     function VND(x, l_max)
3:         repeat
4:             l ← 1
5:             x' ← x
6:             repeat
7:                 x' ←arg min_{y∈N_l(x)} f(y)    // Find the best neighbor in N_l(x)
8:                 NeighborhoodChange(x, x', l)    // Change neighborhood
9:             until l = l_max
10:        until (f(x) ≥ f(x'))
```

- Start by applying local search (LS) to the first neighborhood structure in the list, then to the second neighborhood, and so forth.

- Begin again with the first neighborhood structure on the list if a better solution is discovered.

- If there isn't a better solution in the $N_{l_{max}}$ neighborhood, the search will end.

It is concluded that the resulting solution is a local minimum with regard to all neighborhood structures in l_{max}. The total cardinalities $\mid N_l(x) \mid$ make up its cardinality. For instance, we must visit $2n$ points if $l_{max} = 2$ and each neighborhood has n points (see [46]).

- Nested VND: Using nested VND, we may conduct a local search at any point in the second neighborhood with regard to the first neighborhood, assuming we have 2 neighborhood structures. Thus, we shall visit $\mid N_1(x) \mid \times \mid N_2(x) \mid$ points using nested VND. Nested VND neighborhoods, as can be seen, take up more CPU time and are larger than sequential VND (see [46]).

- Mixed VND: It begins with nested VND and subsequently transitions to sequential VND. This is done to reduce the huge number of visited points produced by nested VND (see [46]).

4.4.1.2 Reduced Variable Neighbourhood Search (RVNS)

It is helpful for really big instances and searches the neighborhood $N_k(x)$ in a stochastic manner. As stopping criteria, the maximum number of iterations, the maximum number of iterations between two improvements, or the CPU time (t_{max}) are utilized. A perturbation or shaking step and a neighborhood change (Algorithm 8) step make up RVNS. The parameters are t_{max} and k_{max}. The optimal value for k_{max} is 2 or 3, as supported by extensive experimental analysis [213]. Algorithm 12 provides the RVNS algorithm. See [11] [213] for additional details on RVNS and its applications.

Algorithm 12 Reduced Variable Neighbourhood Search (RVNS) algorithm pseudocode

```
1: procedure AlgorithmRVNS
2:     function RVNS(x, k_max, t_max)
3:         repeat
4:             k ← 1
5:             repeat
6:                 x' ← Shake(x, k)    // Shaking
7:                 NeighborhoodChange(x, x', k)    // Change neighborhood
8:             until k = k_max
9:             t ← CpuTime()
10:        until t > t_max
```

4.4.1.3 Basic Variable Neighbourhood Search (BVNS)

Deterministic and stochastic neighborhood changes are combined in BVNS. The greatest CPU time used in the search (t_{max}) could once more be the stopping criterion. Shaking, local search (First improvement (Algorithm 10) or Best improvement (Algorithm 9), and neighborhood change (Algorithm 8) are the three primary processes of BVNS. Algorithm 13 provides a summary of the BVNS algorithm.

Algorithm 13 Basic Variable Neighbourhood Search (BVNS) algorithm pseudocode

```
1: procedure AlgorithmBVNS
2:     function BVNS(x, k_max, t_max)
3:         repeat
4:             k ← 1
5:             repeat
6:                 x' ← Shake(x, k)    // Shaking
7:                 x'' ← FirstImprovement(x')    // Local Search
8:                 NeighborhoodChange(x, x'', k)    // Change neighborhood
9:             until k = k_max
10:            t ← CpuTime()
11:        until t > t_max
```

4.4.1.4 General Variable Neighbourhood Search (GVNS)

It is a mixture of variable neighbourhood desert (VND) and basic variable neighbourhood search (BVNS). Put otherwise, the VND algorithm presented in Algorithm 11 takes the role of the local search in Algorithm 13. The Algorithm 14 contains the GVNS stages. See [175] [46] [155] for additional details on GVNS and its productive applications.

Algorithm 14 General Variable Neighbourhood Search (GVNS) algorithm pseudocode

```
1: procedure AlgorithmGVNS
2:     function GVNS(x, l_max, k_max, t_max)
3:         repeat
4:             k ← 1
5:             repeat
6:                 x' ← Shake(x, k)    // Shaking
7:                 x'' ← VND(x', l_max)    // Using VND instead of Local Search
8:                 NeighborhoodChange(x, x'', k)    // Change neighborhood
9:             until k = k_max
10:            t ← CpuTime()
11:        until t > t_max
```

4.4.1.5 Skewed Variable Neighbourhood Search (SVNS)

It looks into the neighborhoods that are distant from the incumbent using a diversification search. SVNS uses a function $\rho(x, x'')$ to calculate the distance between the incumbent x and the derived local optimal x''. Then, based on the value of the parameter α and the function $\rho(x, x'')$, it decides whether to move or not (see Algorithm 16). The user selects the value of parameter α in accordance with the testing outcomes. Algorithm 15 provides the SVNS steps, while Algorithm 16) provides the Neighborhood Change for SVNS steps. For additional details on a few SVNS applications, read [101] [138].

Algorithm 15 Skewed Variable Neighbourhood Search (SVNS) algorithm pseudocode

```
1: procedure AlgorithmSVNS
2:     function SVNS(x, k_max, t_max, α)
3:         repeat
4:             k ← 1,   x_best ← x
5:             repeat
6:                 x' ← Shake(x, k)    // Shaking
7:                 x'' ← FirstImprovement(x')    // Local Search
8:                 NeighborhoodChange(x, x'', k, α)    // Change neighborhood
9:             until k = k_max
10:            if (f(x) < f(x_best)) then x_best ← x
11:                x ← x_best
12:                t ← CpuTime()
13:                until t > tmax
```

Algorithm 16 Neighborhood Change for SVNS algorithm pseudocode

```
1: procedure AlgorithmNeighborhoodChangeSVNS
2:     function NeighborhoodChangeSVNS(x, x'', k, α)
3:         if f(x'') − αρ(x, x'') < f(x) then
4:             x' ← x'', k ← 1    // Make a move
5:         else
6:             k ← k + 1    // Next neighborhood
```

4.4.1.6 Variable Neighbourhood Decomposition Search (VNDS)

By dividing Variable Neighborhood Search (VNS) into two levels through problem decomposition, VNDS is regarded as an extension of Basic Variable Neighborhood Search (BVNS) [174]. It is employed to resolve significant problems cases. Algorithm 17 presents the steps of the algorithm. The running time of the decomposed problems that VNS solves at the second level is denoted by t_d. Check [133] [184] [103] [117] for additional details on VNDS and its uses.

Algorithm 17 Variable Neighbourhood Decomposition Search (VNDS) algorithm pseudocode

```
1: procedure AlgorithmVNDS
2:     function VNDS(x, k_max, t_max, t_d)
3:         repeat
4:             k ← 2,   x_best ← x
5:             repeat
6:                 x' ← Shake(x, k); y ← x'/x    // Shaking
7:                 y' ← VNS(y, k, t_d); x'' ← x'/y ∪ y'    // Using Variable Neighborhood Search
8:                 x''' ← FirstImprovement(x'')    // Local Search
9:                 NeighborhoodChange(x, x''', k)    // Change neighborhood
10:            until k = k_max
11:        until t > tmax
```

Regarding VNS, visit [10] [159] [158] [157] [156] for additional details.

Following the description of several kinds of algorithms and strategies for solving VRPs using exact methods, classical heuristics, and metaheuristics, the following are some observations regarding the techniques displayed in the table 4.1.

	Algorithms	Remarks
Exact algorithms	Branch and bound method [110] [125]	The depth of the branch and bound tree depends on the efficiency.
	Set segmentation method [111] [219]	Difficult to establish the lowest cost for every solution.
	Dynamic programming method [74] [226]	Effective for small tasks, but difficult to consider concrete demands such as time constraints.
	Integer programming algorithm [59] [38]	High accuracy, time-consuming, and complicated.
Classical heuristic algorithms	Savings algorithm [13] [197]	Computes quickly, however it is difficult to find the best solution.
	Sweep algorithm [167] [6]	Appropriate for a small number of routes with an equal number of consumers for each route.
	Two-phase algorithm [57] [154]	Difficult to find the best option.
Metaheuristic algorithms	Tabu search algorithm [105] [23] [108]	Has good local search capabilities, but is time-consuming and is dependent on the initial solution.
	Genetic algorithm [197] [27]	Has a good global search capability, computes quickly, and is difficult to achieve the global optimal solution.
	Iterated local search [88] [112]	Has a quick convergence rate and a low computational complexity.
	Simulated annealing algorithm [236] [21]	Slow convergence rates, with carefully selected configurable parameters.
	Variable neighborhood Search [8] [150]	Is appropriate for large, difficult optimization problems with constraints.
	Ant colony algorithm [91] [198] [232]	It has a good positive feedback mechanism, yet takes time and is vulnerable to standstill.
	Neural network algorithm [225] [205]	It computes quickly, has sluggish convergence, and is easily stuck in a local optimum.
	Artificial bee colony algorithm [232] [22]	It is associated with the piecewise linear cost approximation and achieves a rapid convergence speed.
	Particle swarm optimization [217] [3] [202]	Is robust and has a quick searching speed, resulting in easy premature convergence.
	Hybrid algorithm [234] [74] [13] [108] [91] [95] [238]	Is straightforward, with quick optimization and minimal calculation.

Table 4.1: Observations regarding the techniques for solving VRPs.

4.5 Free/Open-Source Solvers

There have been developed some free/open-source solvers that can handle various VRP kinds with varying numbers of consumers. The solvers that implement some of the more sophisticated and promising algorithms are summarized here, along with some extra information we learned from the cited sources and a brief explanation of each solution [143].

- **VRPLIB** [118] A Python module called VRPLIB is used to interact with instances of the Vehicle Routing Problem (VRP). Reading VRPLIB and Solomon instances and solutions, producing instances and solutions in the VRPLIB style, and downloading datasets (instances) and well-known solutions from CVRPLIB are the primary functionalities.
- **Jsprit** [200] Jsprit is an open-source toolkit for Java that may be used to solve vehicle routing problems (VRP) and rich traveling salesman (TSP) problems. The ruin-and-recreate principle, which involves destroying some of the answers and creating them in a different way, is applied by the main metaheuristic algorithm. You can go to the referenced site to learn more about the algorithm. The library is actively being developed, and ODL Studio makes use of its software.
- **Open-VRP** [162] Open-VRP is a platform designed to let academics, corporations, hobbyists, and students alike simulate and solve VRP-like challenges. It involves the use of tabu search and greedy heuristics. It was written in LISP, and more than five years have passed since any development was done on it.
- **PyVRP** [186] PyVRP is a cutting-edge, open-source solution for vehicle routing problems (VRPs), combines the global search capabilities of genetic algorithms with the fine-tuning efficiency of (local) search methods. It is made to be as simple to use as possible and offers state-of-the-art performance and speed in a highly configurable Python package. Python can be used by users to modify several parts of the algorithm, such as operator selection in the local search, granular neighborhoods, crossover techniques, and population management. Furthermore, users can build and install PyVRP straight from the source code for more complex use cases, including supporting more VRP variants.
- **OptaPlanner** [209] OptaPlanner is a constraint fulfillment solver, written in Java,. It provides examples and supports a wide range of issues. While it remains very active, its efficacy in solving CVRP has not shown much in the testing. However, because it is a universal solver, the results were really good.
- **SYMPHONY** [214] The COIN-OR website hosts SYMPHONY, an open-source C solver for mixed-integer linear programming (MILPs). Additionally, it allows for parallel execution.
- **OR-Tools** [81] Open-source combinatorial optimization software, known as OR-Tools, looks for the optimal answer to a given problem from a wide range of potential solutions. Google is developing it in Python. These kinds of issues typically have a large number of potential solutions—too numerous for a computer to look up. To get around this, OR-Tools finds an optimal (or nearly optimal) solution by reducing the search set using state-of-the-art algorithms. Solvers for vehicle routing, graph algorithms, linear and mixed-integer programming, and constraint programming are all included in OR-Tools.
- **VRP Spreadsheet Solver** [55] One open-source, unified platform for representing, solving, and visualizing the VRP results is the Microsoft Excel workbook VRP Spreadsheet Solver. It combines metaheuristics, public GIS, and Excel. It is claimed to be able to resolve VRP for up to 200 customers. Nevertheless, we could only receive it by registering and attending certain VeRoLog workshops or conferences, as it is exclusively accessible to VeRoLog members.
- **VROOM** [185] Several libraries and frameworks are included in VROOM, particularly for resolving dynamic VRPs using Solomon I1 heuristics, clustering heuristics using spanning trees, Christofides heuristic techniques. The goal of the open-source, C++20 optimization engine VROOM is to quickly compute effective solutions to a variety of real-world VRPs.
- **VRPH** [83] A collection of free and open-source heuristics for the CVRP is called VRPH. It contains multiple sample apps that may be used to solve VRP situations with thousands of client locations quickly and effectively. It was created as a component of Chris Groer's dissertation under Bruce Golden's guidance at the University of Maryland. Since its initial release in 2009, it has not been updated and is currently hosted on the COIN-OR website.

Chapter 5

Evaluation of the Methods

5.1 Proposed Algorithms

The research design for an evaluation of traditional heuristic and metaheuristic approaches to solving the vehicle routing problem with time windows (VRPTW) is covered in this chapter. The benchmarking data sets utilized for the VRPTW issue instances are Solomon (1987) [212] and Homberger & Gehring (1999) [212]. Solomon (1984) developed and published a set of test problems that evaluated the computational capability of different optimization solution strategies applied to the VRPTW [211]. To do that, we will employ and put into practice several algorithms in order to develop several models that will compute the best routing solution and its associated routing cost, display them on a graph, and compare the outcomes with the $best - known\ solutions$ (BKS).

These are the names of the algorithms we will use to develop the multiple models, compare the outcomes to each model, and plot them in the graph:

- **Model I (HGS)** - We will employ a metaheuristic hybrid algorithm in this model dubbed HGS, which combines the fine-tuning effectiveness of local search techniques with the global search capabilities of genetic algorithms (GA). (The fields that fall under local search are Hill climbing (HC), simulated annealing (SA) (fit for local or global search), tabu search (TS), late acceptance hill climbing (LAHC), and reactive search optimization (combining machine learning and local search heuristics)).
- **Model II (GLS)** - We will solve the difficulties in this model using the two-step Google OR-Tools Open-Source Solver. Provide a traditional heuristic method known as the First Solution Strategy, or initial solution, as the first step. Savings algorithms, sweep algorithms, Christofides algorithms, and other techniques can be applied in the first solution strategy as optional. The second step involves using guided local search (GLS), a metaheuristic technique, to keep improving it until the time limitation is reached.
- **Model III (ACO)** - We are going to develop ant colony optimizer (ACO) algorithm, in this model. A multiple ant colony system (MACS) will be using here to handle multiple objective function: first colony minimizes the number of vehicles while second colony minimizes the traveled distances, a metaheuristic method local search algorithm will be used to look for feasible neighboring solutions to improve the existing solution.
- **Model IV (SA)** - We will develop the simulated annealing algorithm (SA), initialising with greedy randomized adaptive search procedure (GRASP) as initiator, a metaheuristic technique, to this model.

5.2 Hybrid Genetic Search (HGS) Methods

A potential risk of GA is that individuals could get stuck in local optima, which is typically brought on by designs that don't keep the population diverse enough or by having too few search agents. In this work, we leverage local search's

improved neighborhood search capability to help individuals overcome the challenge of being trapped in a local optima. We do multiple iterations of the local search (LS) method, starting with the elite individuals identified by GA, in order to find better solutions. Then, the inputs are replaced by these solutions in the next generation. GA and LS algorithms have already been covered in Chapter 4, Section 4.3.2.1, and Section 4.3.1.10; in HGS, we are merely combining these two algorithms. Figure 5.1 shows how the procedure is carried out.

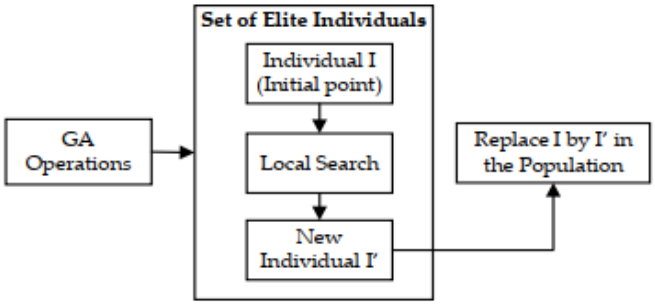


Figure 5.1: Combination of genetic algorithm (GA) and local search (LS) [218].

The search method used by hybrid genetic search (HGS) and genetic algorithm (GA) algorithms is similar. The sole distinction is that, before producing a new generation, HGS keeps using local search (LS) to identify neighbors with better fitness values. This should theoretically guarantee that the HGS has a higher probability than the previous version of avoiding the local optima. Our trials have further supported this. However, the overall time needed to look for each solution grows dramatically because many individuals have to carry out local search operations after genetic operations.

The hybrid genetic search (HGS) algorithm for vehicle routing problems (VRPs) combines the fine-tuning effectiveness of local search techniques with the global search capabilities of genetic algorithms (GA). High-quality solutions can be obtained by efficient $exploration$ and $exploitation$ of the search space made possible by this hybrid method.

HGS functions in pseudocode as follows:

Algorithm 18 Hybrid Genetic Search algorithm

1: **procedure** AlgorithmHybridGeneticSearch
Input: initial solutions $S_1, \ldots, S_n$
Output: the best-found solution s^*
2: Set s^* to the initial solution with the best objective value
3: **repeat** until stopping criterion is met:
4: Select two parent solutions (s^{p_1}, s^{p_2}) from the population using $k-ary$ tournament.
5: Apply crossover operator XO to generate an offspring solution $s^o = XO(s^{p_1}, s^{p_2})$
6: Improve the offspring using a search procedure LS to obtain $s^c = LS(s^o)$
7: Add the candidate solution to the population
8: **if** s^c has a better objective value than s^*: **then**
9: $s^* \leftarrow s^c$
10: **if** population size exceeds maximum size: **then**
11: Remove the solutions with the lowest fitness until the population is at minimum size
12: **return** s^*

In order to accommodate time windows, we modified the HGS-CVRP code [228] and included construction heuristics, an enhanced crossover process, and a more intense local search. In addition to performance enhancements, we introduced caching mechanisms and pre-checks for local search operators. Since we needed 1-decimal precision, we multiplied distances (and time windows) by 10 and employed integer arithmetic, which resulted in a significant speedup. Furthermore, instead of using a C++ vector of vectors, we constructed the distance matrix as a flat array, which is more efficient. Lastly, we eliminated pointless route-duration analyses.

5.2.1 Implemented HGS Algorithm Description

Here's how the HGS algorithm functions (briefly). An initial set of solutions supplied as algorithmic input establishes the $population$ of solutions that HGS maintains. Using a $k-ary$ tournament, the algorithm chooses two parent solutions from the population for each iteration of the $search$ loop, giving preference to solutions with better $fitness$. Then, using these two parent solutions, a $crossover$ operator creates an offspring solution that has characteristics from both parents. Following the $crossover$, a search procedure is used to further refine the offspring solution. First, the $candidate$ solution that is produced is included in the population. The candidate solution is also recorded as the new $best\ solution$ if it outperforms the best solution that has been identified thus far. A survivor $selection$ mechanism eliminates the least fit solutions after the population reaches its maximum size, continuing until the population returns to its minimal size. The algorithm returns the $best\ solution$ discovered after iterating until a given $stopping\ criterion$ is satisfied.

The HGS approach will be explained theoretically in the following sections, and Chapter A describes the components and realistic implementations of the HGS algorithm in programming format.

5.2.2 HGS Supporting Time Windows

Following the original HGS paper for VRPTW [218], we support time frames. When using time windows, the vehicle needs to reach a customer within the earliest and latest arrival times. Once there, it needs to stay for a specific amount of time for servicing before moving on to the next customer. After, we put into practice the $time-warp$ principle [242], which allows the vehicle, it arrives at a customer too late, to "travel back in time" to the earliest arrival time [218]. The goal is increased by multiplying the total $timewarp$ by a penalty weight. Similar to the capacity penalty, the time-warp penalty weight is initialized at 1 and modified every 100 iterations. If less than 15% of the solutions returned by the local search are viable (concerning time windows), it is increased by 20%; if more than 25% are possible, it is dropped by 15%. Furthermore, we included a $penalty\ booster$ that raises the penalty by 100% if no feasible solution (within time windows) has been discovered thus far. If this is not the case, it may take a long time to find a feasible solution. Compared to employing a greater initial penalty from the beginning, which allows the algorithm to converge too rapidly to poor solutions, harming overall performance, we found that this worked well.

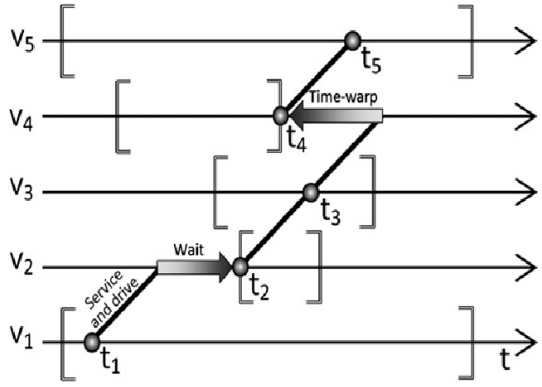


Figure 5.2: An example of a 'Time Warp' and 'Waiting Times' (influenced by Nagata et al. [242]).

By monitoring specific statistics for subsequences of nodes inside a route, which we refer to as TW-data (time window data), it is possible to compute the total $time-warp$ efficiently [218]. In contrast to [218], we precompute (cache) $prefix$ TW-data (corresponding to the start-route: the sequence from the depot up to and including that customer) and $postfix$ TW-data (corresponding to the end-route: starting at the customer and ending at the depot) for each customer in a route. This makes it possible to calculate a route $total\ time-warp$ once a customer is inserted or removed in $O(1)$, but not both. We use a two-level hierarchical data structure to precompute TW-data for smaller route segments between seed customers (one for every four customers), which can be combined to calculate TW-data for longer route segments [96]. This is necessary when there is simultaneous insertion and removal at different positions in a route (e.g. for RELOCATE or SWAP*, see Section 5.2.5).

5.2.3 HGS Construction Heuristics

To ensure that the genetic algorithm succeeds, the primary objective of the initially selected solutions in the collections is to provide diversity. Because of this, [229] starts with random solutions and refines them via local search. In addition to variety, our goal is to swiftly locate feasible solutions of reasonable quality. To this end, we add three more construction heuristics: $nearest$, $farthest$, and $sweep$. Together, these heuristics generate 5% of the 100 original solutions. Our ultimate goal is to reduce the primal integral.

We generate routes one at a time using $nearest$ and $farthest$ (influenced by VROOM [96]), inserting the unassigned customer that is closest to the depot or farthest from it, respectively, to begin the route. After that, we repeatedly identify and insert the unassigned customer and insertion point combination that results in the shortest detour distance and doesn't violate any time windows or vehicle capacity. We go on to the next step if no customers can be added without resulting in infractions. Since these heuristics are deterministic, we provide additional (almost feasible) solutions by permitting each route to exceed the capacity constraint (with a tolerance of 0 to 50) and employ a slight time-warp (with a random tolerance of 0 to 100).

We use $sweep$ to sort the customers based on their angle concerning the depot. One by one, we construct the routes. Until capacity is exceeded, we continue allocating customers to the route starting at the initial position of the sorted list of unassigned customers. First, all customers with shorter time windows (at most half of the planning horizon) are sorted gradually by the latest arrival time to decide the order of the customers within the route. The remaining customers are then added to the partially created route at the position that results in the shortest detour, in an arbitrary order. Keep in mind that this heuristic might create routes that don't follow time windows. We apply the heuristic with a capacity that is randomly decreased by 0% to 40% to construct additional solutions.

5.2.4 HGS Offspring Generation

To produce new offspring from two parents chosen through $binary\ tournaments$, HGS employs a $ordered\ crossover$ (OX) [93]: each parent is the best of two random solutions in the collection based on the $fitness$ (taking equality and diversity into consideration) [229]. The OX includes the missing nodes in the order that they appear in the second parent after randomly selecting a subsequence of nodes from the first parent as a starting point (ignoring any depot visits). The SPLIT [227] [16] algorithm is used to re-insert depot visits into the solution; it ignores time windows. Local search is then used to resolve any time window issues. We modify the maximum number of routes for the SPLIT algorithm in comparison to HGS-CVRP [229]: instead of using the number of routes of the first parent, we utilize all available vehicles because we don't require to reduce the number of vehicles used.

By including a second crossover operator, we enhance the offspring of the previous generation: $Selective\ Route\ Exchange$ $(SREX)$ [241]. To produce offspring solutions, $SREX$ mixes the complete routes from the two parents. To be more precise, a subset of routes from one parent is chosen at random and substituted with corresponding routes from the other parent. From routes with a single parent, any nodes that exist in two routes are eliminated. The distance of the detour is used to insert missing nodes into a route. Depending on which parent route the duplicate nodes are eliminated from, two distinct offspring solutions are generated. We then proceed with the optimal solution (in terms of $penalized\ cost$). Because $SREX$ does not employ the SPLIT algorithm and maintains depot visits, it is a better fit for time windows. Lastly, we generate offspring with both OX and $SREX$, and with the best (in terms of $penalized\ cost$) offspring between these two candidates, we proceed to the local search phase.

5.2.5 HGS Local Search and SWAP* Neighborhood

The actions $SWAP$, $RELOCATE$, $2-OPT$, and $2-OPT^*$ that are detailed in [218] [229] make up the local search. Each move has two nodes (u, v) that can describe it. Only moves characterized by promising arcs (u, v) are taken into account in [218], whereby the measure of spatial and time proximity is determined by an asymmetric measure $\gamma(u, v)$ (Eq. (4) in Vidal et. al [218], which is also expressed in Section 2.2.2.2 Eq. 2.2.20). For most movements (e.g. when swapping u and v), the arc (u, v) does not end up in the resultant solution. Therefore, we employ the symmetric measure $\hat{\gamma}(u, v) = min\{\gamma(u, v), \gamma(v, u)\}$ as a substitute. Only moves involving the Γ "nearest" neighbors in terms of $\hat{\gamma}(u, v)$ are taken into account for each node. We incorporate pre-checks based on bounds to avoid costly TW-data computations (see Section

5.2.2) by avoiding duplicate checking symmetric operators for (u, v) and (v, u). Specifically, if the routes involved in a move currently have 0 time-warp, we then proceed to initial check only if the move will reduce the total distance. If it's not, we may disregard the costly TW-data computation and discard the move, which can never be improved.

Intensification: SWAP* and RELOCATE*

When a new local search is initiated (either after creating a solution or producing new offspring), we run the search with a larger neighborhood modeled after the SWAP* neighborhood for a given $intensification\ probability\ \theta$ [229]. If an improvement is not obtained from any of the basic operators, we then iterate over all pairs of routes with overlapping circle sectors (refer to [229]), where we enforce a minimum circle sector size of 15° to ensure overlap for extensively clustered routes. We attempt the distance-based SWAP*, the TW-data-based SWAP*, and the RELOCATE* operators one after the other for every pair of routes. We go back to the fundamental operators and repeat whether any of these result in an improvement.

A node is attempted to be moved from one route to the optimal position in the other route using the RELOCATE* operator. In the worst scenario, this operator is $O(n^2)$ since it takes into account all RELOCATE moves. By swapping two nodes between the two routes, the SWAP* operator hopes to place them in the optimal position on the other route. The SWAP* operator for the CVRP is precise in that it always determines the best move. This is accomplished effectively (in a total $O(n^2)$ computation) by calculating the top three insertion positions (estimated on distance) for each node in the alternative route in advance. The optimum insertion position after deleting a node (another) remains one of the top three positions, or the deleted node's position, therefore we may safely disregard all other positions.

Even though there is no guarantee of this property with time windows, we can still use the top three insertion positions to serve as a heuristic. The distance-based SWAP* operator only determines the optimal SWAP move based on distance (ignoring time windows) and then attempts this move, operating under the assumption that better moves shouldn't cause large detours. TW-data-based SWAP* assumes that adding a customer and removing a customer from a route would have an independent impact on the overall time-warp, allowing the time-warp penalties from both operations to be applied.

Under this assumption, we can easily precompute the optimum SWAP move and the top three insertion locations for each node (that includes time-warp penalties). Similar to the distance-based SWAP*, the optimum move is the only one for which the entire cost (including real-time-warp penalties) is calculated. In practice, we discovered that this move was effective.

5.2.6 HGS Growing Population and Neighborhood Size

Sorting instances according to their $long$ or $short$ routes (above 25 customers per route, calculated by average demand and vehicle capacity) and having at least one customer with big-time windows (above 70% of the horizon) proved useful. We adjust the minimum population size μ and neighborhood size Γ growth schedules as well as the intensification probability θ based on these attributes.

We utilize $\theta = 15\%$, $\mu = 25$, $\Gamma = 40$ for instances with $long$ paths, and we grow both μ and Γ by 5 every 10000 iterations.

We use $\theta = 100\%$ and only increase the population size μ for instances with $short$ routes. For instances $without$ big time windows, we set $\Gamma = 40$ and increase $\mu = 25$ by 5 every 10000 iterations; however, if an instance has (any) big time windows, we put $\Gamma = 20$, and increase $\mu = 25$ by 5 only every 20000 iterations, to account for faster iterations.

This allows for greater effectiveness because of the flexibility provided by extended time windows, as more time is spent on $intensification$. Without making any progress after 10000 iterations, we restart the process to create a fresh population without changing any of the parameters.

Refer to Chapter A, which contains descriptions of the components and realistic implementations of the HGS algorithm in programming format.

5.3 Simulated Annealing (SA) Methods

We employed a simulated annealing technique to solve the VRPTW issues, initializing with a random feasible solution that has been initialized using a randomized greedy function and neighbor search. We will discuss the entire case as it is detailed; a brief explanation of each of these two approaches is already provided in Chapter 4 Sections 4.3.1.2 and 4.3.1.7.

5.3.1 Implemented SA Algorithm Description

Initially, we need to describe what a single solution's representation is. The solution in this instance is shown as a list of lists. One solution with three possible routes is represented, for instance, by [[0, 4, 1, 2, 0], [0, 5, 3, 0], [0, 6, 0]]. Customers 4, 1, and 2 are visited by the first vehicle, followed by 5 and 3 by the second, and customer 6 by the third. Naturally, a depot is represented by customer 0, where vehicles begin and end their routes.

The second thing we have to do is specify which two solutions are neighbors and which aren't. We "move" a selected customer to produce a single neighbor of some solution. It can be placed "inside" the present vehicle or in a specific position within another vehicle. Let's use the previously stated solution [[0, 4, 1, 2, 0], [0, 5, 3, 0], [0, 6, 0]] as an example. Assume this solution's neighbors include [[0, 4, 1, 2, 0], [0, 5, 6, 3, 0]] (The third route has moved to the second for Customer 6), [[0, 4, 2, 0], [0, 5, 3, 0], [0, 6, 1, 0]] (The first route has moved to the third for Customer 1), etc. Observe how the third vehicle in the first example "vanished" since it had no clients left to serve. Naturally, we must ensure that neighbors meet specified constraints when producing neighbors.

We generate a random neighbor of the current solution at every step of the method we employ. We select one of the vehicles at random first, and then we select a customer at random from the selected vehicle. Then, as previously mentioned, we move this customer to a random position. It should be noted that this method of giving customers an option won't result in a uniform distribution. Stated differently, not every customer will be selected equally. The reason behind this is that we select a vehicle initially. Customers in vehicles with fewer customers will therefore have a higher chance of getting selected, which is precisely what we want since it would allow us to "move" toward solutions that have fewer routes, which is what we desire.

Similar to the majority of heuristic algorithms, we required an initial solution for our search. We have employed to produce such a solution using a greedy algorithm. The only thing this algorithm does is force vehicles to visit first customers who have a minimum total readiness time and distance from the vehicle's current position. Although this "choice" might appear a little strange at first, it produced better results than if we had merely taken the customer's distance into account.

The time the vehicle starts providing a service at a specific customer's location along a route is equivalent to the sum of:

1. The time of service for the previous customer (or, in the case of the first customer, the time the depot was left);

2. `SERVICE TIME` at the previous location;

3. The maximum distance (calculated as ceiling) that can be traveled between the previous and current locations. Euclidean distance is used to calculate the distance, $D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, where D is the Euclidean distance, and (x_1, y_1) and (x_2, y_2) are the Cartesian coordinates of the two points.

If the current customer's `READY TIME` is less than that value, the current customer's service will start at `READY TIME`.

The main goal is to reduce the number of vehicles needed to service every customer, with the secondary goal being to reduce the total distance traveled on all routes. The Euclidean distance is used to determine the distance between two customers. Fewer vehicles are used in the solutions, which are thought to be better than solutions with a smaller overall distance. Only when there are equal numbers of vehicles in the solutions, distance is taken into account for comparison.

Our algorithm's `objective function` is to determine how many routes are required to serve every consumer. The solution with a smaller total distance is preferable when two solutions employ the same number of routes.

We employed geometric decrement: $T = \alpha * T$ and very slow temperature reduction: $T = \frac{T}{(1+\beta*T)}$ to reduce the temperature in each iteration.

When the temperature is low enough (e.g. 0.01 in the stopping criteria), the algorithm stops.

5.3.2 Simulated Annealing (SA) Algorithm

Metropolis et al. [152] were the ones that invented SA. Its many uses are attributed to Kirkpatrick et al. [196]. The concept known as "statistical mechanics" served as the foundation for SA.

The physical annealing process in metallurgy and the SA approach are comparable [196] [30] [152]. Solid metals are melted and then carefully cooled to create flawless crystal formations free of imperfections. Atoms can move randomly between states of higher energy when heated, releasing themselves from their original positions that match the local optimum of minimal energy. In this comparison, the solid's internal energy is represented by the evaluation function, and potential solutions are represented by the various energy levels [221]. To maximize the likelihood of obtaining a configuration with the least amount of internal energy (global optimum), cooling must be done slowly.

We must choose the solution representation for the vehicle routes before using the established approach to solve the VRPTW. The delivery depot is represented by 0 in the solution representation. According to the number in the solution representation, the first vehicle must leave the depot and visit every customer in sequence. Every vehicle that arrives at a customer must do so within the time limit that each customer is assigned. Even if the vehicle arrives before the time window opens, it must wait until the delivery window opens to avoid going beyond the time window constraint. One customer can only be served by one vehicle, and the vehicle must return to the depot within the depot's time window. Other vehicles then depart in line, and the procedure is repeated until every customer is routed.

To solve the VRPTW using simulation, three primary procedures are needed, which are as follows:

1. Generating a feasible random initial solution.
2. Employing special neighbor operators to generate a new solution.
3. Using a function to reduce the probability of adopting a new worse solution that is useful in escaping local optima during the search for the global optima.

Starting with a random feasible initial solution produced using a greedy algorithm known as the greedy randomized adaptive search procedure (GRASP), which is detailed in Section 5.3.3, simulated annealing (Algorithm 19) proceeds. Every time the main loop function is executed, $sa\_algorithm(s)$ as a current solution, produces a new solution, s'.

The local optimum is represented by solution s'', which is produced in the next step by applying the local search process [223]. If solution s'' produces a better result than solution s, then solution s'' is automatically accepted as the starting point for the next iteration. If not, the Metropolis condition [152], a probability function, is the acceptance criteria for solution s'':

$$\rho_t = e^{-\frac{f(s'')-f(s)}{T}} \tag{5.3.1}$$

The acceptance probability of solution s'' is dependent on temperature T. Accepting the worse solution is less probable as the process progresses and the temperature decreases. This determines the number of algorithm iterations and is referred to as the cooling schedule. A geometric function $T = \alpha * T$, where $0 \leq \alpha \leq 1$, represents the cooling schedule. For every problem set, the maximal (initial) and minimal temperatures are found empirically. The following formula determines the objective function that measures the quality of the provided solution:

$$f(s) = v_s * d_s \tag{5.3.2}$$

where d_s is the total route distance and v_s is the number of vehicles employed in the solution. If the evaluation function is improved, the $best\ solution$ is updated 5.3.2. The algorithm is forced to accept solutions with fewer used vehicles (routes) since the first factor of this function significantly raises the function's value. If the number of used vehicles is kept constant, the second factor is required to lower the total distance. The algorithm is terminated by the function $stop\_criterion()$ when temperature T falls to the lowest permitted value. To repeat the cooling process from the best solution discovered at that time, the temperature is also reset to its initial value a certain number of times.

The technique is built on the premise that if we set T high enough, we give more area for exploring the neighbouring solutions. The system then accepts more and more only improving movements as we approach the global minimum. The algorithm operates as a hill climb search (HC) if T is extremely low.

Algorithm 19 Simulated Annealing algorithm

procedure AlgorithmSimulatedAnnealing
  Let $t \leftarrow tempo0$
  Generate random initial solution s as current solution
  $bestSolution \leftarrow s$
  while until stop criterion met **do**
    current solution s produces a new solution s'
    Let $d_1 =$ value of the objective function (distance) of s
    Produce s'' (neighbor of s) as a new solution
    Let $d_2 =$ value of the objective function (distance) of s''
    if $d_2 \leq d_1$ **then**
      accept change $s \leftarrow s''$
      else
      accept change with probability $\rho_t = e^{-\frac{f(s'')-f(s)}{T}}$
      accept change $s \leftarrow s''$
      end if
      if $(f(s) < f(bestSolution))$ **then**
        $bestSolution \leftarrow s$
        end if
        $T \leftarrow T.\alpha * T\ cooling\ factor$
        end while
        return $bestSolution$

However, manually setting up and pruning the parameters every time when attempting to solve real situations is very challenging and ineffective. A random inner iteration loop is also used, and it attempts to improve the solution for a predetermined number of iterations at the same temperature level.

5.3.3 Greedy Randomised Adaptive Search Procedure (GRASP) Algorithm

Greedy heuristics, randomization, and local search are combined in the GRASP iterative solution framework [220]. A solution is constructed during the GRASP building phase, and its neighborhood is examined during the improvement phase to achieve a local minimum concerning an evaluation function. Upon termination, the best-found solution is provided back. Since greedy randomized constructs introduce a degree of $diversification$ into the search process, an efficient approach optimized for the $intensification$ of local search could comprise the GRASP improvement phase.

5.3.3.1 GRASP Constructive Mechanism

Randomization and a dynamic constructive heuristic define the GRASP constructive mechanism. First, a new element is added to the partially completed solution iteratively in order to develop the solution. Regarding an adaptive greedy function, all items are arranged in a list known as a restricted candidate list (RCL), which is made up of the $lambda$ high-quality entries. Picking one at random from the list—not necessarily the top of the list—is what defines the probabilistic component. Therefore, the degree of randomness and greediness in the creation process is determined and controlled by the length of the candidate list.

The primary elements that compose a GRASP minimization technique are shown in Algorithm 20, where $seed$ is the initial seed for the pseudo-random number generator and $maxIterations$ iterations are performed out.

The suggested GRASP construction mechanism makes use of a penalty-based greedy function that combines a set of criteria in a weighted manner with a parallel insertion solution construction technique. However, as different settings dominate solution quality and fine-tuning is a must, a decisive factor that considerably affects the quality of the solutions produced is the value of the weight parameters linked with the greedy function. Generally speaking, even though these parameters are closely related to one another and greatly rely on the specifics of the problem, several authors point out

Algorithm 20 Greedy Randomised Adaptive Search Procedure (GRASP) algorithm

```
1: procedure AlgorithmGreedyRandomisedAdaptiveSearchProcedure(maxIterations, seed)
2:     Read_Input()
3:     for <k = 1,..., maxIterations> do
4:         solution ←greedyRandomizedConstruction(seed)
5:         if solution is not feasible then
6:             solution ← repair(solution)
7:             end
8:             solution ← localSearch(solution)
9:             updateSolution(solution, bestSolution)
10:            end
11:            return bestSolution
```

that it is difficult to find a clear correlation between them or to determine a robust value for each of them, which would yield positive results for every test problem [24]. Algorithm 21 depicts the construction phase.

Algorithm 21 Greedy Randomized Construction algorithm

```
1: procedure AlgorithmGreedyRandomizedConstruction(seed)
2:     solution ← ∅
3:     Initialize the set of candidate elements
4:     Evaluate the incremental costs of the candidate elements
5:     while there exists at least one candidate element do
6:         Build the restricted candidate list (RCL)
7:         Select an element s from the RCL at random
8:         solution ← solution ∪ {s}
9:         Update the set of candidate elements
10:        Reevaluate the incremental costs
11:        end
12:        return solution
```

The majority of multi-start metaheuristics found in literature operate on a set of predefined initial solutions. Because local search is inherently difficult, it could be better to create multiple workable solutions and then use local search just on the most promising ones. In order to identify the parameter values that yield the best results, the typical approach is to first identify particular parameter value ranges and then modify input values in small increment units within these ranges [161]. In our implementation, a collection of well-performing parameter settings is first identified, and only the best-performing parameter settings are used at each GRASP iteration, as opposed to constructing a set of initial solutions.

Algorithm 22 Local Search (LS) algorithm

```
1: procedure AlgorithmLocalSearch(solution)
2:     while solution is not locally optimal do
3:         Find s′ ∈ N(solution) with f(s′) < f(solution)
4:         solution ← s′
5:         end
6:         return solution
```

Algorithm 22 is a basic local search algorithm that uses a neighborhood N and the $solution$ created in the first phase (perhaps made possible by the repair heuristic).

At last, an effort is made to reduce the amount of routes. In most current VRPTW applications [195] [161] [24], the idea of applying distinct strategies for decreasing both the number of routes and the distance traveled is implemented. Therefore, the route elimination process indicated in [24] is used within the proposed solution methodology before the GRASP improvement phase. The latter relies on an intelligent reordering process known as IR-insert, which enhances the

Ejection Chains (EC) heuristic. The main idea is to reorder or remove other customers from the same route in order to provide room for relocation by combining a number of movements into a compound move.

The building of high-quality first solutions for the local search is a crucial component of the construction process. Typically, simple neighborhoods are utilized. One of two strategies can be used to implement the neighborhood search: $best - improving$ or $first - improving\ strategy$. When using the $best - improving\ strategy$, the best neighbor takes the place of the existing solution once all neighbors are examined. The current solution goes to the first neighbor whose cost function value is less than that of the current solution in the case of a $first - improving\ strategy$. In actuality, we found that in numerous applications, both procedures frequently result in the same solution, but when the $first - improving$ strategy is applied, calculation times are reduced. We also noticed that using a $best - improving$ technique increases the probability of premature convergence to a poor local minimum.

5.3.3.2 Greedy Function

The parallel construction framework and penalty measurements presented in [72] are adapted by the proposed greedy function, which is further improved with extra mechanisms and customer selection criteria. Solomon's sequential insertion framework [211] states that at each iteration, a non-routed consumer is inserted into an existing partially created route to develop a workable solution. Solomon's sequential insertion is used in the context of parallel construction methods, taking numerous routes into consideration at once. In particular, a customer is iteratively assigned between two adjusted customers in a current partial route after initializing a set of r routes. An unassigned "seed" customer is detected and a new route is initiated if, during an iteration, an unassigned customer cannot be incorporated into any of the current set of routes. Until every customer is allotted a vehicle, the entire process is repeated.

The selection and assignment of "seed" customers is crucial to producing the initial batch of vehicles as well as any additional "surplus" vehicles that may be needed. The tight correlation between customers' time availability for service—introduced by time windows that specify the order in which customers are served by vehicles—complicates matters even further. On the other side, the task becomes more challenging the more away customers are from the depot. A two-phase "seed" customer selection approach is used to capture both of these characteristics. According to a proposal in [72], "seed" customers for the first set of routes are chosen based on how geographically or temporally distributed they are. If initializing more vehicles is required in later stages of building, random selection is used.

Let, the insertion cost of an unassigned customer u when inserted between i and j in a partial solution Ω is denoted as $\pi_{ij,u}$. The minimal insertion cost $\pi_{\rho,u} = min_{i,j\in\rho}\pi_{ij,u}$ is computed for each possible position for insertion of u into a route ρ. In a similar way the best possible insertion position at route $\rho*$ of u is indicated by the overall minimum insertion cost $\pi_{\rho*,u}$, which equates to the $min_{\rho\in\Omega}\pi_{\rho,u}$. After that, Π_u, the penalty cost, is computed for each unassigned client. This penalty represents the amount that would need to be paid in the future if the related customer is not placed in the best position available at that time.

$$\Pi_u = \sum_{\rho\in\Omega}(\pi_{\rho,u} - \pi_{\rho*,u}) \tag{5.3.3}$$

Big Π_u values suggest that u should be taken into account initially to prevent having to pay an excessive amount later on. Customers with low penalty levels, however, can wait for insertion. Consequently, to force Π_u to big values, the insertion cost for customers who cannot be realistically added to a route must be set to a large number lv. lv was set to infinity in [72]. As an alternative, we suggest an intuitively intelligent method that tunes lv values adaptively. More specifically, for the existing set of routes, lv is set equal to the difference between the total maximum and minimum insertion cost of all unassigned customers u. Upon the inability to insert a customer u onto a route ρ, the existing penalty Π_u is increased by $max_{u\in V}\{max_{\rho\in\Omega}\{\pi_{\rho,u}\}\}$ - $min_{u\in V}\{min_{\rho\in\Omega}\{\pi_{\rho,u}\}\}$ at each instance. Lastly, a weighted combined result from many sub-metrics defines cost $\pi_{ij,u}$.

$$\pi_{ij,u} = \vartheta_1\pi^1_{ij,u} + \vartheta_2\pi^2_{ij,u} + \vartheta_3(\pi^{3s}_{ij,u} + \pi^{3g}_{ij,u}) \tag{5.3.4}$$

Where $\vartheta_1 + \vartheta_2 + \vartheta_3 = 1$ is the result of nonnegative weights ϑ_1, ϑ_2, and ϑ_3. The distance increase that occurs by the insertion of u is measured by component $\pi^1_{ij,u}$ [211].

$$\pi^1_{ij,u} = t_{iu} + t_{uj} - t_{ij} \tag{5.3.5}$$

Vehicle utilization is measured by component $\pi^2_{ij,u}$ in terms of total waiting time before and after u is inserted [24].

$$\pi^2_{ij,u} = \sum_{i \in \rho \cup \{u\}} (a_i - \omega_{ik})^+ - \sum_{i \in \rho \cap \{u\}} (a_i - \omega_{ik})^+ \tag{5.3.6}$$

Lastly, two metrics, $\pi^{3s}_{ij,u}$ and $\pi^{3g}_{ij,u}$, are combined in the third component. When comparing the vehicle's k arrival time ω_{uk} at u to the closest possible time that service can occur a_u, the first measure is used [71].

$$\pi^{3s}_{ij,u} = \omega_{uk} - a_u \tag{5.3.7}$$

The second metric indicates if the chosen customer u's time window and the particular insertion point in the present route are compatible [71].

$$\pi^{3g}_{ij,u} = b_u - (\omega_{ik} + s_i + t_{iu}) \tag{5.3.8}$$

Refer to Chapter A, which represents the components and realistic implementations of the SA algorithm in programming format.

5.4 Ant Colony Optimization (ACO) Methods

The traveling salesman problem (TSP) was addressed by the original ant colony system (ACS) (Gambardella and Dorigo [123] [130] [129]). To solve a VRPTW where the number of vehicles and the trip time must be decreased, MACS-VRPTW has been proposed. Such multiple-objective minimization is accomplished through the use of two ACS-based artificial ant colonies. The challenge of determining the shortest closed tour which stops at every city in a given collection is known as the TSP.

The application of ACS to the TSP involves assigning two metrics to every arc on the TSP graph: the pheromone trail τ_{ij} and the proximity (closeness) η_{ij}. While the pheromone trail is constantly altered by ants at runtime, closeness, which can be stated as the inverse of the arc length, serves as a static heuristic value that never changes for a given problem instance. As a result, the management of pheromone trails—which are combined with the objective function to create new solutions—is the most crucial aspect of ACS. Pheromone levels, to put it loosely, indicate how desirable it is to include a certain arc in a solution. The purpose of pheromone trails is $exploration$ and $exploitation$. The probabilistic selection of the parts that make up a solution is called $exploration$; components with a strong pheromone trail are assigned a greater likelihood. The component that maximizes a combination of pheromone trail values and heuristic assessments is selected via $exploitation$.

5.4.1 Implementation of Ant Colony Systems (ACS)

5.4.1.1 ACS Tour Construction

The shortest tour is what ACS wants to find. In ACS, where m is a parameter, m ants construct tours in parallel. Every ant is given a beginning node at random and is required to construct a solution—a comprehensive tour. One ant iteratively adds new nodes up until all nodes are being visited, constructing a tour node by node. Ant k selects the next node j probabilistically from the set of feasible nodes N^k_i (for instance, the set of nodes that need to be visited) after finding itself in node i.

The tour is constructed using the following probabilistic rule: with probability q_0, the node j is selected with a probability p_{ij} proportional to $\tau_{ij} \cdot [\eta_{ij}]^\beta$, $j \in N^k_i$ ($exploration$, Equation 5.4.1). Similarly, with probability $(1 - q_0)$, the node j is selected with a probability p_{ij} proportional to $\tau_{ij} \cdot [\eta_{ij}]^\beta$, $j \in N^k_i$ ($exploration$) [121].

$$p_{ij} = \begin{cases} \frac{\tau_{ij} \cdot [\eta_{ij}]^\beta}{\sum_{l \in n^k_i} \tau_{il} \cdot [\eta_{il}]^\beta} & \text{if } j \in N^k_i \\ 0 & \text{otherwise} \end{cases} \tag{5.4.1}$$

β and q_0 are the parameters whereas q_0 $(0 \leq q_0 \leq 1)$ defines the relative importance of $exploitation$ versus $exploration$, and β weighs the relative importance of the heuristic value. When q_0 is smaller, the probability of using the probabilistic rule represented by Equation 5.4.1 is higher [121].

After every ant has constructed a complete solution, a local search process is used to improvise it. The pheromone trails are then updated using the optimal solution discovered at the start of the experiment. After that, m ants are restarted to repeat the procedure until a stopping criterion is satisfied. When a predetermined number of solutions have been generated, a predetermined amount of CPU time has passed, or no improvement has been made after a predetermined number of iterations, ACS ends.

5.4.1.2 ACS Pheromone Trails Update

Both locally and globally, the pheromone trail is updated in ACS. While global updating happens at the conclusion of the constructive phase, local updating happens during the building of solutions. In summary, local updating has the effect of dynamically altering the desirability of edges: each time an ant utilizes an edge, the amount of pheromone linked with the edge decreases, making the edge less appealing. Conversely, global updating is employed to enhance the search within the vicinity of the computed optimal solution.

In ACS, the pheromone trail is modified globally only when the optimal solution is applied. The updating strategy employed in the Ant System (AS) (Dorigo et al. [127] [128]), in which all constructed solutions are used to update the pheromone trails, is less effective than this one (Gambardella and Dorigo [122] [123] [130] [129]). The reasoning for this is that a "preferred route" is thereby encoded into the pheromone trail matrix, and ants will utilize this knowledge in the future to create new routes in the vicinity of this preferred route. The following is an update of τ_{ij}:

$$\tau_{ij} = (1-p) \cdot \tau_{ij} + p/J_{\psi}^{gb} \qquad \forall (i,j) \in \psi^{gb} \tag{5.4.2}$$

Where J_{ψ}^{gb} is the length of ψ^{gb}, the shortest path created by ants since the computations start, and p $(0 \leq p \leq 1)$ is a parameter. Every cycle, that is, when the constructive phase has been finished, this global updating mechanism is applied.

Local updating is carried out as follows: the quantity of pheromone trail on arc (i,j) is reduced in accordance with the following rule when an ant travels from node i to node j:

$$\tau_{ij} = (1-p) \cdot \tau_{ij} + p \cdot \tau_0 \tag{5.4.3}$$

Wherein the initial value of trails is τ_0. It was discovered that an acceptable value for this parameter is $\tau_0 = \frac{1}{n \cdot J_{\psi}^{h}}$, where n is the number of nodes and J_{ψ}^{h} is the length of the initial solution generated by the nearest neighbor heuristic (Flood [60]).

An intriguing feature of the local updating is that, even as ants visit edges, Equation 5.4.3 causes their path to narrow, making the edges less and less appealing and, as a result, encouraging the discovery of unexplored edges and diversity in the creation of solutions.

5.4.2 MACS-VRPTW (Multiple Ant Colony Systems for VRPTW)

Bullnheimer et al. [18] [19] built the first ant system for vehicle routing, even though VRPs are comparatively simple extensions of the TSP. They took into consideration the problem's most basic version, the capacitated vehicle routing problem (CVRP). But VRPTW is covered in this section. It has two objective functions: (i) minimizing the number of tours (or vehicles) and (ii) minimizing the overall trip time, to minimize the number of tours taking precedence over minimizing the travel time.

The MACS-VRPTW algorithms are as follows:

Procedure MACS-VRPTW ()

Initialization:

$\psi^{gb} \leftarrow$ feasible initial solution unlimited number of vehicles produced with a nearest neighbor heuristic

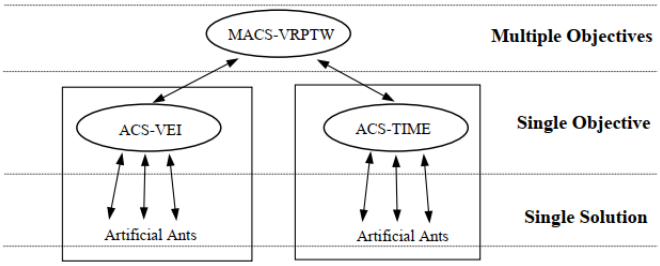


Figure 5.3: Multiple Ant Colony System for VRPTW architecture [121].

#active_vehicles(ψ) calculates the number of active vehicles in the feasible solution ψ

Main loop:

repeat

v $\leftarrow$ #active_vehicles (ψ^{gb})

Activate ACS-VEI ($v-1$)

Activate ACS-TIME (v)

while ACS-VEI and ACS-TIME are active

wait for an improved solution ψ from ACS-VEI or ACS-TIME

$\psi^{gb} \leftarrow \psi$

if #active_vehicles (ψ^{gb}) $<$ v

kill ACS-TIME and ACS-VEI

end while

until a stopping criterion is met

Two ACS-based colonies work together to coordinate their actions to simultaneously optimize both objectives in the MACS-VRPTW algorithm. While ACS-TIME enhances the feasible solutions identified by ACS-VEI, the initial colony, ACS-VEI, aims to reduce the number of vehicles used. While using separate pheromone trails, the two colonies work together by sharing the ψ^{gb} variable, which is controlled by MACS-VRPTW. Using the nearest neighbor heuristic, a feasible VRPTW solution is initially discovered to be ψ^{gb}. Then, the two colonies improve ψ^{gb}. Upon activation, ACS-VEI searches for a feasible solution using one fewer vehicle than the total number of vehicles in ψ^{gb}. Optimizing the total trip time of solutions with the same number of vehicles as those used in ψ^{gb} is the aim of ACS-TIME. Every time one of the colonies calculates an improved viable solution, ψ^{gb} is updated. If there are fewer vehicles in the improved solution than there were in ψ^{gb}, then MACS-VRPTW eliminates ACS-TIME and ACS-VEI. After that, the procedure is repeated and two new colonies are established using the new, smaller fleet of vehicles [121].

5.4.2.1 ACS-TIME and ACS-VEI colonies

This article describes the ACS-VEI and ACS-TIME colonies' operational principles.

The ACS-TIME: travel time minimization algorithms are as follows:

Procedure ACS-TIME(v)

Initialization:

initialize pheromone and data structures using v (//where v is the minimum number of vehicles for which a feasible solution has been determined)

For loop:

repeat

for each ant k

new_active_ant(k, $local\_search = TRUE$, 0) (//create a solution ψ^k)

end for each

if $\exists\ k : \psi^k$ is feasible and $J_\psi^k < J_\psi^{gb}$ (//update the best solution if it is improved)

send ψ^k to MACS-VRPTW

$\tau_{ij} = (1-p) \cdot \tau_{ij} + p/J_\psi^{gb} \qquad \forall (i,j) \in \psi^{gb}$ (//perform global updating according to Equation 5.4.2)

end while

until a stopping criterion is met

The objective of the conventional ACS-TIME colony is to calculate a tour in the shortest amount of time. Within ACS-TIME In order to create problem solutions $\psi^1, \ldots, \psi^m$, m artificial ants are triggered. Every solution is constructed by invoking the new_active_ant procedure, a constructive procedure akin to the ACS constructive procedure created for the TSP and detailed in Section 5.4.2.3. Following computation, $\psi^1, \ldots, \psi^m$ are compared to ψ^{gb}; if one solution proves to be better, it is forwarded to MACS-VRPTW. This solution is used by MACS-VRPTW to update ψ^{gb}. Equation 5.4.2 and ψ^{gb} are used to carry out the global updates following the production of solutions [121].

The ACS-VEI colony maximizes the number of consumers it visits in an attempt to find a feasible solution. Using $v-1$ vehicles—that is, one fewer vehicle than the smallest number of vehicles for which a workable solution has been determined—ACS-VEI begins its computation (the number of vehicles in ψ^{gb}). The colony generates unworkable solutions during this search, which some clients choose not to visit. The variable $\psi^{ACS-VEI}$ in ACS-VEI stores the solution generated since the start of the trial with the largest number of visited customers. A solution is better compared to $\psi^{ACS-VEI}$ only when there is an increase in the number of customers visited. As a result, ACS-VEI differs from the conventional ACS used with the TSP. The solution with the greatest number of visited customers in ACS-VEI is the current best solution $\psi^{ACS-VEI}$, which is typically not feasible; in contrast, when ACS is applied to TSP, the current best solution is the shortest one [121].

The ACS-VEI: number of vehicles minimization algorithms are as follows:

Procedure ACS-VEI(s)

Initialization:

initialize pheromone and data structures using s

$\psi^{ACS-VEI} \leftarrow$ initial solution with s vehicles produced with a nearest neighbor heuristic (//$\psi^{ACS-VEI}$ is not necessary feasible)

(//Parameter s is set to $v-1$, which is, one vehicle less than the smallest number of vehicles for which a feasible solution has been calculated)

(//#visited_customers(ψ) calculates the number of customers that have been visited in solution ψ)

For loop:

repeat

for each ant k

new_active_ant(k, $local\_search = FALSE$, IN) (//create a solution ψ^k)

$\forall$ customer $j \notin \psi^k : IN_j \leftarrow IN_j + 1$

end for each

if $\exists\ k$: #visited_customers(ψ^k) > #visited_customers($\psi^{ACS-VEI}$) then (//update the best solution if it is improved)

$\psi^{ACS-VEI} \leftarrow \psi^k$

$\forall\ j : IN_j \leftarrow 0$ ($//restIN$)

if $\psi^{ACS-VEI}$ is feasible then

send $\psi^{ACS-VEI}$ to MACS-VRPTW

(//perform global updating according to Equation 5.4.2 both for $\psi^{ACS-VEI}$ and ψ^{gb})

$\tau_{ij} = (1-p) \cdot \tau_{ij} + p/J_{\psi}^{ACS-VEI} \qquad \forall (i,j) \in \psi^{ACS-VEI}$

$\tau_{ij} = (1-p) \cdot \tau_{ij} + p/J_{\psi}^{gb} \qquad \forall (i,j) \in \psi^{gb}$

until a stopping criterion is met

A vector IN of integers is managed by ACS-VEI in order to optimize the number of customers serviced. The number of times client j hasn't been inserted into a solution is stored in the entry IN_j. The constructive procedure new_active_ant uses IN to provide preference to the customers who are not as often featured in the solutions. Pheromone trails in ACS-VEI are globally updated after each cycle with two distinct solutions: $\psi^{ACS-VEI}$, the feasible solution with the fewest vehicles and the shortest travel time, and ψ^{gb}, the infeasible solution with the greatest number of visited customers [121].

It has been demonstrated through numerical tests that a double update significantly boosts system performance. In fact, the trails toward customers not included in the solution are not getting longer with the updates with $\psi^{ACS-VEI}$. The updates with ψ^{gb} are increasing trails toward all consumers since ψ^{gb} is viable.

5.4.2.2 Solution Model

Each ant constructs a single tour in the MACS-VRPTW solution model (Figure 5.4). The following is an example of a solution: Initially, the depot is duplicated several times, equivalent to the number of available vehicles, with all of its connections to and from the clients. The depot's copies are spaced apart by zero. With this method, VRP problem becomes more similar to the conventional TSP problem.

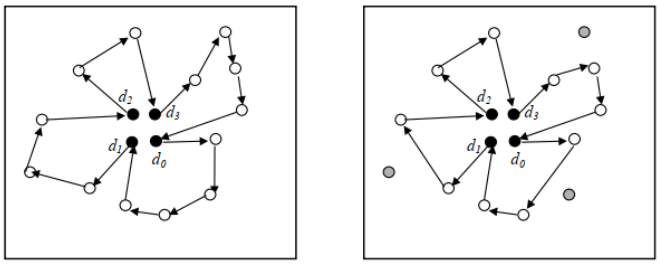


Figure 5.4: An example of feasible (left) and infeasible (right) solutions, four duplicate depots and four active vehicles make up the VRP [121].

Therefore, a path that visits every node exactly once is a possible solution in both this model and the TSP. A single tour representing a solution to the VRP problem is shown in Figure 5.4. Clients are white points, and duplicate depots $(d_0, \ldots, d_3)$ are black points. The duplicate depots have been divided to improve the image, however, they all have the same locations. One benefit of this type of solution representation is that, because of the pheromone update rules, the trails heading toward the multiple depots are not as appealing as they would be if there was only one depot. This has a favorable impact on the caliber of solutions generated by the constructive process [121].

5.4.2.3 Solution Constructive Procedure

The new_active_ant: constructive procedure for ant k used by ACS-VEI and ACS-TIME steps.

Procedure $new\_active\_ant(k, local\_search, IN)$

Initialization:

put ant k in a randomly selected duplicated depot i

$\psi^k \leftarrow < i >$

$current\_time_k \leftarrow 0$, $load_k \leftarrow 0$

For loop: (//This is where ant k constructs its tour. The tour is kept in ψ^k.)

(//The set N_i^k of feasible nodes, which includes all nodes j that have yet to be visited and are acceptable with time windows $[b_j, e_j]$ and delivery quantity q_j of customer j, is computed starting from node i. $\forall\ j \in N_i^k$ compute the attractiveness n_{ij} as follows:)

$delivery\_time_j \leftarrow max(current\_time_k + t_{ij}, b_j)$

$delta\_time\_ij \leftarrow delivery\_time_j - current\_time_k$

$distance_{ij} \leftarrow delta\_time_{ij} * (e_j - current\_time_k)$

$distance_{ij} \leftarrow max(1.0, (distance_{ij} - IN_j))$

$n_{ij} \leftarrow 1.0/distance_{ij}$

Choose probabilistically the next node j using n_{ij} in $exploitation$ and $exploration$ (Equation 5.4.1) mechanisms

$\psi^k \leftarrow \psi^k + < 1 >$

$current\_time_k \leftarrow delivery\_time_j$

$load_k \leftarrow load_k + q_j$

If j is a depot then $current\_time_k \leftarrow 0$, $load_k \leftarrow 0$

$\tau_{ij} = (1 - p) \cdot \tau_{ij} + p \cdot \tau_0$ (//local pheromone updating (Equation 5.4.3))

$i \leftarrow j$ (//new node for ant k)

Until $N_i^k = \{\}$ (//no more feasible nodes are available)

(//This step extends the ψ^k route by speculatively adding non-visited clients.)

$\psi^k \leftarrow$ insertion_procedure(ψ^k)

(//In this step, a local search process optimizes possible routes. In ACS-TIME, the parameter $local\_search$ is TRUE; in ACS-VEI, it is FALSE)

if $local\_search = TRUE$ and ψ^k is feasible then

$\psi^k \leftarrow$ local_search_procedure(ψ^k)

The ACS constructive approach created for the TSP is comparable to this one: Starting from a randomly selected depot copy, each artificial ant travels, at each step, to a node that hasn't been visited yet and doesn't go against vehicle capacity and time frame restrictions. If the ant is not found in a duplicated depot, the set of available nodes also includes duplicated depots that have not yet been visited.

An ant situated on node i uses the $exploration$ and $exploitation$ mechanisms to probabilistically select the next node j to visit. The traveling time t_{ij} between nodes i and j, the time window $[b_j, e_j]$ associated with node j, and the number of times IN_j node j has not been placed in a problem solution are all taken into account when computing the attractiveness n_{ij}. The variables IN are not used when the new_active_ant is called by ACS-TIME, and the associated parameter is set to zero [121].

Equation 5.4.3 states that a local update of the pheromone trail is carried out each time an ant goes from one node to another. Lastly, the solution may still be unfinished after the constructive phase (some customers may have been left out), in which case it is tentatively finished by making further insertions. All non-visited clients are sorted by lowering delivery quantities to complete the insertion. Until no more feasible insertions are possible, the best feasible insertion (shortest travel time) is found for each client.

Furthermore, ACS-TIME conducts a local search process to enhance the caliber of the feasible solutions. Moves used in the local search are akin to CROSS exchanges (Taillard et al. [56]). The interchange of two client sub-chains is the basis of this process. Eventually, one of these sub-chains might be vacant, in which case a more traditional customer insertion would take place.

Refer to Chapter A, which represents the components and realistic implementations of the ACO algorithm in programming format.

5.5 GLS (OR-Tools) Methods

Since 2008, Google has been developing OR-Tools, a general-purpose optimization toolkit that it made publicly available in 2015 [81]. Several solvers, both first- and third-party, can be accessed uniformly with this toolbox. Specifically, it provides an elevated interface for vehicle-routing problems (VRPs). Multiple solvers are included in OR-Tools, including two CP solvers (CP*, since the initial open-source release, 2009) and CP-SAT (since 2009). Additionally, there are two linear solvers which are built for generic modelling, (access to MIP solvers) PDLP [40], a first-order large-scale linear solver, and the simplex-based Glop, which has been around since 2014 [173].

Large-scale industrial vehicle routing problems with complex constraints, such as vehicle capacities with multiple starting/ending depots, client time windows taking into account traffic and driver breaks, pick-up-and-delivery priority rules, not compatible shipments throughout the same vehicle, solution similarities to a previous call to the solver, etc., are the main focus of the routing component, which has historically played a significant role in the development of the overall solver. In order to achieve this, even while the user gets access to the core constraint-programming model, a high-level modeling API is proposed to them in Python, C++, Java, and C#, utilizing just routing notions.

Three components make up the routing solution from an algorithmic perspective:

1. First-solution heuristics produce good candidate vehicle tours;
2. Local search refines the first solutions using metaheuristics to lead the search;
3. A CP engine validates the best solution's optimality or enhances it.

The emphasis on generality in the solver, including its algorithms, sets it apart from many academic solvers.

5.5.1 GLS (OR-Tools) VRPTW Modeling Concepts

The route modeling API is a high-level interface specifically designed to handle this class of problems and may be used to enter VRP-like models directly. Through a callback function (`RoutingModel:: RegisterTransitCallback`), the solver consumes the distance-weighted directed graph. A collection of optimization variables (`RoutingModel:: Start` and `RoutingModel:: Next`) describe vehicles. The solution uses `RoutingDimension` and `CumulVar` to describe quantities that accumulate along a path, such as time, distance, or vehicle load. Time windows are modeled using this approach.

Although arc-routing problems aren't supported out of the package, users can depict arcs that nodes should traverse in order to convert them to standard vehicle-routing problems [92]. The sole way edge routing differs from arc routing is that it only needs to traverse each edge once, in either direction. This difference can be represented using a disjunction (`RoutingModel:: AddDisjunction`).

A few modeling orthogonal concepts are provided by OR-Tools:

- Weighted graph, wherein one or more depots are included.
- Vehicles with no capacity.
- Visits with time windows.
- Dimensions: Callback for arc lengths, New dimension: time, Slack: waiting time, Variable bounds: time windows

Dimensions

The routing solver tracks quantities that build up along a vehicle's path, including the travel time or, if the vehicle is performing pickups and deliveries, the total weight it is carrying, using an object known as a *dimension*. We must define a dimension in order to indicate such a quantity if it appears in the constraints or the objective function of a routing problem.

A dimension is created in the VRPTW example to track the $cumulative\ travel\ time$ of each vehicle. The constraint that a vehicle can only visit a site during its time window is enforced by the solver using the dimension.

(In the CVRP example, the $cumulative\ load$ that the vehicle is carrying throughout the route is tracked by creating a dimension for the $demands$ (weights, volumes of items to be picked up, etc.). The constraint that a vehicle's load cannot exceed its capacity is enforced by the solver through the usage of dimension).

These are the inputs for the `AddDimension` method: `callback_index`, `slack_max`, `capacity`, `fix_start_cumulative_to_zero`, `dimension_name`, see details [81].

Slack Variables

This example shows how to use slack variables in a travel time problem. Assume that, in one step of its trip, a vehicle travels from place i to position j, and that [81]:

- At i, the $cumulative\ travel\ time$ of the vehicle is 100 minutes.
- At j, the $cumulative\ travel\ time$ of the vehicle is 200 minutes.
- 75 minutes is the $travel\ time$ between i and j.

When the vehicle arrives at place i, it cannot depart right away, or otherwise its $cumulative\ time$ at location j would be 175. Rather, the vehicle has to wait at place i for 25 minutes before leaving; that is, there are 25 minutes of $slack$ at site i.

A VRPTW must account for slack since time window restrictions may force vehicles to wait before arriving at a destination. Set `slack_max` to the longest duration you wish to permit vehicles to wait at a place before moving on to the next one in a situation similar to this one. To make it irrelevant how long individuals wait, simply set `slack_max` to a very large number.

(Conversely, with the CVRP, there is never any slack because the change in the $accumulated\ load$ from i to j always equals the $demand$ at i. We can adjust `slack_max` to 0 for issues such as these).

Two categories of internal variables involving quantities that accumulate along routes are stored within a dimension:

- $Transit\ variables$: The amount that changes at every step of a route. When a route consists of steps `i -> j`, the transit variable is either the callback value at location i (if the callback depends on only one location) or the i, j entry in the transit matrix (for a transit callback).
- $Cumulative\ variables$: The total amount that has built up (accumulated quantity) at every location. By using `dimension_name.CumulVar(i)`, we may get to the cumulative variable at i.

Given the assumption that a vehicle travels from point i to point j in a single step, the relationship between these variables and the slack is as follows [81]: `cumul(j) = cumul(i) + transit(i, j) + slack(i)`

Resource Constraints

Thus far, we have studied routing issues with constraints that hold when traveling by vehicle. We then introduce a VRPTW with depot-specific restrictions: every vehicle must be loaded before leaving and emptied upon returning to the depot. Two vehicles can only be loaded or unloaded simultaneously from the two loading docks that are available. Consequently, certain vehicles have to wait for other vehicles to be loaded, which causes a delay in their exit from the depot. The challenge is in determining the optimal vehicle routes for the VRPTW while simultaneously adhering to the depot's loading and unloading regulations [81].

Furthermore, the VRPTW data sets contain the following attributes: `time_matrix`, `time_windows`, `vehicle_load_time`, `vehicle_unload_time`, `depot_capacity` which are utilized during loading and unloading time windows and during depot resource constraints.

- Add Time Windows for Loading and Unloading: Different from the time windows at the locations, time windows generated by the method `FixedDurationIntervalVar` have $variable\ time\ windows$. `Vehicle_load_time` and `Vehicle_unload_time` determine the window sizes.

- Add Resource Constraints at the Depot: The maximum number of vehicles that can be loaded or unloaded simultaneously is known as `depot_capacity`. The relative quantities of space needed by each vehicle when loading or unloading are represented by the vector `depot_usage`.

Penalties and Dropping Visits

We describe how to address routing situations for which there is no feasible solution because of constraints. For instance, no solution is feasible if you are provided a VRP with capacity limits, meaning that the entire demand at all sites exceeds the total capacity of the vehicles. In these situations, vehicles have to skip stops at certain places. The challenge is in determining which visits to forfeit.

To address the issue, we impose new costs, or penalties, everywhere. The penalty gets added to the total distance traveled each time a visit to a destination is canceled. Next, the route that minimizes the overall distance as well as the total of the penalties for all dropped places is determined by the solver.

Assume, the VRP with capacity constraints is displayed in Figure 5.5, where the numbers corresponding to the three sites (apart from the depot) represent demands.

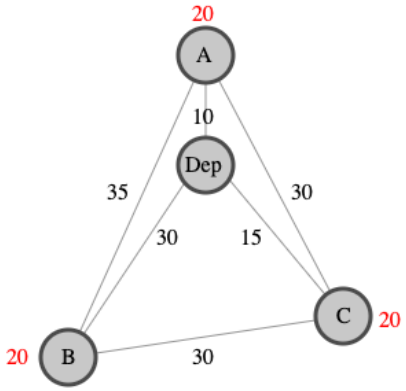


Figure 5.5: An example of penalty and dropping visits [81].

Let's say there is only a single vehicle with a capacity of 50. The demand is 60, so it cannot visit all three places, A, B, and C. You give each site a significant penalty, say 100, to fix the issue. When the solver determines that the issue is not possible, it abandons position B and takes the following route: $Depot- > A- > C- > Depot$. The shortest path, covering a distance of 55, stops at two of the three destinations [81].

Penalty Sizes

We selected penalties in the above-mentioned scenario that exceed the total of all distances between the places (apart from the depot). Consequently, the solver does not drop any more locations after dropping one to make the problem feasible because the penalty for performing so would be excessive to offset any additional travel distance decrease. This provides a good solution to the problem, assuming you wish to make as many deliveries as feasible.

If you can apply smaller penalties and hence don't need to execute as many deliveries as possible, the solver may drop more locations than needed to complete the problem. For instance, you may take this action if visiting particular areas has charges above and beyond the fundamental cost of travel [81].

Search Limits

Solving vehicle routing issues with numerous locations can be a time-consuming task. Setting a search limit, which ends the search after a predetermined amount of time or the number of solutions returned, is a smart concept for these kinds of situations. For example, time limits, solution limits, iteration limits etc.

Setting Initial Routes for a Search

Rather than letting the solver come up with an initial solution, we could consider providing a set of initial routes for a VRP for some issues. For instance, suppose we already know a good solution to a problem and wish to use it as an initial

basis for solving a modified problem. Follow these procedures to establish the initial routes:

- Define an array with the starting routes in it.
- Use the `ReadAssignmentFromRoutes` function to create the first solution.

5.5.2 Vehicle-Routing Solving Procedures

Three basic components (steps) are used in or-tools, along with their respective applications.

1. They begin with a solution (could be feasible or not).
2. Then they make this solution better locally.
3. When they come to a stopping criterion, they end the search, but typically they don't guarantee the quality of the solutions they find.

5.5.2.1 First-Solution Heuristics

We must begin with a solution to improve it locally. This solution needs to be feasible in or-tools. With a `DecisionBuilder` that we supply to the local search algorithm, we can either generate a first solution and send it to the solver, or we can let the solver find one for us.

What if coming up with a feasible solution is our challenge? Up till we can build an initial solution for that relaxed model, we $relax$ the constraints. Next, we add comparable terms to the objective function (e.g., in a Lagrangian relaxation [44]) to enforce the $relaxed\ constraints$.

The approach the solver takes to arrive at an initial solution is known as the first solution strategy. The alternatives (options to set the methods) for `first_solution_strategy` are listed in the following [81]:

- `AUTOMATIC` : Allows the solver to determine the best approach based on the model being solved.
- `PATH_CHEAPEST_ARC` : Route extension is achieved by iteratively adding nodes to the route, beginning with a route "start" node and connecting it to the node that generates the cheapest route segment.
- `PATH_MOST_CONSTRAINED_ARC` :Arcs are evaluated using a comparison-based selector that prioritizes the most confined arc first, much like `PATH_CHEAPEST_ARC` . Use the function `ArcIsMoreConstrainedThanArc()` to assign a selector to the routing model.
- `EVALUATOR_STRATEGY` : Like `PATH_CHEAPEST_ARC` , but with arc costs evaluated by function supplied to `SetFirstSolutionEvaluator()` instead of the function itself.
- `SAVINGS` : Savings algorithm (Clarke and Wright [64]).
- `SWEEP` : Sweep algorithm (Wren and Holliday [9]).
- `CHRISTOFIDES` : Christofides algorithm (a modified version of the Christofides algorithm that does not guarantee the 3/2 factor of the approximation on a metric traveling salesman; it uses maximal matching rather than maximum matching). extends a route until no more nodes may be added to it to work with vehicle routing models [36].
- `ALL_UNPERFORMED` : Turn every node off. Discovers a solution only in the case where nodes are optional and part of a disjunction constraint with a finite penalty cost.
- `BEST_INSERTION` : Insert the cheapest node at its cheapest position to iteratively create a solution; the insertion cost is determined by the routing model's global cost function. Only functions on models having optional nodes as of 2/2012 (with finite penalty costs).

- `PARALLEL_CHEAPEST_INSERTION` : Insert the cheapest node at its cheapest position iteratively to develop a solution; the insertion cost is determined by the arc cost function. Compares to `BEST_INSERTTION` in speed.
- `LOCAL_CHEAPEST_INSERTION` : Insert each node at the lowest possible location iteratively to construct a solution; the insertion cost is determined by the arc cost function. differs from `PARALLEL_CHEAPEST_INSERTION` in that the node chosen for insertion is taken into account according to its creation order. Compares to `PARALLEL_CHEAPEST_INSERT` in speed.
- `GLOBAL_CHEAPEST_ARC` : Connect the two nodes that result in the cheapest route segment iteratively.
- `LOCAL_CHEAPEST_ARC` : Choose the node that has the earliest unbound successor and link it to the node that generates the least expensive route segment.
- `FIRST_UNBOUND_MIN_VALUE` : Connect the first node that is available to the first node that has an unbound successor. This is the same as using `ASSIGN_MIN_VALUE` in conjunction with `CHOOSE_FIRST_UNBOUND` .

5.5.2.2 Local Search and Metaheuristics

$Local$ improvements are made to the initial solution. This implies that for a given solution, we must specify a $neighborhood$ and a method for exploring it, either explicitly or implicitly. Depending on how a neighborhood is defined, two solutions can be either very far apart or $near$ (i.e., they belong to the same neighborhood).

The objective is to discover a better solution in the neighborhood surrounding an initial solution, investigate it (partially or completely), and then repeat the process until a stopping criterion is satisfied.

Let us indicate the neighborhood of a solution x by N_x. At its most basic, local search (LS) could be expressed as follows [164]:

$x \leftarrow x_0$

While stop criteria not met, do

1. Find neighborhood N_x
2. Find "best" solution in N_x: x_{best}
3. $x \leftarrow x_{best}$

end while

Steps 1 and 2 are often completed at the same time. That's how it functions In or-tools. Figure 5.6 illustrating this procedure:

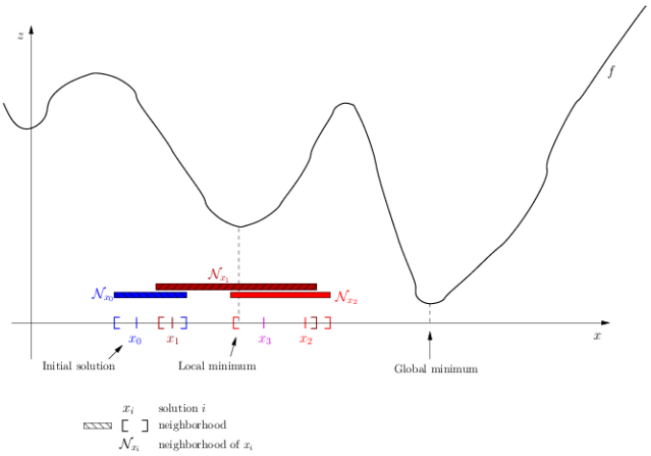


Figure 5.6: An idea of OR-Tools neighborhood local search process [164].

The function f to minimize is shown in Figure 5.6. Don't let its two-dimensionality deceive us. In a multidimensional space, solutions are represented by the $x - axis$. A 1-dimensional space containing the values of the objective function f is represented by the $z - axis$.

Starting with an initial feasible solution x_0, the Local Search process examines the neighborhood N_{x_0} of this solution. Found to be the "best" solution is x_1. The Local Search process is restarted, but this time, x_1 is the first solution. The best solution identified is x_2 in the neighborhood N_{x_1}. The process keeps going till the halting requirements are satisfied. Assume that x_3 is the result of the search and that one of these requirements is satisfied. It is evident that although the strategy approaches the local optimum, it fails to reach the global optimum [164].

The method's inability to locate a global optimum—aside from chance—justifies its name as local search. It is likely to identify a local optimum, or very near to one.

Our process would likely have looped around the $local\ optimum$ if we had carried on with the search. We refer to this situation as a local optimum trapping the local search algorithm. While several LS techniques, such as tabu search, are created to get out of such a local optimum, there is still no assurance that these techniques will be successful.

As an example, (according to Figure 5.6) it is evident that neighborhoods do not have to be the same size or revolve around a variable x_i. The relationship "being in the neighborhood of" is not always symmetric, as we can also observe: $x_1 \in N_{x_0}$, but $x_0 \notin N_{x_1}$. (Be aware that the majority of LS techniques consider this relation's symmetry to be a good attribute. In the event of a false start, we would be able to go back and investigate additional options if this relation is symmetric. Consider moving left on the Figure 5.6 to investigate the area beyond the $z - axis$ [164].)

By implementing the `MakeNextNeighbor()` callback method from a `LocalSearchOperator`, we can define a neighborhood in or-tools: Each time the solver calls this method internally, it creates a single neighborhood solution. Upon building a successful candidate, make `MakeNextNeighbor()` yields a true result. It returns false once every neighborhood has been checked.

Local Search Options

The local search methods (also known as metaheuristics) that we can use with OR-Tools are listed in the following list [81]:

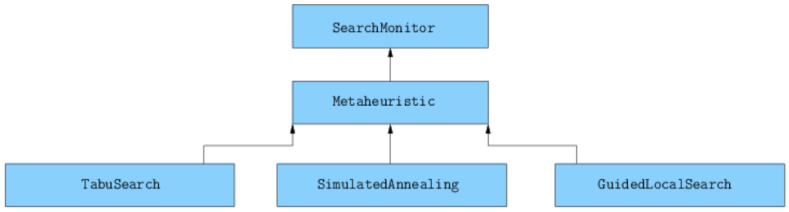


Figure 5.7: The idea of Metaheuristics in OR-Tools [164].

- `AUTOMATIC`: Let the metaheuristic be chosen by the solver.
- `GREEDY_DESCENT`: Until a local minimum is met, accepts optimizing (lowering the cost of) nearby local searches.
- `GUIDED_LOCAL_SEARCH`: To get out of a local minimum, use guided local search. For vehicle routing, this is typically the most effective metaheuristic [84].
- `SIMULATED_ANNEALING`: Breaks free from local minima by using simulated annealing [208].
- `TABU_SEARCH`: Breaks free from local minima by using tabu search [105].
- `GENERIC_TABU_SEARCH`: Utilizes tabu search on the solution's objective value to get out of local minima.

5.5.2.3 Constraint Satisfaction

Operators for local searches ignore constraints. Very few false positives and infeasible neighbors are rejected by local search technique's lightning-fast filters (read how to check VRP instance feasibility [33]).

When a problem can be characterized in terms of arbitrary restrictions, finding feasible solutions from a vast pool of possibilities is known as constraint optimization, or constraint programming (CP). Here, OR-Tools checks constraint fulfillment after the search using the CP engine such as CP* or CP-SAT and concluded the search [163].

Tree exploration can be used to check the feasibility, if desired, to demonstrate optimality [239].

Refer to Chapter A, which represents the components and realistic implementations of the GLS (OR-Tools) algorithm in programming format.

Chapter 6

Results

6.1 Computational Experiments

This study's established methodology makes use of the Python programming language, MacBook Pro (2014) computers with a 2.7 GHz Dual-Core Intel Core i5 processor, 8 GB of 1867 MHz DDR3 memory, and the macOS operating system.

The VRPTW benchmark problem instances supplied by Solomon [211] are utilized as instances, and the results computed from the method developed are compared with the outcomes of alternative ways to confirm the efficacy of the developed approach. Every problem in the benchmark problem instances is presumptively composed of 100 customers, one delivery depot, and vehicles with the same loading capacity. Every customer has a time window or the earliest and latest possible service time. There is a fixed payload capacity for every vehicle. It is possible to convert time and distance into equivalent units, and it is known how much each customer demands. Six sets comprise the VRPTW benchmark problem instances: C1, C2, R1, R2, RC1, and RC2.

Three data sets from each of the R1, R2, C1, C2, RC1, and RC2 will be tested. Each data set will be run three times, after which the model's parameters will be changed three times and the same data sets will be run again three more times (see Section 6.1.2 for the parameter tuning). The results will be tabulated in Section 6.1.3.

(**Note** - Using the **Python Jupyter Notebook** is highly advised. While compiling, each model executes independently. However, if you run the code straight from `main.py`, the application may crash and display errors due to the ACO procedure's use of multiple threads! However, it functions perfectly now that I'm using the Python Jupyter Notebook code.)

6.1.1 Solomon's VRPTW Benchmarking Data Sets

The VRPTW benchmarking data was first presented by Solomon in Solomon (1984) [212]. According to Christofides, Mingozzi, and Toth (1979) [151], the standard set of routing test problem data provides the foundation for creating the Solomon Benchmarking instances. Each of Solomon's 56 instances has 100 customers. The VRPTW Benchmark Problems by Solomon are categorized in Table 6.1.

Problem Set	Random	Clustered	Random & Clustered
Short Scheduling Horizon	R1-type	C1-type	RC1-type
Long Scheduling Horizon	R2-type	C2-type	RC2-type

Table 6.1: Solomon's VRPTW Benchmark Problems.

Remarks of Solomon's benchmark problems:

1. Solomon came up with six sets of problems. Several elements that influence the behavior of scheduling and routing

algorithms are highlighted in their design. They are as follows:

- Geographical data arrangement of customers;
- Number of customers served by the same vehicle;
- Percent of time-constrained customers;
- Tightness and positioning of time windows.

2. Geographical data are produced in two ways: randomly in problem sets R1 and R2, clustered in problem sets C1 and C2, and in problem sets by RC1 and RC2, a combination of randomly and clustered structures.
3. Problem sets R1, C1, and RC1 only permit a small number of consumers per route (about five to ten) and have a limited scheduling horizon. On the other hand, the sets R2, C2, and RC2 have a long scheduling horizon that allows the same vehicle to handle a large number of customers (more than 30).
4. For all problems within a single kind (R, C, and RC), the customer's coordinates are identically set.
5. Problems vary in terms of the temporal window widths. While some have barely any time constraints, others have extremely tight time constraints. We generated issues with 25, 50, 75, and 100% time windows in terms of time window density, or the proportion of customers having time windows.
6. Larger problems are 100 customers the Euclidean problems, where the distances and times of travel are equal. By taking into account just the first 25 or 50 customers, small difficulties have been produced for each of these larger ones.

6.1.2 Parameter Calibration

6.1.2.1 Hybrid Genetic Search (HGS) Parameter Tuning

`MaxIterations` and `MaxRuntime` are the two options that allow us to control the HGS algorithm's runtime. We are not required to use them simultaneously. We can choose to control the algorithm's execution by varying the number of iterations or the duration.

I'm going to use `MaxRuntime` in my experiment by setting three distinct times.

`MaxRuntime` = 60 then 120 then 300 (seconds)

(or `MaxIterations` = 1000 or 5000 or 10000)

These are the settings I have made for the Hybrid Genetic Search (HGS) algorithm:

For better Feasibility:

`capacity_penalty` = 200

`tw_penalty` = 200

`repair_probability` = 0.8 (for GA)

`nb_iter_no_improvement` = 20000 (for GA)

For Penalty:

`init_capacity_penalty` = 20

`init_time_warp_penalty` = 6

`repair_booster` = 12

`num_registrations_between_penalty_updates` = 50

`penalty_increase` = 1.34

`penalty_decrease` = 0.32

`target_feasible` = 0.43

For the Population for Genetic Algorithm:

`min_pop_size` = 25

`generation_size` = 40

`nb_elite` = 4

`nb_close` = 5

`lb_diversity` = 0.1

`ub_diversity` = 0.5

`overlap_tolerance` = 0.05 (for intensify)

For Neighbourhood Search:

`weight_wait_time` = 0.2

`weight_time_warp` = 1.0

`nb_granular` = 40

`symmetric_proximity` = True

`symmetric_neighbours` = False

6.1.2.2 Guided Local Search OR-Tools (GLS) Parameter Tuning

In the same manner as HGS managed. `MaxIterations` and `MaxRuntime` are the two options that allow us to control the GLS (OR-Tools) algorithm's runtime. We are not required to use them simultaneously. We can choose to control the algorithm's execution by varying the number of iterations or the duration.

I'm going to use `MaxRuntime` in my experiment by setting three distinct times.

`MaxRuntime` = 60 then 120 then 300 (seconds)

(or `MaxIterations` = 1000 or 5000 or 10000)

For `AddDimension`:

10 ** 10 (allow waiting time at nodes)

10 ** 10 (maximum time per vehicle route)

6.1.2.3 Ant Colony Optimization (ACO) Parameter Tuning

To experiment ACO algorithm, three runs are performed for each set of problem data, stopping the process by parameter `runtime_in_minutes`. We've conducted experiments using the following parameter settings:

`ants_num` = 10, then 20, then 30 (number of ants)

`q0` = 0.3, then 0.5, then 0.9 (as α)

`beta` = 0.3, then 0.5 then 0.9 (as β)

`rho` = 0.1, then 0.5 then 0.9 (as ρ)

`runtime_in_minutes` = 2, then 3, then 5 (minutes)

6.1.2.4 Simulated Annealing (SA) Parameter Tuning

To experiment SA algorithm, three runs are performed for each set of problem data, ending after a predetermined amount of processing time. We've conducted experiments using the following parameter settings:

`INIT_TEMP` = 350, then 500, then 700 (as initial temperature)

`UPDATE_TEMP` = lambda t: 0.9999 $*$ t (decreasing the temperature by multiplying of 0.9999 $*$ 0.01, we will set the cooling procedure by setting 0.3333, then 0.5555 then 0.9999)

`STOP_CRITERIA` = lambda t: $t <= 0.01$ (at temperate 0.01 the procedure will stop)

6.1.3 Comparison of Performances

Using three distinct sets of parameter values, we have executed the algorithm thrice for every dataset. The parameters we have specified for our first test (Test-1 findings in Table 6.2) are as follows:

Test-1:

For HGS and GLS (OR-Tools), `MaxRuntime` = 60 (in seconds)

For ACO, `ants_num` = 10, `q0` = 0.3, `beta` = 0.3, `rho` = 0.1, `runtime_in_minutes` = 2

For SA, `INIT_TEMP` = 350, `UPDATE_TEMP` = lambda t: 0.3333 $*$ t, `STOP_CRITERIA` = lambda t: $t <= 0.01$

We were able to obtain Test-1, which is displayed in Table 6.2, by establishing those parameters. (NV = Number of Vehicles, Costs = Total costs of route distance, BKS = Best Known Solutions)

Datasets	BKS	HGS	GLS (OR-Tools)	ACO	SA
-	NV / Costs	NV / Costs	NV / Costs	NV / Costs	NV / Costs
c101	10 / 827.3	10 / 827.3	10 / 827.3	11 / 1391.3	11 / 1019.1
c105	10 / 827.3	10 / 827.3	10 / 827.3	10 / 875.9	12 / 1728.2
c109	10 / 827.3	10 / 827.3	10 / 995.4	10 / 1318.9	11 / 1730.1
c201	3 / 589.1	3 / 589.1	3 / 589.1	4 / 1815.8	3 / 779.9
c204	3 / 588.1	3 / 588.1	3 / 611.9	4 / 2997.9	4 / 1538.9
c208	3 / 585.8	3 / 585.8	3 / 742.2	4 / 2297.1	3 / 794
r101	20 / 1637.7	20 / 1637.7	19 / 2605.1	20 / 1684.3	23 / 2507.9
r106	13 / 1234.6	13 / 1234.6	13 / 1488.6	13 / 1460.2	18 / 2066.6
r112	10 / 948.6	10 / 948.6	10 / 1094.3	11 / 1372.4	14 / 1745
r201	8 / 1143.2	8 / 1143.2	4 / 2534.5	4 / 1909.8	6 / 2519
r206	5 / 875.9	5 / 875.9	3 / 1549.1	3 / 1680.2	4 / 2088
r211	4 / 746.7	4 / 746.7	3 / 1278.8	3 / 1914.5	4 / 2071.2
rc101	15 / 1619.8	15 / 1619.8	15 / 2071.3	16 / 1843.5	20 / 2399
rc104	10 / 1132.3	10 / 1132.3	10 / 1310.7	12 / 1461.6	16 / 2041.1
rc108	11 / 1114.2	11 / 1114.2	10 / 1266.9	11 / 1496.5	16 / 2145.6
rc201	9 / 1261.8	9 / 1262.4	4 / 2347.8	4 / 2159.3	6 / 2979.2
rc204	4 / 783.5	4 / 783.5	3 / 1596.2	3 / 1624.9	5 / 1606.6
rc208	4 / 776.1	4 / 776.1	3 / 1294.2	4 / 2282.9	5 / 2619.5

Table 6.2: Test-1 results.

When we compare the findings to the $best - known\ solutions$, we can see that they are extremely diverse and a mix of good and poor results. ACO and SA are not producing good results at this time, but we plan to improve them by adjusting the parameters and displaying the change in Table 6.3 (as Test-2) and Table 6.4 (as Test-3). HGS is producing really good results on every different type of dataset. GLS (OR-Tools) is working well on clustered base datasets but not working well on random base datasets.

Test-2:

For HGS and GLS (OR-Tools), `MaxRuntime` = 120 (in seconds)

For ACO, `ants_num` = 20, `q0` = 0.5, `beta` = 0.5, `rho` = 0.5, `runtime_in_minutes` = 3

For SA, `INIT_TEMP` = 500, `UPDATE_TEMP` = lambda t: 0.5555 $* t$, `STOP_CRITERIA` = lambda t: $t <= 0.01$

We were able to obtain Test-2, which is displayed in Table 6.3, by establishing those parameters.

Test-2 results show that while HGS and GLS (OR-Tools) produce results that are comparable to each other, ACO and SA continue to produce poor results. To improve the performance of these algorithms, we must adjust their processing power, slow them down, and allow them more time to do neighbor searches.

Test-3:

For HGS and GLS (OR-Tools), `MaxRuntime` = 300 (in seconds)

For ACO, `ants_num` = 30, `q0` = 0.9, `beta` = 0.9, `rho` = 0.1, `runtime_in_minutes` = 5

For SA, `INIT_TEMP` = 700, `UPDATE_TEMP` = lambda t: 0.9999 $* t$, `STOP_CRITERIA` = lambda t: $t <= 0.01$

We were able to obtain Test-3, which is displayed in Table 6.4, by establishing those parameters. Table 6.4 displays the results, which indicate that the ACO and SA parameters have been adjusted to provide better outcomes than previously. (NV = Number of Vehicles, Costs = Total costs of route distance, BKS = Best Known Solutions)

Datasets	BKS	HGS	GLS (OR-Tools)	ACO	SA
-	NV / Costs	NV / Costs	NV / Costs	NV / Costs	NV / Costs
c101	10 / 827.3	10 / 827.3	10 / 827.3	10 / 890.7	11 / 1019.1
c105	10 / 827.3	10 / 827.3	10 / 827.3	10 / 919.2	12 / 1728.2
c109	10 / 827.3	10 / 827.3	10 / 995.4	10 / 1481.1	11 / 1729.6
c201	3 / 589.1	3 / 589.1	3 / 589.1	3 / 701.4	3 / 779.9
c204	3 / 588.1	3 / 588.1	3 / 611.9	4 / 2658.2	4 / 1621.5
c208	3 / 585.8	3 / 585.8	3 / 742.2	4 / 2031.1	3 / 794
r101	20 / 1637.7	20 / 1637.7	19 / 2519.3	19 / 1683.7	23 / 2529.9
r106	13 / 1234.6	13 / 1234.6	13 / 1484.4	13 / 1429.6	18 / 2039.4
r112	10 / 948.6	10 / 948.6	10 / 1100.6	11 / 1252.3	14 / 1714.1
r201	8 / 1143.2	8 / 1143.2	4 / 2544	4 / 2025.7	6 / 2509
r206	5 / 875.9	5 / 875.9	3 / 1528.2	3 / 1805.8	4 / 2049.5
r211	4 / 746.7	4 / 746.7	3 / 1278.8	3 / 1830.6	4 / 2118.6
rc101	15 / 1619.8	15 / 1619.8	15 / 2063	15 / 1788.6	20 / 2388.4
rc104	10 / 1132.3	10 / 1132.3	10 / 1310.7	11 / 1414.5	16 / 2020.8
rc108	11 / 1114.2	11 / 1114.2	10 / 1266.9	12 / 1533.7	16 / 2177.4
rc201	9 / 1261.8	9 / 1261.8	4 / 2347.8	5 / 2412.2	6 / 2963.3
rc204	4 / 783.5	4 / 783.5	3 / 1487.6	3 / 1645.1	5 / 1626.5
rc208	4 / 776.1	4 / 776.1	3 / 1254.4	4 / 2396.3	5 / 2573.6

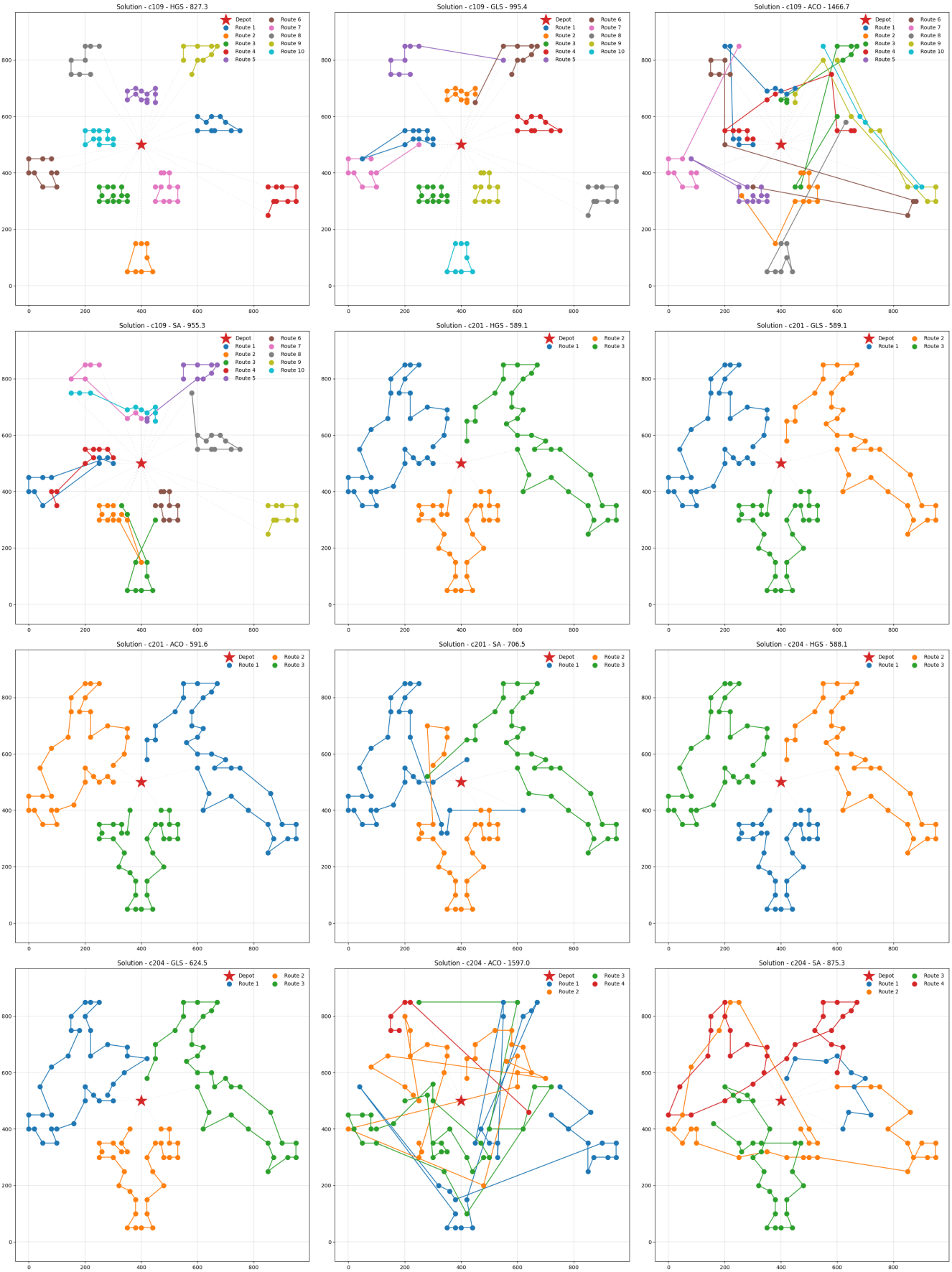
Table 6.3: Test-2 results.

Datasets	BKS	HGS	GLS (OR-Tools)	ACO	SA
-	NV / Costs	NV / Costs	NV / Costs	NV / Costs	NV / Costs
c101	10 / 827.3	10 / 827.3	10 / 827.3	10 / 852.9	11 / 934.9
c105	10 / 827.3	10 / 827.3	10 / 827.3	10 / 849	11 / 923.5
c109	10 / 827.3	10 / 827.3	10 / 995.4	10 / 1466.7	10 / 955.3
c201	3 / 589.1	3 / 589.1	3 / 589.1	3 / 591.6	3 / 706.5
c204	3 / 588.1	3 / 588.1	3 / 624.5	4 / 1597	4 / 875.3
c208	3 / 585.8	3 / 585.8	3 / 742.2	3 / 895.8	3 / 686.5
r101	20 / 1637.7	20 / 1637.7	19 / 2570.5	20 / 1652.6	20 / 1710
r106	13 / 1234.6	13 / 1234.6	12 / 1435.6	13 / 1408.7	14 / 1332.7
r112	10 / 948.6	10 / 948.6	10 / 1100.6	10 / 1159.8	11 / 1047.1
r201	8 / 1143.2	8 / 1143.2	4 / 2509.6	4 / 2057.3	5 / 1258
r206	5 / 875.9	5 / 875.9	3 / 1573.7	3 / 1732.8	3 / 1007.8
r211	4 / 746.7	4 / 746.7	3 / 1237.4	3 / 1484.7	3 / 904
rc101	15 / 1619.8	15 / 1619.8	15 / 2063	16 / 1822.6	15 / 1700.1
rc104	10 / 1132.3	10 / 1132.3	10 / 1253.7	11 / 1320.5	11 / 1291.9
rc108	11 / 1114.2	11 / 1114.2	10 / 1266.9	11 / 1321.8	12 / 1237.6
rc201	9 / 1261.8	9 / 1261.8	4 / 2357.8	4 / 2533.5	6 / 1469
rc204	4 / 783.5	4 / 783.5	3 / 1538	3 / 1542.2	4 / 927.6
rc208	4 / 776.1	4 / 776.1	3 / 1254.4	3 / 1662.7	4 / 944

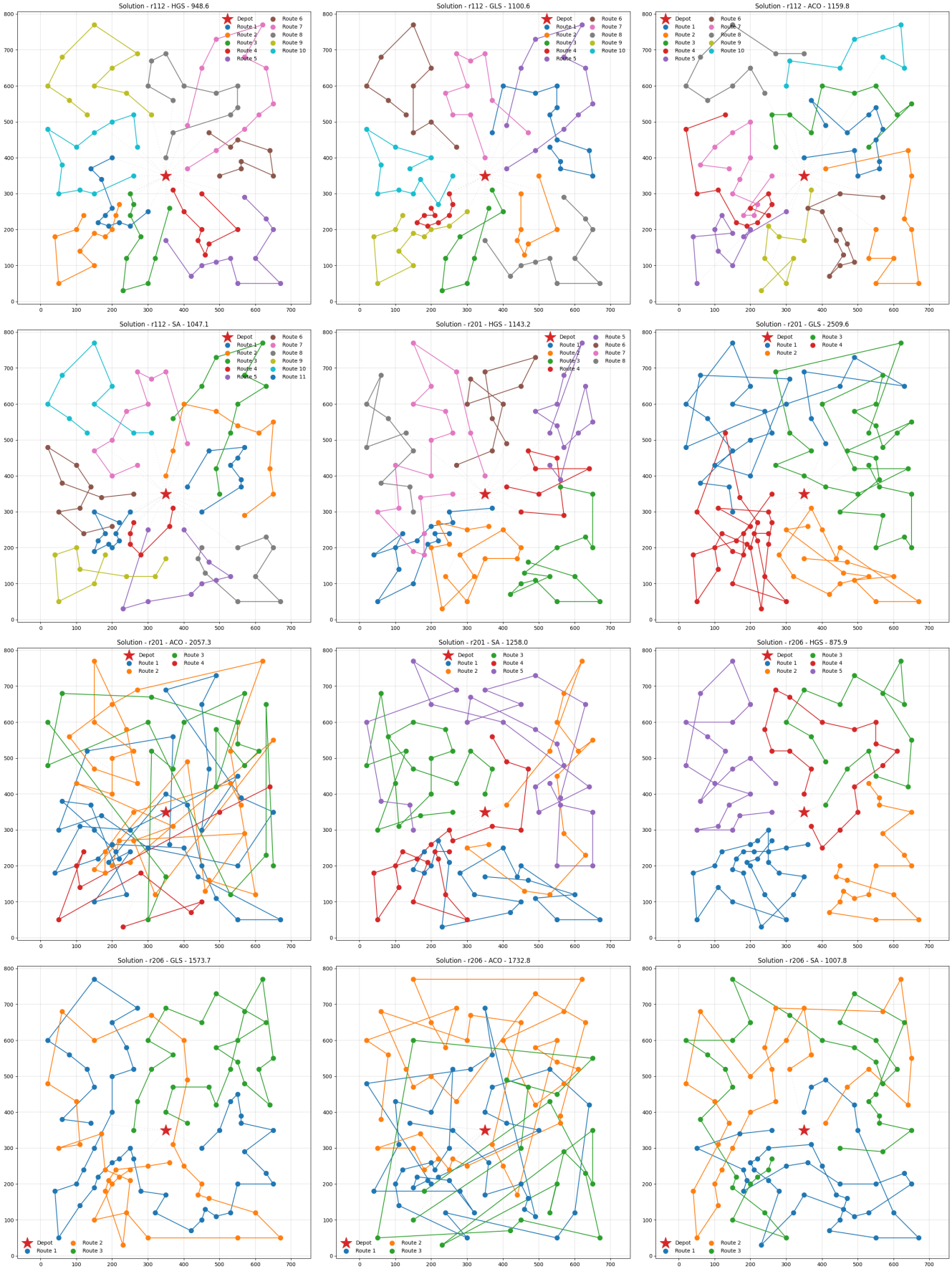
Table 6.4: Test-3 results.

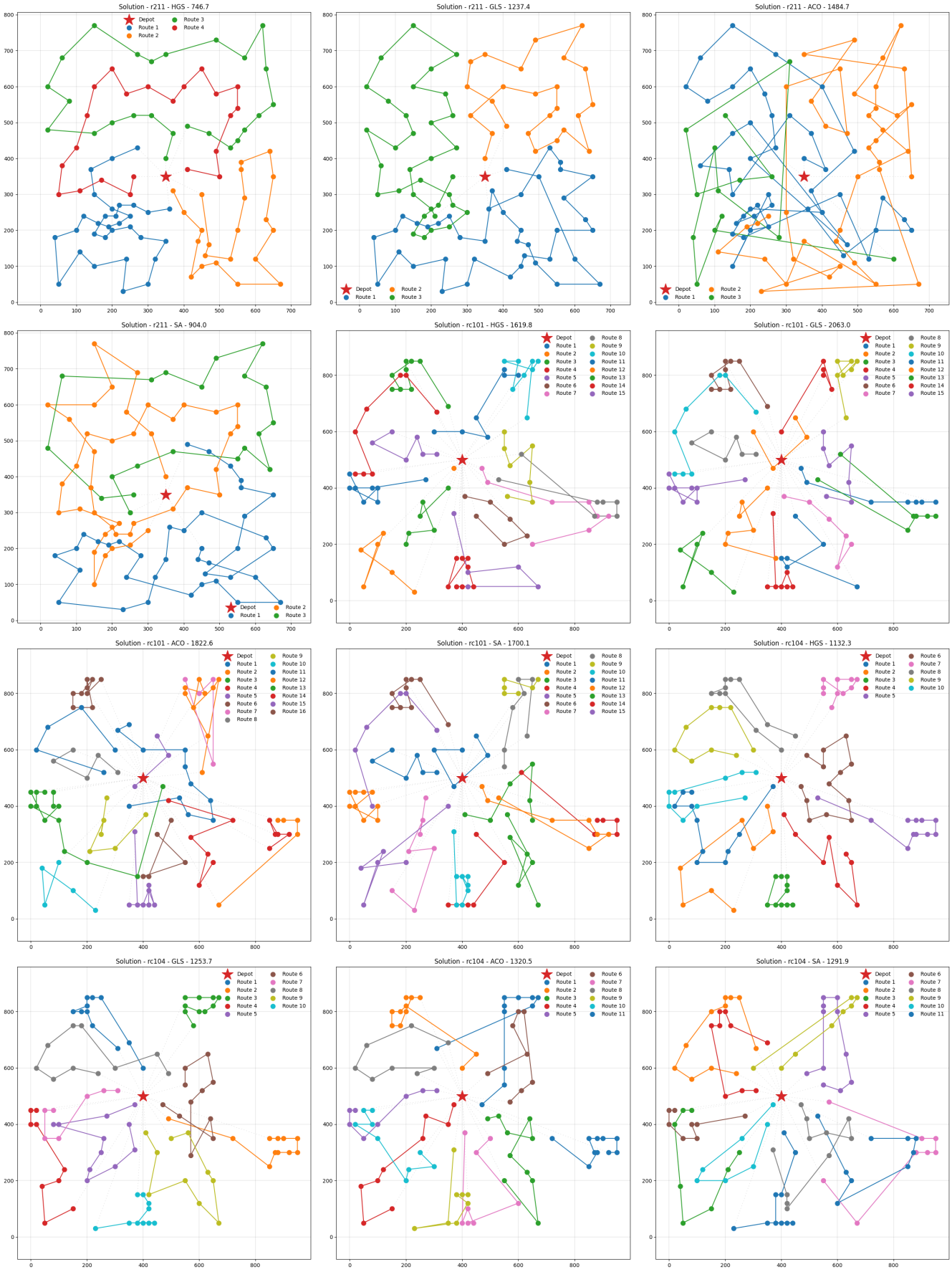
We will display the graphical depiction of Test-3, which provides better results on each method, instead of the graph results of Tests-1 and-2, as they do not provide generally better results.

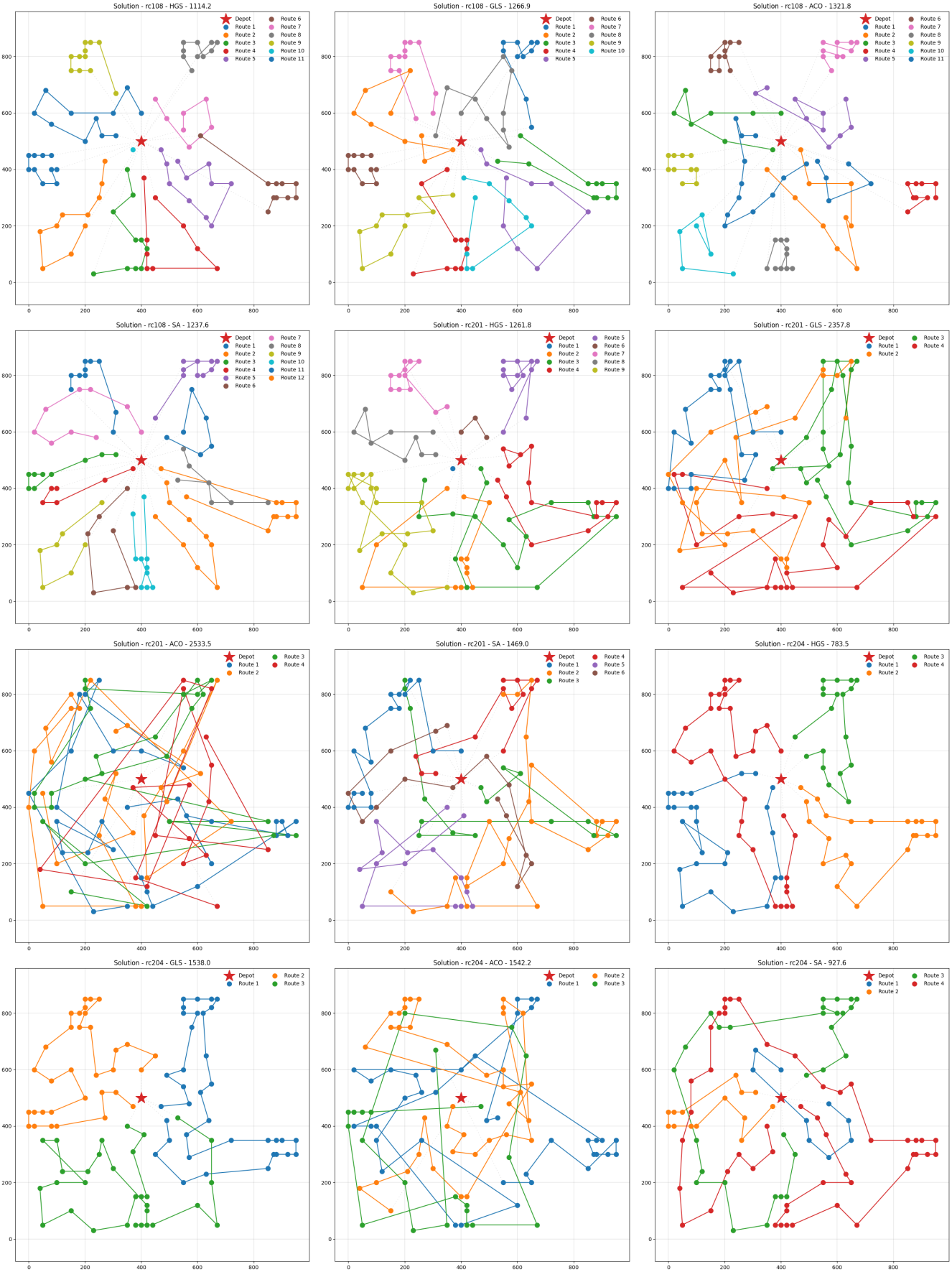
Figure 6.1: Test-3 results of routes in graph of C101 and c105.


Solution - c109 - HGS - 827.3
Solution - c109 - GLS - 995.4
Solution - c109 - ACO - 1466.7
Solution - c109 - SA - 955.3
Solution - c201 - HGS - 589.1
Solution - c201 - GLS - 589.1
Solution - c201 - ACO - 591.6
Solution - c201 - SA - 706.5
Solution - c204 - HGS - 588.1
Solution - c204 - GLS - 624.5
Solution - c204 - ACO - 1597.0
Solution - c204 - SA - 875.3
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10


Solution - c208 - HGS - 585.8
Solution - c208 - GLS - 742.2
Solution - c208 - ACO - 895.8
Solution - c208 - SA - 686.5
Solution - r101 - HGS - 1637.7
Solution - r101 - GLS - 2570.5
Solution - r101 - ACO - 1652.6
Solution - r101 - SA - 1710.0
Solution - r106 - HGS - 1234.6
Solution - r106 - GLS - 1435.6
Solution - r106 - ACO - 1408.7
Solution - r106 - SA - 1332.7


Solution - r112 - HGS - 948.6
Solution - r112 - GLS - 1100.6
Solution - r112 - ACO - 1159.8
Solution - r112 - SA - 1047.1
Solution - r201 - HGS - 1143.2
Solution - r201 - GLS - 2509.6
Solution - r201 - ACO - 2057.3
Solution - r201 - SA - 1258.0
Solution - r206 - HGS - 875.9
Solution - r206 - GLS - 1573.7
Solution - r206 - ACO - 1732.8
Solution - r206 - SA - 1007.8


Solution - r211 - HGS - 746.7
Depot
Route 1
Route 2
Route 3
Route 4
Solution - r211 - GLS - 1237.4
Depot
Route 1
Route 2
Route 3
Solution - r211 - ACO - 1484.7
Depot
Route 1
Route 2
Route 3
Solution - r211 - SA - 904.0
Depot
Route 1
Route 2
Route 3
Solution - rc101 - HGS - 1619.8
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Route 11
Route 12
Route 13
Route 14
Route 15
Solution - rc101 - GLS - 2063.0
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Route 11
Route 12
Route 13
Route 14
Route 15
Solution - rc101 - ACO - 1822.6
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Route 11
Route 12
Route 13
Route 14
Route 15
Route 16
Solution - rc101 - SA - 1700.1
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Route 11
Route 12
Route 13
Route 14
Route 15
Solution - rc104 - HGS - 1132.3
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Solution - rc104 - GLS - 1253.7
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Solution - rc104 - ACO - 1320.5
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Route 11
Solution - rc104 - SA - 1291.9
Depot
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Route 9
Route 10
Route 11


Solution - rc108 - HGS - 1114.2
Solution - rc108 - GLS - 1266.9
Solution - rc108 - ACO - 1321.8
Solution - rc108 - SA - 1237.6
Solution - rc201 - HGS - 1261.8
Solution - rc201 - GLS - 2357.8
Solution - rc201 - ACO - 2533.5
Solution - rc201 - SA - 1469.0
Solution - rc204 - HGS - 783.5
Solution - rc204 - GLS - 1538.0
Solution - rc204 - ACO - 1542.2
Solution - rc204 - SA - 927.6

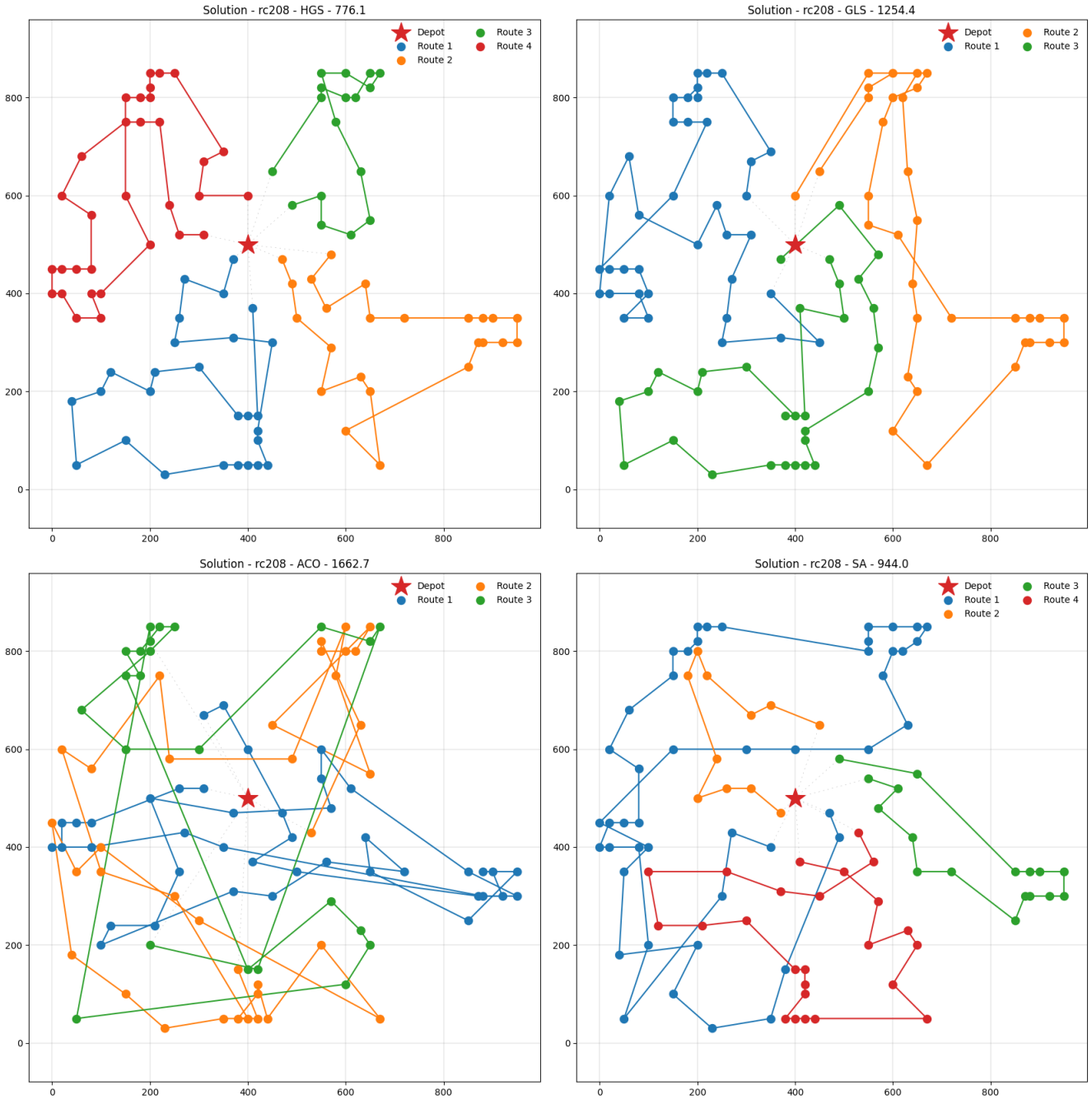


Figure 6.2: Test-3 results of routes in graph of rc208.

Chapter 7

Summary and Conclusion

7.1 Summary

Here are the observations we have made thus far from Test-1 (Table 6.2), Test-2 (Table 6.3), and Test-3 (Table 6.4) after observing all the outcomes by running the program on clustered (c-types), random (r-types), and random clustered (rc-types) Solomon datasets:

- Hybrid Genetic Search (HGS) provides good performance on all kinds of datasets. This indicates that merging numerous metaheuristics or conventional heuristics into one algorithm and applying the combined method to VRPTW issues yields improved performance.
- My research has thus far led us to the conclusion that algorithms using population metaheuristics inspired by nature perform better on VRPTW challenges.
- Furthermore, hybrid algorithms outperform pure metaheuristics or classic heuristic algorithms in solving the problem.
- Using a neighbor search or local search method is preferable instead of simply setting a random initial solution generator function to compute itself for the first solution at the beginning of the procedure.
- We have observed that the Guided Local Search (GLS) algorithm when used with OR-Tools to initialize the solutions by First Solution Strategy method then GLS improves the solutions, performs best on clustered (c-types) datasets; on random (r-types) datasets, however, it performs poorly, and on random clustered (rc-types) datasets, it performs a mix of good and poor results. But it finds the solutions pretty quickly.
- When the Ant Colony Optimization (ACO) algorithm is tested using both single and multiple ant colony systems (MACS), MACS consistently produces better results than the ACO method in its raw form. However, it takes a while to identify the solutions.
- When it comes to random datasets (r-types), MACS solving on VRPTW performs best; on clustered datasets (c-types), however, it does not perform as well as random; and on random clustered datasets (rc-types), it produces a combination of good and poor outcomes.
- Through testing the Simulated Annealing (SA) algorithm in its raw form with the help of Greedy Randomised Adaptive Search Procedure (GRASP) as initiator, we have discovered that while it doesn't always produce the best results, it does produce steady and reasonable outcomes when compared to BKS and other results we have obtained thus far (it is true for results on a variety of datasets, regardless of whether they are clustered, random, or random clustered). But solving the problems takes time.

7.2 Future Work

We think that if we extend the test duration and tweak the ACO and SA parameters, we should be able to obtain better results on all kinds of datasets that are in the neighborhood of the best-known solutions. For that, we need more time. As a result, we want to keep tweaking the parameters until we reach the ideal configuration for our algorithms.

We intend to keep working on GLS, adjusting its parameters with OR-Tools to improve its ability to compute results on random (r-types) datasets. More importantly, we want to use the various algorithms that OR-Tools offers as options so that we can set different algorithms in the `solve_model` to see how the results would be. These algorithms include the Savings algorithm, Sweep algorithm, and Christofides algorithm on `FirstSolutionStrategy` function, and Tabu Search, Greedy Descent, Simulated Annealing, and Generic Tabu Search on `LocalSearchMetaheuristic` function.

7.3 Conclusion

The study addresses several innovative components for controlling the temporal dimension and provides efficient approaches for time-constrained vehicle routing problems including Hybrid Genetic Search (HGS), Guided Local Search (GLS), Ant Colony Optimization (ACO), and Simulated Annealing (SA). Novel move evaluation techniques are put forth that enable the evaluation of movements from any classical neighborhood in terms of the time window. We have developed multiple models using multiple classic heuristics and metaheuristics algorithms that adhere to the state-of-the-art algorithms standard, compared the results with each one including the best-known solutions, and represented the results in graphs. We have demonstrated that the combined classic heuristic or metaheuristic algorithms yield better results (as we can see, HGS has produced 95% of the same outcomes as the most well-known solutions).

Appendix A

Programmcode

A.1 HGS Methods Implementation

To resolve VRPTW cases which employs a hybrid genetic search algorithm, we have depended on the `Model` interface. We will solve the VRPTW benchmark dataset of Solomon, `C101.txt`, as an example. There are one hundred clients in this instance.

```
        from pyvrp import read
        INSTANCE = read("data/C101.txt", "solomon", "trunc1")

```

We are going to construct a `solve()` function that accepts a random number generator `seed` and a `stop` stopping criterion. This definition is almost the same as `Model.solve`.

```
        def solve(stop: StoppingCriterion, seed: int) ->     Solution: ...

```

A.1.1 A Tour of HGS model

The `GeneticAlgorithm` object is responsible for overseeing a group of solutions. Employing a crossover operator from `pyvrp.crossover`, two solutions are chosen from this population for crossover in each iteration, producing a new offspring solution. We then refine that offspring solution with a technique from `pyvrp.search`. Next, the population is expanded to include the enhanced offspring solution. Unless a stopping condition is met (see `pyvrp.stop` for various conditions), this procedure keeps going.

We must first define the initial population, search strategy, penalty manager, and random number generator to implement the `GeneticAlgorithm`.

Random Number Generator

```
        from pyvrp import RandomNumberGenerator
        rng = RandomNumberGenerator(seed=42)

```

Search Method

We have developed a very configurable `LocalSearch` approach that utilizes several operators and search neighborhoods. Diverse operators explore distinct regions throughout the solution space, potentially leading to more optimal solutions. Which edges are examined, is determined by the neighborhood. The local search strategy operates significantly more quickly by limiting that set of edges. We offer neighborhoods and default operator sets that can be utilized in the ways listed below.

```
        from pyvrp.search import (
            LocalSearch,
            NODE_OPERATORS,
            ROUTE_OPERATORS,
            compute_neighbours,
        )

        neighbours = compute_neighbours(INSTANCE)
        ls = LocalSearch(INSTANCE, rng, neighbours)

        for node_op in NODE_OPERATORS:
        ls.add_node_operator(node_op(INSTANCE))

        for route_op in ROUTE_OPERATORS:
        ls.add_route_operator(route_op(INSTANCE))

```

Solution Representation and Evaluation

Our local search technique is currently operational. It only requires two more parts to function: a `CostEvaluator` which can be employed to evaluate various moves and a `Solution` that outlines a set of routes.

```
        from pyvrp import Solution, CostEvaluator

        cost_evaluator = CostEvaluator(capacity_penalty=20, tw_penalty=20)
        sol = Solution.make_random(INSTANCE, rng)

```

We currently have a random solution `sol`, however, it is not feasible yet. This is not an issue because penalties are used internally by HGS to assess each solution's viability. This is accomplished by utilizing the `penalised_cost` function of the `CostEvaluator`, which also enables us to assess the quality of unfeasible solutions.

```
        assert not sol.is_feasible()
        print(cost_evaluator.penalised_cost(sol))

```

Let's investigate whether the local search can make this solution even better.

```
        new_sol = ls.search(sol, cost_evaluator)

        assert not sol.is_feasible()
        print(cost_evaluator.penalised_cost(new_sol))

```

Much improved. However, the new solution is not feasible yet. We must strengthen the penalties forced through the impossibilities.

```
        cost_evaluator = CostEvaluator(capacity_penalty=200, tw_penalty=200)
        new_sol = ls.search(sol, cost_evaluator)

        assert new_sol.is_feasible()

```

Again to observe the solutions.

```
        print(cost_evaluator.penalised_cost(new_sol))

```

It's getting better now. HGS ensures that a sufficient number of feasible solutions are found by adjusting the penalty parameters. This is how it handles infeasibilities. The penalties increase with too few feasible options to choose from; they decrease with too many. This guarantees a balanced population of workable and impractical options, which promotes crossover and diversity.

The `PenaltyManager` is the entity responsible for managing the penalty terms. It is possible to request a `CostEvaluator` in the format we previously observed.

```
        from pyvrp import PenaltyManager

        pen_manager = PenaltyManager()
        cost_evaluator = pen_manager.get_cost_evaluator()

```

Population Management

We have to provide a `Population` along with a set of initial solutions (random).

```
        from pyvrp import Population
        from pyvrp.diversity import broken_pairs_distance

        pop = Population(broken_pairs_distance)

```

The population monitors the range of its responses. There are various methods for calculating the variety (dissimilarity) of two solutions. `pyvrp.diversity` has functions for doing this, which may be given to the `Population`. In this case, we employ the `broken_pairs_distance`, which, given the number of dissimilar edges in the solutions, computes a number in [0,1].

A new population is initially empty:

```
        assert len(pop) == 0

```

`Population.add` allows us to augment the population with new solutions. Remember that the solutions `sol` and `new_sol` are, respectively, infeasible and feasible.

```
        assert not sol.is_feasible()
        pop.add(sol, cost_evaluator)

        assert new_sol.is_feasible()
        pop.add(new_sol, cost_evaluator)

        assert len(pop) == 2
        assert pop.num_feasible() == 1
        assert pop.num_infeasible() == 1

```

Genetic Algorithm and Crossover

By constructing a list of random solutions, it is simple to produce a set of initial solutions.

```
        init_sols = [Solution.make_random(INSTANCE, rng) for _ in range(25)]

```

Building the genetic algorithm is now possible. In addition, this object accepts a crossover operator from `pyvrp.crossover`. Selective route exchange, or SREX, will be our method.

```
        from pyvrp import GeneticAlgorithm
        from pyvrp.crossover import selective_route_exchange as srex

        algo = GeneticAlgorithm(
            INSTANCE,
            pen_manager,
            rng,
            pop,
            ls,
            srex,
            init_sols,
        )

```

`algo.run` can be invoked, the algorithm iterating until a halting condition is satisfied. You can import `pyvrp.stop` to get these stopping criteria.

```
        from pyvrp.stop import MaxIterations, MaxRuntime

        iter_res = algo.run(stop=MaxIterations(500))
        time_res = algo.run(stop=MaxRuntime(1))  # seconds

```

To observe the solutions.

```
        print(iter_res)

```

Solve Function

Combines all of our knowledge into a `solve` function.

```
        def solve(stop, seed):
            rng = RandomNumberGenerator(seed=seed)
            pm = PenaltyManager()
            pop = Population(broken_pairs_distance)

            neighbours = compute_neighbours(INSTANCE)
            ls = LocalSearch(INSTANCE, rng, neighbours)

            for node_op in NODE_OPERATORS:
            ls.add_node_operator(node_op(INSTANCE))

            for route_op in ROUTE_OPERATORS:
            ls.add_route_operator(route_op(INSTANCE))

            init = [Solution.make_random(INSTANCE, rng) for _ in range(25)]
            algo = GeneticAlgorithm(INSTANCE, pm, rng, pop, ls, srex, init)

            return algo.run(stop)

```

Uses the `solve` function to solve the instance once more.

```
        res = solve(stop=MaxIterations(1000), seed=1)
        print(res)

```

In order to examine a data instance or solution result, HGS additionally offers a wide range of plotting tools.

```
        import matplotlib.pyplot as plt
        from pyvrp.plotting import plot_result

```

```
        fig = plt.figure(figsize=(12, 8))

        plot_result(res, INSTANCE, fig=fig)
        plt.tight_layout()

```

In this model, we've implemented our own `solve` function using a few of the various HGS components.

Additionally, distinct search algorithms can be created using the parts we observed in this notebook. For instance, an iterated local search strategy might be easily implemented using our `LocalSearch` search technique, and various VRPTW models may be plotted using our `pyvrp.charting` plotting method. The HGS package can be heavily reused because of its modularity.

A.1.2 HGS API Reference

Several essential classes required to execute the VRP solver are exposed by the HGS module. These consist of the `Population` that oversees a `Solution` pool and the core `GeneticAlgorithm`. While the majority of classes accept

parameter objects that enable sophisticated setup, they also have acceptable defaults. The `GeneticAlgorithm` finally provides a `Result` object after running. The best-observed solution and comprehensive runtime statistics can be obtained using this object.

```
        add_client(
            x: int,
            y: int,
            demand: int = 0,
            service_duration: int = 0,
            tw_early: int = 0,
            tw_late: int = 0,
            release_time: int = 0,
            prize: int = 0,
            required: bool = True
        ) -> Client

```

```
        add_depot(
            x: int,
            y: int,
            tw_early: int = 0,
            tw_late: int = 0
        ) -> Client

```

```
        add_edge(
            frm: Client,
            to: Client,
            distance: int,
            duration: int = 0
        ) -> Edge

```

Creates a new edge (i, j) connecting `frm` (i) and (j). Both duration and distance attributes can be applied to the edge. Although the default duration is 0, distance is required.

```
        add_vehicle_type(
            capacity: int,
            num_available: int,
            fixed_cost: int = 0,
            tw_early: int | None = None,
            tw_late: int | None = None
        ) -> VehicleType

```

Uses the attributes of this model to create and return a `ProblemData` object.

```
        solve(
            stop: StoppingCriterion,
            seed: int = 0
            ) -> Result

```

`solve` this model.

```
        class Edge(
            frm: Client,
            to: Client,
            distance: int,
            duration: int
        )

```

Keeps an edge between two locations in storage.

```
        class GeneticAlgorithm(
            data: ProblemData,
            penalty_manager: PenaltyManager,
            rng: RandomNumberGenerator,
            population: Population,
            search_method: SearchMethod,
            crossover_op: Callable[[tuple[Solution, Solution], ProblemData, CostEvaluator,
    RandomNumberGenerator], Solution],
            initial_solutions: Collection[Solution],
            params: GeneticAlgorithmParams = GeneticAlgorithmParams(repair_probability=0.8,
    nb_iter_no_improvement=20000)
        )

```

Creates a GA instance.

Penalty Manager

```
        class PenaltyParams(
            init_capacity_penalty: int = 20, init_time_warp_penalty: int = 6,
            repair_booster: int = 12, num_registrations_between_penalty_updates: int = 50,
            penalty_increase: float = 1.34,
            penalty_decrease: float = 0.32,
            target_feasible: float = 0.43
        )

```

Penalty manager parameters.

`init_capacity_penalty` : The first charge for using more capacity. This is the amount that the objective, at the beginning of the search, penalizes one unit of excess load capacity.

`init_time_warp_penalty` : Initial penalty on the time warp. This is the amount that is deducted from the target at the beginning of the search for each unit of time warp (time window violations).

`repair_booster` : A value $r \geq 1$ for a repair booster. The current penalty terms are temporarily multiplied by this figure in order to enforce feasibility.

`num_registrations_between_penalty_updates` : Number of registrations for feasibility in between changes to the penalty value. Periodically, the penalty manager modifies the penalty terms in light of new feasibility registrations. This parameter sets the frequency at which these updates take place.

`penalty_increase` : Amount $p_i \geq 1$ by which the existing penalties are raised when not enough workable solutions from the most recent registrations are discovered. As $v \leftarrow p_i v$, the penalty values v are updated.

`penalty_decrease` : Amount $p_d \in [0, 1]$ by which, when enough workable solutions have been identified among the most recent registrations, the existing penalties are reduced. As $v \leftarrow p_d v$, the penalty values v are updated.

`target_feasible` : The target percentage $p_f \in [0, 1]$ of feasible registrations in the most recent `num_registrations_between_penalty_updates` registrations. This percentage is applied to update the penalty conditions; for example, fines are increased when not enough feasible solutions are reported, and penalties are dropped when too many possible solutions are registered. This guarantees a balanced population, with solutions that are fractionally practicable (p_f) and fractionally infeasible ($1 - p_f$).

```
        class PenaltyManager(params: PenaltyParams = PenaltyParams(init_capacity_penalty=20,
    init_time_warp_penalty=6, repair_booster=12, num_registrations_between_penalty_updates=50,
    penalty_increase=1.34, penalty_decrease=0.32, target_feasible=0.43))

```

Creates a `PenaltyManager` instance. For specified time warp and load values, this class gives penalty terms and handles time warp and load penalties. It can be used to create a temporary penalty booster object that raises the penalties for a brief period. It updates these penalties based on recent history.

Population

```
class PopulationParams(
    min_pop_size: int = 25,
    generation_size: int = 40,
    nb_elite: int = 4,
    nb_close: int = 5,
    lb_diversity: float = 0.1,
    ub_diversity: float = 0.5
)
```

Generates an object for parameters that can be used with `Population`.

```
class Population(
    diversity_op: Callable[[Solution, Solution], float],
    params: PopulationParams | None = None
)
```

Creates a `Population` instance.

```
select(
    rng: RandomNumberGenerator,
    cost_evaluator: CostEvaluator,
    k: int = 2
) -> tuple[Solution, Solution]
```

Subject to a diversity restriction, chooses two (if possible, non-identical) parents for each tournament.

```
get_tournament(
    rng: RandomNumberGenerator,
    cost_evaluator: CostEvaluator,
    k: int = 2
) -> Solution
```

Using a $k-ary$ tournament, chooses a solution from this population depending on the (internal) fitness values of the chosen solutions.

Reading VRPLIB

```
read(
    where: str | Path,
    instance_format: str = 'vrplib',
    round_func: str | Callable[[ndarray], ndarray] = 'none'
) -> ProblemData
```

Returns an instance of `ProblemData` after reading the VRPLIB file from the specified location.

```
read_solution(
    where: str | Path
) -> list[list[int]]
```

Returns the routes contained in a VRPLIB solution that has been read from a file at the specified location.

```
class Result(
    best: Solution,
    stats: Statistics,
    num_iterations: int,
    runtime: float
)
```

Keeps track of a single run's results. The `GeneticAlgorithm` returns an instance of this class upon completion.

Cost Evaluator

```
class CostEvaluator(capacity_penalty: int, tw_penalty: int)
```

Establishes an instance of `CostEvaluator`. Penalties for a given time warp and load can be computed using this class, which also provides time warp and load penalties.

```
cost(self, solution: Solution) -> int
```

A collection of routes r makes up each solution. Each route $R \in r$ consists of a series of edges that begin and terminate at the depot. The vehicle type t_R assigned to a route R has a fixed vehicle cost f_{t_R}. Let $V_R = \{i : (i,j) \in R\}$ represent the set of locations that route R has traveled. Next, the target value is provided by-

$$\sum_{R \in r} \left[f_{t_R} + \sum_{(i,j) \in R} d_{ij} \right] + \sum_{i \in V} p_i - \sum_{R \in r} \sum_{i \in V_R} p_i \tag{A.1.1}$$

Where the unclaimed awards of unvisited customers are listed in the second part, while the vehicle and distance expenses are listed in the first.

```
load_penalty(self, load: int, capacity: int) -> int
```

Calculates the total penalty for excess capacity for the specified load.

```
penalised_cost(self, solution: Solution) -> int
```

Calculates the penalized cost, or smoothed objective, for a given solution.

```
tw_penalty(self, time_warp: int) -> int
```

Calculates the penalty for time warp for the specified time warp.

```
class Route(
    data: ProblemData,
    visits: list[int],
    vehicle_type: int
)
```

A Class that holds some statistics and the route plan.

```
make_random(
    data: ProblemData,
    rng: RandomNumberGenerator
) -> Solution
```

Generates a solution at random.

```
class Client(
    x: int,
    y: int,
    demand:
    int = 0,
    service_duration: int = 0,
    tw_early: int = 0,
    tw_late: int = 0,
    release_time: int = 0,
    prize: int = 0,
    required: bool = True
)
```

All customer data is stored as (read-only) properties in a data object.

```
        class VehicleType(
            capacity: int,
            num_available: int,
            fixed_cost: int = 0,
            tw_early: int | None = None,
            tw_late: int | None = None
        )

```

All vehicle type data is stored as properties in a data object.

```
        class ProblemData(
            clients: list[Client],
            depots: list[Client],
            vehicle_types: list[VehicleType],
            distance_matrix: list[list[int]],
            duration_matrix: list[list[int]]
        )

```

Generates an instance of a problem data. All the information required to resolve the vehicle routing issue is contained in this instance.

Crossover Operators

`pyvrp.crossover` module gives operators the tools they need to create a new `Solution` offspring by combining two parent solutions.

```
        selective_route_exchange(
            parents: tuple[Solution, Solution],
            data: ProblemData,
            cost_evaluator: CostEvaluator,
            rng: RandomNumberGenerator
        ) -> Solution

```

In order to create a new offspring solution, this crossover operator, credited to Nagata and Kobayashi [241], merges routes from both parents. In order to accomplish this, it carefully chooses routes from the second parent that are interchangeable with those from the first. This frequently leads to imperfect offspring, which can subsequently be fixed via a search technique.

Diversity Measures

The operators to calculate the relative difference between two `Solution` objects are found in the `pyvrp.diversity` module. This distinction offers a diversity metric: The genetic algorithm performs better when there is a `Population` of extremely different solutions. One wants to strike a balance between diversity and quality at the same time; solutions must be good. By calculating a fitness score for every solution and balancing the diversity with the objective value, that balance is kept.

```
        broken_pairs_distance(
            first: Solution,
            second: Solution
        ) -> float

```

Determines the two solutions' symmetric broken pairs distance (BPD). Whether a customer in the problem has neighbors in both the first and second solution is determined by this function. If not, the customer belongs to a "broken pair" which is a connection that is included in one solution but excluded from another.

Considering the two solutions The previous customer (or depot) of the customer $i = 1, \ldots, n$ in f and s, respectively, should be denoted by the variables $p_f(i)$ and $p_s(i)$. For the next customer (or depot), define $s_f(i)$ and $s_s(i)$ in a similar manner. Then, we have

$$BPD(f, s) = \frac{\sum_{i=1}^{n} 1_{p_f(i) \neq p_s(i)} + 1_{s_f(i) \neq s_s(i)}}{2n} \tag{A.1.2}$$

Repair Operators

Operators in charge of repairing a `Solution` in a large neighborhood search (LNS) context are provided with resources by the `pyvrp.repair` module.

```
        greedy_repair(
            solution: Solution,
            unplanned: list[int],
            data: ProblemData,
            cost_evaluator: CostEvaluator
        ) -> Solution

        greedy_repair(
            routes: list[Route],
            unplanned: list[int],
            data: ProblemData,
            cost_evaluator: CostEvaluator
        ) -> Solution

```

With this greedy repair operator, each customer on the list of unexpected customers is inserted into the solution. To achieve this, it calculates a quadratic runtime by analyzing every move that may be made and applying the best one for each customer.

```
        nearest_route_insert(
            routes: list[Route],
            unplanned: list[int],
            data: ProblemData,
            cost_evaluator: CostEvaluator
        ) -> Solution

```

Operator for nearest route insert. Each customer in the list of unexpected customers is inserted by this operator into one of the designated routes. It accomplishes this by first figuring out which route contains the customer's closest center point, and then assessing all potential customer insert moves into that closest route. The best move is employed. At the expense of some solution quality, this operator is usually significantly more efficient than `greedy_repair()`, although in the worst situation it has a quadratic runtime.

Search Methods

The classes and search functions in the `pyvrp.search` module is in charge of enhancing a newly constructed offspring solution. This occurs immediately following the `GeneticAlgorithm`'s `pyvrp.crossover`. As of right now, HGS has a `LocalSearch` method. The `SearchMethod` protocol is implemented by all search methods.

```
        class SearchMethod(*args, **kwargs)

```

```
        __call__(
            solution: Solution,
            cost_evaluator: CostEvaluator
        ) -> Solution

```

Dig around the provided solution to produce a new, perhaps better solution. Also, the `search()` and `intensify()` techniques are used in this method to iteratively improve the provided solution. `search()` is used first. `intensify()` is subsequently used. Until no more advancements are discovered, this is repeated. Ultimately, the improved resolution is given back.

```
        class LocalSearch(
            data: ProblemData,
            rng: RandomNumberGenerator,
            neighbours: list[list[int]]
        )

```

Local search strategy uses user-provided node and route operators to efficiently explore a granular neighborhood. This leads swiftly to improved solutions.

```
        add_node_operator(op: NodeOperator)

```

This local search object has a node operator added to it. `search()` will make use of the node operator to improve a solution.

```
        add_route_operator(op: RouteOperator)

```

This local search object has a route operator added to it. `intensify()` will employ the route operator to improve a solution by utilizing more costly route operators.

```
        set_neighbours(neighbours: list[list[int]])

```

A way to replace the local search object's current granular neighborhood.

```
        get_neighbours() -> list[list[int]]

```

Provides the granular neighborhood that the local search is currently using.

```
        intensify(
            solution: Solution,
            cost_evaluator: CostEvaluator,
            overlap_tolerance: float = 0.05
        ) -> Solution

```

To improve the provided solution, this method makes advantage of the intensifying route operators on this local search object. The `overlap_tolerance` option can be used to restrict the number of route pairs that are evaluated, hence reducing the computing needs of intensification.

```
        search(
            solution: Solution,
            cost_evaluator: CostEvaluator
        ) -> Solution

```

This technique improves the provided solution by using the node operators on this local search object.

```
        class NeighbourhoodParams(
            weight_wait_time: float = 0.2,
            weight_time_warp: float = 1.0,
            nb_granular: int = 40,
            symmetric_proximity: bool = True, symmetric_neighbours: bool = False
        )

```

Setup for computing a fine-grained neighborhood.

`weight_wait_time` : The proximity calculation's minimal wait time component is given weight. A lengthy wait time suggests that customers are spaced out in terms of time.

`weight_time_warp` : The minimum time warp component of the proximity estimate is given weight. The customers are spaced out in terms of time and duration by a large time warp.

`nb_granular` : The total number of other customers in the specific neighborhood of each customer. This option establishes the total neighborhood size.

`symmetric_proximity` : In case a symmetric proximity matrix needs to be computed. In this way, edge (i, j) is guaranteed to have the same weight as (j, i).

`symmetric_neighbours` : if the neighborhood structure should be symmetrical. This guarantees that (j, i) is in when the edge (i, j) is in. Keep in mind that this isn't the same as `proximity_symmetric` .

```
        compute_neighbours(
            data: ProblemData,
            params: NeighbourhoodParams = NeighbourhoodParams(weight_wait_time=0.2,
    weight_time_warp=1.0, nb_granular=40, symmetric_proximity=True, symmetric_neighbours=False)
        ) -> list[list[int]]

```

Calculates how a problem instance's neighbors define the neighborhood.

Node Operators

The `add_node_operator()` function can be used to add instances of these operators to the `LocalSearch` object. Every operator for a node inherits from `NodeOperator` . All of these operators are conveniently available as `NODE_OPERATORS` through the `pyvrp.search` module:

```
        from pyvrp.search import NODE_OPERATORS

```

```
        class NodeOperator

```

```
        class Exchange10(data: ProblemData)

```

N consecutive customers from U's route (beginning at U) are exchanged with M consecutive customers from V's route (beginning at V) by the (N, M)-exchange operators. As special cases, this also applies to the RELOCATE and SWAP operators. The (N, M)-exchange class evaluates these moves quickly by using C++ templates for various N and M. There are many such `Exchance` library class using here in the same fashion.

```
        class MoveTwoClientsReversed(data: ProblemData)

```

Examines whether it is better to insert U and its successor $n(U)$ after V such that $V \rightarrow n(U) \rightarrow U$, where two customers, U and V.

```
        class TwoOpt(data: ProblemData)

```

Testing with two customers, U and V,

Tests replacing the arcs of U to its successor $n(U)$ and V to $n(V)$ by $U \rightarrow n(V)$ and $V \rightarrow n(U)$ should be performed if U and V are not on the same route.

Replace $U \rightarrow n(U)$ and $V \rightarrow n(V)$ with $U \rightarrow V$ and $n(U) \rightarrow n(V)$ if U and V are on the same route. The route section is now in reverse, going from $n(U)$ to V.

Route Operators

The `add_route_operator()` method can be used to add instances of these operators to the `LocalSearch` object. Every route operator has a `RouteOperator` inheritance. The `pyvrp.search` module provides all of these operators as `ROUTE_OPERATORS` for convenience:

```
        from pyvrp.search import ROUTE_OPERATORS

```

```
        class RouteOperator

```

```
        class RelocateStar(data: ProblemData)

```

Moves between routes U and V in the best possible $(1, 0)$-exchange move.

```
        class SwapRoutes(data: ProblemData)
```

The exchange of two routes U and V's visits is evaluated by this operator.

```
        class SwapStar(data: ProblemData)
```

Investigates SWAP* neighborhood [229]. Customers U and V are explored in the SWAP* neighborhood as free-form re-insertions in the provided routes; that is, the customers are switched between routes, but they are not always put in the same position as the other switched customers. Our SWAP* neighborhood implementation mostly adheres to Algorithm 2 of [229].

Stopping Criteria

The different stopping criteria are contained in the `pyvrp.stop` module. These can be used to halt the search process of the `GeneticAlgorithm` whenever a certain condition is satisfied, such as when a predetermined maximum number of iterations or run-time is reached. The `StoppingCriterion` protocol is implemented by all stopping criteria.

```
        class StoppingCriterion(*args, **kwargs)
```

```
        class MaxIterations(max_iterations: int)
```

Criterion that ends after a predetermined amount of repetitions.

```
        class MaxRuntime(max_runtime: float)
```

Criterion that ends after a predetermined amount of time.

```
        class NoImprovement(max_iterations: int)
```

```
        class TimedNoImprovement(
            max_iterations: int,
            max_runtime: float)
```

A stopping criterion that terminates the process after a predetermined amount of repetitions without improvement or after a set amount of time, whichever comes first.

Script solve.py for HGS

```
from pyvrp import Model, read
from pyvrp.stop import MaxIterations, MaxRuntime

ITERATIONS = 10

def solve_with_hgs(input_path, runtime):
    INSTANCE = read(input_path, instance_format="solomon", round_func="trunc1")
    model = Model.from_data(INSTANCE)
    result = model.solve(stop=MaxRuntime(runtime), seed=0)
    # result = model.solve(stop=MaxIterations(ITERATIONS), seed=0)

    print("HGS cost:", result.cost() / 10)
    print("HGS solution:")
    print(result.best)
    # create list of routes for result.best
    routes = []
    for route in result.best.get_routes():
        routes.append(route.visits())

    return routes, round(result.cost() / 10, 1)
```

A.2 SA Methods Implementation

Script solve.py for SA

```python
from sa.instance_loader import load_from_file
from sa.simulated_annealing import sa_algorithm

INIT_TEMP = 700
UPDATE_TEMP = lambda t: 0.9999 * t
STOP_CRITERIA = lambda t: t <= 0.01

def solve_using_sa(input_path):
    instance = load_from_file(input_path)
    instance.find_initial_solution()

    results = sa_algorithm(instance, INIT_TEMP, UPDATE_TEMP, STOP_CRITERIA)

    routes = results[2][0].get_solution()
    cost = results[2][0].get_total_distance()

    print(f"SA cost: {cost}")
    print("SA solution:")
    for i, route in enumerate(routes, start=1):
        print(f"Route #{i}: {' '.join(str(node) for node in route)}")

    return routes, round(cost, 1)
```

Customer

```python
from util import *
from math import ceil
import random


class Customer:
    def __init__(self, cust_no, x, y, demand, ready_time, due_date, service_time):
        self.cust_no = cust_no
        self.x = x
        self.y = y
        self.demand = demand
        self.ready_time = ready_time
        self.due_date = due_date
        self.service_time = service_time
        self.is_served = False
        self.vehicle_num = None

    def copy(self):
        return Customer(self.cust_no, self.x, self.y, self.demand, self.ready_time, self.due_date,
     self.service_time)

    def served(self, vehicle_num):
        self.is_served = True
        self.vehicle_num = vehicle_num

    def unserve(self):
        self.is_served = False
        self.vehicle_num = None

    def __str__(self):
        return f'Customer NO. : {self.cust_no}; X : {self.x}; Y : {self.y}; Demand : {self.demand
    }; Ready Time : {self.ready_time}; Due Date : {self.due_date}; Service Time : {self.
    service_time}; Vehichle num: {self.vehicle_num}'

    def __eq__(self, other):
        return self.cust_no == other.cust_no
```

Vehicle

```
class Vehicle:
    def __init__(self, id, depo, max_capacity, min_capacity=0):
        self.id = id
        self.x = depo.x
        self.y = depo.y
        self.max_capacity = max_capacity
        self.min_capacity = min_capacity
        self.capacity = max_capacity
        self.last_service_time = 0
        self.service_route = [(depo, 0)]
        self.total_distance = 0
        self.depo = depo

    def serve_customer(self, customer):
        if not (customer.ready_time <= (ceil(distance(customer, self)) + self.last_service_time)
    <= customer.due_date) or self.capacity <= customer.demand:
            return False
        if self.depo.due_date < ceil(distance(customer, self)) + self.last_service_time + customer
    .service_time + distance(customer, self.depo):
            return False
        dist = distance(customer, self)
        self.x = customer.x
        self.y = customer.y
        self.capacity -= customer.demand
        self.last_service_time += ceil(dist)
        self.service_route += [(customer, self.last_service_time)]
        self.last_service_time += customer.service_time
        customer.served(self.id)
        self.total_distance += dist
        if self.capacity < self.min_capacity:
            self.return_home()
        return True
```

```
def serve_customer_force(self, customer):
        if customer.ready_time > ceil(distance(customer, self)) + self.last_service_time:
            last_service_time = self.last_service_time
            self.last_service_time = customer.ready_time - ceil(distance(customer, self))
            if self.serve_customer(customer):
                return True
            else:
                self.last_service_time = last_service_time
        return False

    def return_home(self):
        if self.x != self.depo.x or self.y != self.depo.y:
            self.capacity = self.max_capacity
            self.last_service_time += ceil(distance(self, self.depo))
            self.service_route += [(self.depo, self.last_service_time)]
            self.total_distance += distance(self, self.depo)
            self.x = self.depo.x
            self.y = self.depo.y
```

```
def remove_customer(self, customer):
        customer_idx = [route_node[0] for route_node in self.service_route].index(customer)
        del self.service_route[customer_idx]
        self.capacity += customer.demand
        customer.unserve()

        for i, (curr_customer, curr_time) in enumerate(self.service_route[customer_idx :]):
            prev_customer, prev_time = self.service_route[customer_idx + i - 1]
            new_time = prev_time + prev_customer.service_time + ceil(distance(prev_customer,
    curr_customer))
            new_time = max(new_time, curr_customer.ready_time)
            self.service_route[customer_idx + i] = (curr_customer, new_time)

        next_customer = self.service_route[customer_idx][0]
        prev_customer = self.service_route[customer_idx - 1][0]
```

```
        self.total_distance -= (distance(customer, next_customer) + distance(customer,
    prev_customer))
        self.total_distance += distance(prev_customer, next_customer)
        self.last_service_time = self.service_route[-1][1]

        self.reset_vehicle_used()
```

```
def try_to_serve_customer(self, new_customer):
        if len(self.service_route) == 1:
            return self.serve_customer(new_customer) or self.serve_customer_force(new_customer)
        shuffled = list(range(1, len(self.service_route)))
        random.shuffle(shuffled)
        for i in shuffled:
            vehicle = Vehicle(self.id, self.depo, self.max_capacity, self.min_capacity)
            vehicle.hard_reset_vehicle()
            index = i
            should_use_route = True
            new_customer.is_served = False
            for e, (customer, curr_time) in enumerate(self.service_route[1:]):
                if e + 1 == index:
                    if not vehicle.serve_customer(new_customer):
                        if not vehicle.serve_customer_force(new_customer):
                            should_use_route = False
                            break
                if not vehicle.serve_customer(customer):
                    if not vehicle.serve_customer_force(customer):
                        should_use_route = False
                        break
            if not should_use_route or not new_customer.is_served:
                new_customer.is_served = False
                continue
            self.service_route = vehicle.service_route[:]
            self.last_service_time = vehicle.last_service_time
            self.capacity = vehicle.capacity
            self.total_distance = vehicle.total_distance
            return True
        return False
```

```
def hard_reset_vehicle(self):
        self.service_route = [(self.depo, 0)]
        self.last_service_time = 0
        self.capacity = self.max_capacity
        self.total_distance = 0

    def reset_vehicle_used(self):
        if self.service_route[0] == self.service_route[1]:
            self.hard_reset_vehicle()

    def __str__(self):
        return f'self.id = {self.id}; x={self.x}; y={self.y}; capacity={self.capacity};
    last_service_time={self.last_service_time}; {self.service_route}'
```

```
def all_served(customers, b=False):
    result = True
    for c in customers:
        if not c.is_served:
            if b:
                print(c)
            result = False

    return result
```

Instance

```
class Instance:
    def __init__(self, num_vehicles, capacity, customer_list):
        assert (
```

```
            num_vehicles > 0 and capacity > 0
        ), f'Number of vehicles and their capacity must be positive! {num_vehicles}, {capacity}'
        self.num_vehicles = num_vehicles
        self.capacity = capacity


        depo = customer_list[0]
        self.vehicles = [Vehicle(i, depo, capacity) for i in range(num_vehicles)]
        assert (
            customer_list[0].cust_no == 0 and customer_list[0].ready_time == 0 and customer_list
    [0].demand == 0
        ), f'Customer list must contain depot with customer number 0!'
        self.customer_list = [customer_list[0]] + sorted(customer_list[1:], key=lambda c: c.
    ready_time)

    def __getitem__(self, key):
        return self.customer_list[key]

    def __str__(self):
        result = f'Vehicle Number: {self.num_vehicles}; Capacity: {self.capacity};'
        for customer in self.customer_list:
            result += f'\n{customer}'
        return result

    def sort_by_ready_time(self):
        self.customer_list.sort(key=lambda c: c.ready_time)
```

```
def find_initial_solution(self):
        for i, v in enumerate(self.vehicles):
            while True:
                self.customer_list.sort(key = lambda c: distance(c, v) + c.ready_time)
                found = False
                for customer in self.customer_list:
                    if customer.is_served or customer.cust_no == 0:
                        continue

                    if v.serve_customer(customer):
                        found = True
                        break

                if not found:
                    for customer in self.customer_list:
                        if customer.is_served or customer.cust_no == 0:
                            continue

                        if v.serve_customer_force(customer):
                            found = True
                            break

                if not found:
                    break

            self.customer_list.sort(key = lambda c: c.cust_no)
            if all_served(self.customer_list[1:]):
                break
        self.customer_list.sort(key = lambda c: c.cust_no)

        for vehicle in self.vehicles:
            if vehicle.last_service_time == 0:
                continue
            vehicle.return_home()
        if not all_served(self.customer_list[1:], True):
            print("Not all vehicles has been served!\n")
```

Generate Random Neighbour

```
def generate_random_neighbour(self):
```

```
        #rand_cust = self.customer_list[random.randint(1, len(self.customer_list) - 1)]
        rand_cust = random.choices(self.customer_list[1:], [1./len(self.vehicles[c.vehicle_num].
    service_route) for c in self.customer_list[1:]], k = 1)[0]
        current_serving_vehicle = self.vehicles[rand_cust.vehicle_num]
        current_serving_vehicle.remove_customer(rand_cust)
        v = None
        while not rand_cust.is_served:
            if self.get_neighbour(rand_cust):
                return
            self.get_neighbour(rand_cust, True)

    def get_neighbour(self, customer, force=False):
        shuffled = [i for i in range(0, len(self.vehicles) - 1)]
        random.shuffle(shuffled)
        for vehicle_num in shuffled:
            vehicle = self.vehicles[vehicle_num]
            if force or vehicle.last_service_time != 0:
                if vehicle.try_to_serve_customer(customer):
                    return True
```

Total Distance and Vehicle Calculation

```
def get_output(self):
        dist = 0
        result = ""
        i = 1
        for vehicle in self.vehicles:
            if vehicle.last_service_time == 0:
                continue
            vehicle.return_home()
            dist += vehicle.total_distance
            result += f'{i}: '
            for node in vehicle.service_route:
                result += f'{node[0].cust_no}({node[1]})->'
            result = result[:-2] + '\n'
            i += 1

        print("vehicle count: ", i-1)
        print("distance: ", dist)
        return f'{i-1}\n{result}{dist}\n'
```

```
    def get_total_distance_and_vehicles(self):
        dist = 0
        vehicles_used = 0
        for vehicle in self.vehicles:
            if vehicle.last_service_time == 0:
                continue
            vehicle.return_home()
            vehicles_used += 1
            dist += vehicle.total_distance
        return dist, vehicles_used

    def __str__(self):
        return self.get_output()
```

Load File

```
def load_from_file(filepath):
    i = 0
    customer_list = []
    num_vehicles = 0
    capacity = 0

    with open(filepath) as f:
        line = f.readline()
        while line:
            if not line.strip(): # skip empty lines
```

```
                line = f.readline()
                continue

            if i == 2: # number of vehicles and capacity
                params = [int(p) for p in line.split()]
                num_vehicles = params[0]
                capacity = params[1]
            elif i >= 5: # customer info
                params = [int(p) for p in line.split()]
                customer_list.append(Customer(*params))

            i += 1
            line = f.readline()

    return Instance(num_vehicles, capacity, customer_list)
```

Calculate Distance

```
from math import sqrt

def distance(object1, object2):
    diff_x = object1.x - object2.x
    diff_y = object1.y - object2.y
    return sqrt(diff_x * diff_x + diff_y * diff_y)
```

Simluated Annealing

```
import instance_loader
import random
from copy import deepcopy
from math import exp
import time

counter = 0

def objective_function(num_vhcls, total_distance):
    global counter
    counter += 1
    return num_vhcls * total_distance

def sa_algorithm(instance, temp_start = 350, update_temp = lambda t : 0.9999 * t, stop_criterion =
     lambda t : t <= 0.01):
    curr_solution = incumb_solution = deepcopy(instance)
    # print("Inside sa: ", instance.get_total_distance_and_vehicles())
    curr_dist, curr_vhcls = incumb_dist, incumb_vhcls = curr_solution.
    get_total_distance_and_vehicles()

    temp = temp_start

    start = time.time()
    afterOneMin, cOneMin = None, 0
    afterFiveMin, cFiveMin = None, 0
    total = None
    incumb_vhcls_o = objective_function(incumb_vhcls, incumb_dist)

    while not stop_criterion(temp):
        # print(temp, incumb_vhcls, incumb_dist)
        neighbour = deepcopy(curr_solution)
        neighbour.generate_random_neighbour()

        neighbour_dist, neighbour_vhcls = neighbour.get_total_distance_and_vehicles()

        curr_sol_o = objective_function(curr_vhcls,  curr_dist)
        neigh_sol_o = objective_function(neighbour_vhcls, neighbour_dist)

        if neigh_sol_o < curr_sol_o \
            or random.random() < exp(- (abs(curr_sol_o - neigh_sol_o)) / temp):
            curr_solution = neighbour
```

```
            curr_dist, curr_vhcls = neighbour_dist, neighbour_vhcls

            if (objective_function(curr_vhcls, curr_dist) < incumb_vhcls_o):
                incumb_solution = curr_solution
                incumb_dist, incumb_vhcls = curr_dist, curr_vhcls
                incumb_vhcls_o = objective_function(incumb_vhcls, incumb_dist)

            temp = update_temp(temp)
            if not afterOneMin and time.time() - start >= 60:
            afterOneMin, cOneMin = deepcopy(incumb_solution), counter
            if not afterFiveMin and time.time() - start >= 5 * 60:
            afterFiveMin, cFiveMin = deepcopy(incumb_solution), cFiveMin

    if not afterOneMin:
        afterOneMin = incumb_solution
    if not afterFiveMin:
        afterFiveMin = incumb_solution

    print(time.time() - start)
    return [(afterOneMin, cOneMin), (afterFiveMin, cFiveMin), (incumb_solution, counter)]
```

A.3 ACO Methods Implementation

Script solve.py for ACO

```
from aco.vrptw_base import VrptwGraph
from aco.multiple_ant_colony_system import MultipleAntColonySystem


ants_num = 30
q0 = 0.9
beta = 0.9
rho = 0.1
show_figure = False
runtime_in_minutes = 5


def solve_with_aco(input_path):
    graph = VrptwGraph(input_path, rho)
    macs = MultipleAntColonySystem(
        graph,
        ants_num=ants_num,
        beta=beta,
        q0=q0,
        whether_or_not_to_show_figure=show_figure,
        runtime_in_minutes=runtime_in_minutes,
    )
    macs.run_multiple_ant_colony_system()
    print("ACO cost:", macs.best_path_distance.value)
    routes = get_best_route_from_path(macs.best_path)
    print("ACO solution:")
    for i, route in enumerate(routes, start=1):
        print(f"Route #{i}: {' '.join(str(node) for node in route)}")
    print()

    return routes, round(macs.best_path_distance.value, 1)


def get_best_route_from_path(best_path):
    if not best_path:
        return []
    routes = []
    route = []
    for node in best_path:
```

```
        if node != 0:
            route.append(node)
        else:
            if route:
                routes.append(route)
                route = []
    return routes
```

Script ant.py for ACO

```
import numpy as np
import copy
from aco.vrptw_base import VrptwGraph
from threading import Event

class Ant:
    def __init__(self, graph: VrptwGraph, start_index=0):
        super()
        self.graph = graph
        self.current_index = start_index
        self.vehicle_load = 0
        self.vehicle_travel_time = 0
        self.travel_path = [start_index]
        self.arrival_time = [0]

        self.index_to_visit = list(range(graph.node_num))
        self.index_to_visit.remove(start_index)

        self.total_travel_distance = 0

    def clear(self):
        self.travel_path.clear()
        self.index_to_visit.clear()

    def move_to_next_index(self, next_index):
        # Update ant path
        self.travel_path.append(next_index)
        self.total_travel_distance += self.graph.node_dist_mat[self.current_index][
            next_index
        ]

        dist = self.graph.node_dist_mat[self.current_index][next_index]
        self.arrival_time.append(self.vehicle_travel_time + dist)

        if self.graph.nodes[next_index].is_depot:
            # If the next location is a server point, the vehicle load, etc. must be cleared.
            self.vehicle_load = 0
            self.vehicle_travel_time = 0

        else:
            # Update vehicle load, travel distance, time
            self.vehicle_load += self.graph.nodes[next_index].demand
            # If it is earlier than the time window (ready_time) required by the customer, you
    need to wait

            self.vehicle_travel_time += (
                dist
                + max(
                    self.graph.nodes[next_index].ready_time
                    - self.vehicle_travel_time
                    - dist,
                    0,
                )
                + self.graph.nodes[next_index].service_time
            )
            self.index_to_visit.remove(next_index)

```

```
57          self.current_index = next_index
58
59      def index_to_visit_empty(self):
60          return len(self.index_to_visit) == 0
61
62      def get_active_vehicles_num(self):
63          return self.travel_path.count(0) - 1
64
65      def check_condition(self, next_index) -> bool:
66          """
67          Check whether moving to the next point satisfies the constraints
68          :param next_index:
69          :return:
70          """
71          if (
72              self.vehicle_load + self.graph.nodes[next_index].demand
73              > self.graph.vehicle_capacity
74          ):
75              return False
76
77          dist = self.graph.node_dist_mat[self.current_index][next_index]
78          wait_time = max(
79              self.graph.nodes[next_index].ready_time - self.vehicle_travel_time - dist, 0
80          )
81          service_time = self.graph.nodes[next_index].service_time
82
83          # Check whether you can return to the service shop after visiting a certain passenger
84          if (
85              self.vehicle_travel_time
86              + dist
87              + wait_time
88              + service_time
89              + self.graph.node_dist_mat[next_index][0]
90              > self.graph.nodes[0].due_time
91          ):
92              return False
93
94          # Cannot serve passengers outside due time
95          if self.vehicle_travel_time + dist > self.graph.nodes[next_index].due_time:
96              return False
97
98          return True
99
100     def cal_next_index_meet_constrains(self):
101         """
102         Find all customers reachable from the current location (ant.current_index)
103         :return:
104         """
105         next_index_meet_constrains = []
106         for next_ind in self.index_to_visit:
107             if self.check_condition(next_ind):
108                 next_index_meet_constrains.append(next_ind)
109         return next_index_meet_constrains
110
111     def cal_nearest_next_index(self, next_index_list):
112         """
113         Select the customer closest to the current location (ant.current_index) from the customers
     to be selected.
114
115         :param next_index_list:
116         :return:
117         """
118         current_ind = self.current_index
119
120         nearest_ind = next_index_list[0]
121         min_dist = self.graph.node_dist_mat[current_ind][next_index_list[0]]
122
```

```
        for next_ind in next_index_list[1:]:
            dist = self.graph.node_dist_mat[current_ind][next_ind]
            if dist < min_dist:
                min_dist = dist
                nearest_ind = next_ind

        return nearest_ind

    @staticmethod
    def cal_total_travel_distance(graph: VrptwGraph, travel_path):
        distance = 0
        current_ind = travel_path[0]
        for next_ind in travel_path[1:]:
            distance += graph.node_dist_mat[current_ind][next_ind]
            current_ind = next_ind
        return distance

    def try_insert_on_path(self, node_id, stop_event: Event):
        """
        Try to insert node_id into the current travel_path
        The insertion location cannot violate the restrictions on load, time, and travel distance.
        If there are multiple locations, find the optimal location
        :param node_id:
        :return:
        """
        best_insert_index = None
        best_distance = None

        for insert_index in range(len(self.travel_path)):
            if stop_event.is_set():
                # print('[try_insert_on_path]: receive stop event')
                return

            if self.graph.nodes[self.travel_path[insert_index]].is_depot:
                continue

            # Find the nearest depot in front of insert_index
            front_depot_index = insert_index
            while (
                front_depot_index >= 0
                and not self.graph.nodes[self.travel_path[front_depot_index]].is_depot
            ):
                front_depot_index -= 1
            front_depot_index = max(front_depot_index, 0)

            # check_ant starts from front_depot_index
            check_ant = Ant(self.graph, self.travel_path[front_depot_index])

            # Let check_ant walk through the point in path where the subscript starts from
    front_depot_index and ends at insert_index-1
            for i in range(front_depot_index + 1, insert_index):
                check_ant.move_to_next_index(self.travel_path[i])

            # Start tentatively accessing the nodes in the sorted index_to_visit
            if check_ant.check_condition(node_id):
                check_ant.move_to_next_index(node_id)
            else:
                continue

            # If the node_id can be reached, ensure that the vehicle can travel back to the depot.
            for next_ind in self.travel_path[insert_index:]:
                if stop_event.is_set():
                    # print('[try_insert_on_path]: receive stop event')
                    return

                if check_ant.check_condition(next_ind):
                    check_ant.move_to_next_index(next_ind)
```

```
189
190                     # If you return to the depot
191                     if self.graph.nodes[next_ind].is_depot:
192                         temp_front_index = self.travel_path[insert_index - 1]
193                         temp_back_index = self.travel_path[insert_index]
194
195                         check_ant_distance = (
196                             self.total_travel_distance
197                             - self.graph.node_dist_mat[temp_front_index][
198                                 temp_back_index
199                             ]
200                             + self.graph.node_dist_mat[temp_front_index][node_id]
201                             + self.graph.node_dist_mat[node_id][temp_back_index]
202                         )
203
204                         if best_distance is None or check_ant_distance < best_distance:
205                             best_distance = check_ant_distance
206                             best_insert_index = insert_index
207                         break
208
209                 # If it is not possible to return to the depot, return to the previous level
210                 else:
211                     break
212
213         return best_insert_index
214
215     def insertion_procedure(self, stop_even: Event):
216         """
217         Try to find a suitable location for each unvisited node and insert it into the current
    travel_path
218         The insertion location cannot violate the restrictions on load, time, and travel distance.
219         :return:
220         """
221         if self.index_to_visit_empty():
222             return
223
224         success_to_insert = True
225         # Until none of the unvisited nodes can be inserted successfully
226         while success_to_insert:
227             success_to_insert = False
228             # Get unvisited nodes
229             ind_to_visit = np.array(copy.deepcopy(self.index_to_visit))
230
231             # Get the demand for visiting customer points, sort in descending order
232             demand = np.zeros(len(ind_to_visit))
233             for i, ind in zip(range(len(ind_to_visit)), ind_to_visit):
234                 demand[i] = self.graph.nodes[ind].demand
235
236             arg_ind = np.argsort(demand)[::-1]
237             ind_to_visit = ind_to_visit[arg_ind]
238
239             for node_id in ind_to_visit:
240                 if stop_even.is_set():
241                     # print('[insertion_procedure]: receive stop event')
242                     return
243
244                 best_insert_index = self.try_insert_on_path(node_id, stop_even)
245                 if best_insert_index is not None:
246                     self.travel_path.insert(best_insert_index, node_id)
247                     self.index_to_visit.remove(node_id)
248                     # print('[insertion_procedure]: success to insert %d(node id) in %d(index)' %
    (node_id, best_insert_index))
249                     success_to_insert = True
250
251             del demand
252             del ind_to_visit
253         # if self.index_to_visit_empty():
```

```
        #     print('[insertion_procedure]: success in insertion')

        self.total_travel_distance = Ant.cal_total_travel_distance(
            self.graph, self.travel_path
        )

    @staticmethod
    def local_search_once(
        graph: VrptwGraph,
        travel_path: list,
        travel_distance: float,
        i_start,
        stop_event: Event,
    ):
        # Find the locations of all depots in path
        depot_ind = []
        for ind in range(len(travel_path)):
            if graph.nodes[travel_path[ind]].is_depot:
                depot_ind.append(ind)

        # Divide self.travel_path into multiple segments, each segment starts with depot and ends with depot, called route
        for i in range(i_start, len(depot_ind)):
            for j in range(i + 1, len(depot_ind)):
                if stop_event.is_set():
                    return None, None, None

                for start_a in range(depot_ind[i - 1] + 1, depot_ind[i]):
                    for end_a in range(start_a, min(depot_ind[i], start_a + 6)):
                        for start_b in range(depot_ind[j - 1] + 1, depot_ind[j]):
                            for end_b in range(start_b, min(depot_ind[j], start_b + 6)):
                                if start_a == end_a and start_b == end_b:
                                    continue
                                new_path = []
                                new_path.extend(travel_path[:start_a])
                                new_path.extend(travel_path[start_b : end_b + 1])
                                new_path.extend(travel_path[end_a:start_b])
                                new_path.extend(travel_path[start_a:end_a])
                                new_path.extend(travel_path[end_b + 1 :])

                                depot_before_start_a = depot_ind[i - 1]

                                depot_before_start_b = (
                                    depot_ind[j - 1]
                                    + (end_b - start_b)
                                    - (end_a - start_a)
                                    + 1
                                )
                                if not graph.nodes[
                                    new_path[depot_before_start_b]
                                ].is_depot:
                                    raise RuntimeError("error")

                                # Determine whether the changed route a is feasible
                                success_route_a = False
                                check_ant = Ant(graph, new_path[depot_before_start_a])
                                for ind in new_path[depot_before_start_a + 1 :]:
                                    if check_ant.check_condition(ind):
                                        check_ant.move_to_next_index(ind)
                                        if graph.nodes[ind].is_depot:
                                            success_route_a = True
                                            break
                                    else:
                                        break

                                check_ant.clear()
                                del check_ant
```

```
                                # Determine whether the changed route b is feasible
                                success_route_b = False
                                check_ant = Ant(graph, new_path[depot_before_start_b])
                                for ind in new_path[depot_before_start_b + 1 :]:
                                    if check_ant.check_condition(ind):
                                        check_ant.move_to_next_index(ind)
                                        if graph.nodes[ind].is_depot:
                                            success_route_b = True
                                            break
                                    else:
                                        break
                                check_ant.clear()
                                del check_ant

                                if success_route_a and success_route_b:
                                    new_path_distance = Ant.cal_total_travel_distance(
                                        graph, new_path
                                    )
                                    if new_path_distance < travel_distance:
                                        # print('success to search')

                                        # Delete one of the depots connected together in the path
                                        for temp_ind in range(1, len(new_path)):
                                            if (
                                                graph.nodes[new_path[temp_ind]].is_depot
                                                and graph.nodes[
                                                    new_path[temp_ind - 1]
                                                ].is_depot
                                            ):
                                                new_path.pop(temp_ind)
                                                break
                                        return new_path, new_path_distance, i
                                else:
                                    new_path.clear()

        return None, None, None

    def local_search_procedure(self, stop_event: Event):
        """
        Use cross to perform a local search on the current travel_path that has visited all nodes in the graph.
        :return:
        """
        new_path = copy.deepcopy(self.travel_path)
        new_path_distance = self.total_travel_distance
        times = 100
        count = 0
        i_start = 1
        while count < times:
            temp_path, temp_distance, temp_i = Ant.local_search_once(
                self.graph, new_path, new_path_distance, i_start, stop_event
            )
            if temp_path is not None:
                count += 1

                del new_path, new_path_distance
                new_path = temp_path
                new_path_distance = temp_distance

                # Set i_start
                i_start = (i_start + 1) % (new_path.count(0) - 1)
                i_start = max(i_start, 1)
            else:
                break

        self.travel_path = new_path
```

```
386         self.total_travel_distance = new_path_distance
387         # print('[local_search_procedure]: local search finished')
```

Script multiple_ant_colony_system.py for ACO

```
1  import numpy as np
2  import random
3  from aco.vprtw_aco_figure import VrptwAcoFigure
4  from aco.vrptw_base import VrptwGraph, PathMessage
5  from aco.ant import Ant
6  from threading import Thread, Event
7  from queue import Queue
8  from concurrent.futures import ThreadPoolExecutor
9  import copy
10 import time
11 from multiprocessing import Process, Manager
12 from multiprocessing import Queue as MPQueue
13
14 class MultipleAntColonySystem:
15     def __init__(
16         self,
17         graph: VrptwGraph,
18         ants_num=10,
19         beta=1,
20         q0=0.1,
21         whether_or_not_to_show_figure=True,
22         runtime_in_minutes=5,
23     ):
24         super()
25         # The location and service time information of graph nodes
26         self.graph = graph
27         # ants_num number of ants
28         self.ants_num = ants_num
29         # vehicle_capacity represents the maximum load of each vehicle
30         self.max_load = graph.vehicle_capacity
31         # beta heuristic information importance
32         self.beta = beta
33         # q0 represents the probability of directly selecting the next point with the highest
       probability
34         self.q0 = q0
35         manager = Manager()
36         # self.best_path_distance = None
37         self.best_path_distance = manager.Value("d", 0.0)  # Shared double
38         # best path
39         # self.best_path = None
40         self.best_path = manager.list()  # shared list
41         self.best_vehicle_num = None
42
43         self.whether_or_not_to_show_figure = whether_or_not_to_show_figure
44         self.runtime_in_minutes = runtime_in_minutes
45
46     @staticmethod
47     def stochastic_accept(index_to_visit, transition_prob):
48         """
49         Roulette
50         :param index_to_visit: a list of N index (list or tuple)
51         :param transition_prob:
52         :return: selected index
53         """
54         # calculate N and max fitness value
55         N = len(index_to_visit)
56
57         # normalize
58         sum_tran_prob = np.sum(transition_prob)
59         norm_transition_prob = transition_prob / sum_tran_prob
60
61         # select: O(1)
```

```
        while True:
            # randomly select an individual with uniform probability
            ind = int(N * random.random())
            if random.random() <= norm_transition_prob[ind]:
                return index_to_visit[ind]

    @staticmethod
    def new_active_ant(
        ant: Ant,
        vehicle_num: int,
        local_search: bool,
        IN: np.numarray,
        q0: float,
        beta: int,
        stop_event: Event,
    ):
        """
       Explore the map according to the specified vehicle_num. The vehicle num used cannot be more than the specified number. Both acs_time and acs_vehicle will use this method.
        For acs_time, you need to visit all nodes (the path is feasible), and try to find a path with a shorter travel distance.
        For acs_vehicle, the vehicle num used will be one less than the number of vehicles used by the currently found best path. To use fewer vehicles, try to visit the nodes. If all nodes are visited (the path is feasible), macs will be notified:param ant:
        :param vehicle_num:
        :param local_search:
        :param IN:
        :param q0:
        :param beta:
        :param stop_event:
        :return:
        """
        # print('[new_active_ant]: start, start_index %d' % ant.travel_path[0])

        # In new_active_ant, up to vehicle_num vehicles can be used, that is, it can contain at most vehicle_num+1 depot nodes. Since one starting node is used, only vehicle depots are left.
        unused_depot_count = vehicle_num

        # If there are still unvisited nodes, you can return to the depot
        while not ant.index_to_visit_empty() and unused_depot_count > 0:
            if stop_event.is_set():
                # print('[new_active_ant]: receive stop event')
                return

            # Calculate all next nodes that meet load and other constraints
            next_index_meet_constrains = ant.cal_next_index_meet_constrains()

            # If there is no next node that satisfies the restriction, return to the depot.
            if len(next_index_meet_constrains) == 0:
                ant.move_to_next_index(0)
                unused_depot_count -= 1
                continue

            # Start calculating the next node that meets the constraints and select the probability of each node
            length = len(next_index_meet_constrains)
            ready_time = np.zeros(length)
            due_time = np.zeros(length)

            for i in range(length):
                ready_time[i] = ant.graph.nodes[
                    next_index_meet_constrains[i]
                ].ready_time
                due_time[i] = ant.graph.nodes[next_index_meet_constrains[i]].due_time

            delivery_time = np.maximum(
                ant.vehicle_travel_time
```

```
                + ant.graph.node_dist_mat[ant.current_index][
                    next_index_meet_constrains
                ],
                ready_time,
            )
            delta_time = delivery_time - ant.vehicle_travel_time
            distance = delta_time * (due_time - ant.vehicle_travel_time)

            distance = np.maximum(1.0, distance - IN[next_index_meet_constrains])
            closeness = 1 / distance

            transition_prob = ant.graph.pheromone_mat[ant.current_index][
                next_index_meet_constrains
            ] * np.power(closeness, beta)
            transition_prob = transition_prob / np.sum(transition_prob)

            # Directly select the node with the largest closeness according to probability
            if np.random.rand() < q0:
                max_prob_index = np.argmax(transition_prob)
                next_index = next_index_meet_constrains[max_prob_index]
            else:
                # Use the roulette algorithm
                next_index = MultipleAntColonySystem.stochastic_accept(
                    next_index_meet_constrains, transition_prob
                )

            # Update pheromone matrix
            ant.graph.local_update_pheromone(ant.current_index, next_index)
            ant.move_to_next_index(next_index)

        # If you have visited all the points, you need to return to the depot.
        if ant.index_to_visit_empty():
            ant.graph.local_update_pheromone(ant.current_index, 0)
            ant.move_to_next_index(0)

        # Insert unvisited points to ensure that the path is feasible
        ant.insertion_procedure(stop_event)

        # ant.index_to_visit_empty()==True means feasible
        if local_search is True and ant.index_to_visit_empty():
            ant.local_search_procedure(stop_event)

    @staticmethod
    def acs_time(
        new_graph: VrptwGraph,
        vehicle_num: int,
        ants_num: int,
        q0: float,
        beta: int,
        global_path_queue: Queue,
        path_found_queue: Queue,
        stop_event: Event,
    ):
        """
        For acs_time, you need to visit all nodes (the path is feasible), and try to find a path with a shorter travel distance.
        :param new_graph:
        :param vehicle_num:
        :param ants_num:
        :param q0:
        :param beta:
        :param global_path_queue:
        :param path_found_queue:
        :param stop_event:
        :return:
        """

```

```
        # Up to vehicle_num vehicles can be used, that is, among the depots containing at most vehicle_num+1 in the path, find the shortest path.
        # vehicle_num is set to be consistent with the current best_path
        # print("[acs_time]: start, vehicle_num %d" % vehicle_num)
        # Initialize the pheromone matrix
        global_best_path = None
        global_best_distance = None
        ants_pool = ThreadPoolExecutor(ants_num)
        ants_thread = []
        ants = []
        while True:
            # print("[acs_time]: new iteration")

            if stop_event.is_set():
                # print("[acs_time]: receive stop event")
                return

            for k in range(ants_num):
                ant = Ant(new_graph, 0)
                thread = ants_pool.submit(
                    MultipleAntColonySystem.new_active_ant,
                    ant,
                    vehicle_num,
                    True,
                    np.zeros(new_graph.node_num),
                    q0,
                    beta,
                    stop_event,
                )
                ants_thread.append(thread)
                ants.append(ant)

            # You can use the result method here to wait for the thread to finish running
            for thread in ants_thread:
                thread.result()

            ant_best_travel_distance = None
            ant_best_path = None
            # Determine whether the path found by the ant is feasible and better than the global path
            for ant in ants:
                if stop_event.is_set():
                    # print("[acs_time]: receive stop event")
                    return

                # Get the current best path
                if not global_path_queue.empty():
                    info = global_path_queue.get()
                    while not global_path_queue.empty():
                        info = global_path_queue.get()
                    # print("[acs_time]: receive global path info")
                    (
                        global_best_path,
                        global_best_distance,
                        global_used_vehicle_num,
                    ) = info.get_path_info()

                # The shortest path calculated by path ants
                if ant.index_to_visit_empty() and (
                    ant_best_travel_distance is None
                    or ant.total_travel_distance < ant_best_travel_distance
                ):
                    ant_best_travel_distance = ant.total_travel_distance
                    ant_best_path = ant.travel_path

            # Global updates of pheromones are performed here
            new_graph.global_update_pheromone(global_best_path, global_best_distance)
```

```
254
255             # Send the calculated current best path to macs
256             if (
257                 ant_best_travel_distance is not None
258                 and ant_best_travel_distance < global_best_distance
259             ):
260                 # print(
261                 #     "[acs_time]: ants' local search found a improved feasible path, send path
    info to macs"
262                 # )
263                 path_found_queue.put(
264                     PathMessage(ant_best_path, ant_best_travel_distance)
265                 )
266
267             ants_thread.clear()
268             for ant in ants:
269                 ant.clear()
270                 del ant
271             ants.clear()
272
273     @staticmethod
274     def acs_vehicle(
275         new_graph: VrptwGraph,
276         vehicle_num: int,
277         ants_num: int,
278         q0: float,
279         beta: int,
280         global_path_queue: Queue,
281         path_found_queue: Queue,
282         stop_event: Event,
283     ):
284         """
285         For acs_vehicle, the vehicle num used will be one less than the number of vehicles used by
     the currently found best path. To use fewer vehicles, try to visit the nodes. If all nodes
    are visited (the path is feasible), macs will be notified
286         :param new_graph:
287         :param vehicle_num:
288         :param ants_num:
289         :param q0:
290         :param beta:
291         :param global_path_queue:
292         :param path_found_queue:
293         :param stop_event:
294         :return:
295         """
296         # vehicle_num is set to one less than the current best_path
297         # print("[acs_vehicle]: start, vehicle_num %d" % vehicle_num)
298         global_best_path = None
299         global_best_distance = None
300
301         # Initialize path and distance using nearest_neighbor_heuristic algorithm
302         current_path, current_path_distance, _ = new_graph.nearest_neighbor_heuristic(
303             max_vehicle_num=vehicle_num
304         )
305
306         # Find unvisited nodes in the current path
307         current_index_to_visit = list(range(new_graph.node_num))
308         for ind in set(current_path):
309             current_index_to_visit.remove(ind)
310
311         ants_pool = ThreadPoolExecutor(ants_num)
312         ants_thread = []
313         ants = []
314         IN = np.zeros(new_graph.node_num)
315         while True:
316             # print("[acs_vehicle]: new iteration")
317
```

```
            if stop_event.is_set():
                # print("[acs_vehicle]: receive stop event")
                return

            for k in range(ants_num):
                ant = Ant(new_graph, 0)
                thread = ants_pool.submit(
                    MultipleAntColonySystem.new_active_ant,
                    ant,
                    vehicle_num,
                    False,
                    IN,
                    q0,
                    beta,
                    stop_event,
                )

                ants_thread.append(thread)
                ants.append(ant)

            # You can use the result method here to wait for the thread to finish running
            for thread in ants_thread:
                thread.result()

            for ant in ants:
                if stop_event.is_set():
                    # print("[acs_vehicle]: receive stop event")
                    return

                IN[ant.index_to_visit] = IN[ant.index_to_visit] + 1

                # The path found by the ant is compared with the current_path to see whether
    vehicle_num vehicles can be used to access more nodes.
                if len(ant.index_to_visit) < len(current_index_to_visit):
                    current_path = copy.deepcopy(ant.travel_path)
                    current_index_to_visit = copy.deepcopy(ant.index_to_visit)
                    current_path_distance = ant.total_travel_distance
                    # and set IN to 0
                    IN = np.zeros(new_graph.node_num)

                    # If this path is feasible, it must be sent to macs_vrptw.
                    if ant.index_to_visit_empty():
                        # print(
                        #     "[acs_vehicle]: found a feasible path, send path info to macs"
                        # )
                        path_found_queue.put(
                            PathMessage(ant.travel_path, ant.total_travel_distance)
                        )

            # Update the pheromone in new_graph, global
            new_graph.global_update_pheromone(current_path, current_path_distance)

            if not global_path_queue.empty():
                info = global_path_queue.get()
                while not global_path_queue.empty():
                    info = global_path_queue.get()
                # print("[acs_vehicle]: receive global path info")
                (
                    global_best_path,
                    global_best_distance,
                    global_used_vehicle_num,
                ) = info.get_path_info()

          new_graph.global_update_pheromone(global_best_path, global_best_distance)

            ants_thread.clear()
            for ant in ants:
```

```
                ant.clear()
                del ant
            ants.clear()

    def run_multiple_ant_colony_system(self, file_to_write_path=None):
        """
        Start another thread to run multiple_ant_colony_system, and use the main thread to draw
        :return:
        """
        path_queue_for_figure = MPQueue()
        multiple_ant_colony_system_thread = Process(
            target=self._multiple_ant_colony_system,
            args=(
                path_queue_for_figure,
                file_to_write_path,
            ),
        )
        multiple_ant_colony_system_thread.start()

        # Whether to display figure
        if self.whether_or_not_to_show_figure:
            figure = VrptwAcoFigure(self.graph.nodes, path_queue_for_figure)
            figure.run()
        multiple_ant_colony_system_thread.join()
        # print("Finished:", self.best_path)
        # print("Finished distance:", self.best_path_distance)
        # print("Finished from manager:", self.best_path[:])

    def _multiple_ant_colony_system(
        self, path_queue_for_figure: MPQueue, file_to_write_path=None
    ):
        """
        Call acs_time and acs_vehicle to explore paths
        :param path_queue_for_figure:
        :return:
        """
        if file_to_write_path is not None:
            file_to_write = open(file_to_write_path, "w")
        else:
            file_to_write = None

        start_time_total = time.time()

        # Two queues are needed here, time_what_to_do and vehicle_what_to_do, to tell the two
    threads acs_time and acs_vehicle what the current best path is, or to stop them from
    calculating.
        global_path_to_acs_time = Queue()
        global_path_to_acs_vehicle = Queue()

        # Another queue, path_found_queue, is a feasible path calculated by receiving acs_time and
     acs_vehicle that is better than the best path.
        path_found_queue = Queue()

        # Initialize using nearest neighbor algorithm
        (
            self.best_path[:],
            self.best_path_distance.value,
            self.best_vehicle_num,
        ) = self.graph.nearest_neighbor_heuristic()
        path_queue_for_figure.put(
            PathMessage(self.best_path, self.best_path_distance.value)
        )

        while True:
            # print("[multiple_ant_colony_system]: new iteration")
            start_time_found_improved_solution = time.time()

```

```
448             # The current best path information is placed in the queue to inform acs_time and
    acs_vehicle what the current best_path is.
449             global_path_to_acs_vehicle.put(
450                 PathMessage(self.best_path, self.best_path_distance.value)
451             )
452             global_path_to_acs_time.put(
453                 PathMessage(self.best_path, self.best_path_distance.value)
454             )
455 
456             stop_event = Event()
457 
458             # acs_vehicle, try to explore with self.best_vehicle_num-1 vehicles and visit more
    nodes
459             graph_for_acs_vehicle = self.graph.copy(self.graph.init_pheromone_val)
460             acs_vehicle_thread = Thread(
461                 target=MultipleAntColonySystem.acs_vehicle,
462                 args=(
463                     graph_for_acs_vehicle,
464                     self.best_vehicle_num - 1,
465                     self.ants_num,
466                     self.q0,
467                     self.beta,
468                     global_path_to_acs_vehicle,
469                     path_found_queue,
470                     stop_event,
471                 ),
472             )
473 
474             # acs_time tries to explore with self.best_vehicle_num vehicles to find a shorter path
475             graph_for_acs_time = self.graph.copy(self.graph.init_pheromone_val)
476             acs_time_thread = Thread(
477                 target=MultipleAntColonySystem.acs_time,
478                 args=(
479                     graph_for_acs_time,
480                     self.best_vehicle_num,
481                     self.ants_num,
482                     self.q0,
483                     self.beta,
484                     global_path_to_acs_time,
485                     path_found_queue,
486                     stop_event,
487                 ),
488             )
489 
490             # Start acs_vehicle_thread and acs_time_thread. When they find a feasible and better
    path than the best path, they will be sent to macs.
491             # print("[macs]: start acs_vehicle and acs_time")
492             acs_vehicle_thread.start()
493             acs_time_thread.start()
494 
495             best_vehicle_num = self.best_vehicle_num
496 
497             while acs_vehicle_thread.is_alive() and acs_time_thread.is_alive():
498                 # If no better results are found within the specified time, exit the program
499                 # given_time = 5
500                 if (
501                     time.time() - start_time_found_improved_solution
502                     > 60 * self.runtime_in_minutes
503                 ):
504                     stop_event.set()
505                     # self.print_and_write_in_file(file_to_write, "*" * 50)
506                     # self.print_and_write_in_file(
507                     #     file_to_write,
508                     #     "time is up: cannot find a better solution in given time(%d minutes)"
509                     #     % self.runtime_in_minutes,
510                     # )
511                     # self.print_and_write_in_file(
```

```
                    #     file_to_write,
                    #     "it takes %0.3f second from multiple_ant_colony_system running"
                    #     % (time.time() - start_time_total),
                    # )
                    # self.print_and_write_in_file(
                    #     file_to_write, "the best path have found is:"
                    # )
                    # self.print_and_write_in_file(file_to_write, self.best_path)
                    # self.print_and_write_in_file(
                    #     file_to_write,
                    #     "best path distance is %f, best vehicle_num is %d"
                    #     % (self.best_path_distance.value, self.best_vehicle_num),
                    # )
                    # self.print_and_write_in_file(file_to_write, "*" * 50)

                    # Pass in None as the end flag
                    if self.whether_or_not_to_show_figure:
                        path_queue_for_figure.put(PathMessage(None, None))

                    if file_to_write is not None:
                        file_to_write.flush()
                        file_to_write.close()
                    return

                if path_found_queue.empty():
                    continue

                path_info = path_found_queue.get()
                # print("[macs]: receive found path info")
                (
                    found_path,
                    found_path_distance,
                    found_path_used_vehicle_num,
                ) = path_info.get_path_info()
                while not path_found_queue.empty():
                    path, distance, vehicle_num = path_found_queue.get().get_path_info()

                    if distance < found_path_distance:
                        found_path, found_path_distance, found_path_used_vehicle_num = (
                            path,
                            distance,
                            vehicle_num,
                        )

                    if vehicle_num < found_path_used_vehicle_num:
                        found_path, found_path_distance, found_path_used_vehicle_num = (
                            path,
                            distance,
                            vehicle_num,
                        )

                # If the distance of the found path (which is feasible) is shorter, update the current best path information
                if found_path_distance < self.best_path_distance.value:
                    # Search for better results, update start_time
                    start_time_found_improved_solution = time.time()

                    # self.print_and_write_in_file(file_to_write, "*" * 50)
                    # self.print_and_write_in_file(
                    #     file_to_write,
                    #     "[macs]: distance of found path (%f) better than best path's (%f)"
                    #     % (found_path_distance, self.best_path_distance.value),
                    # )
                    # self.print_and_write_in_file(
                    #     file_to_write,
                    #     "it takes %0.3f second from multiple_ant_colony_system running"
                    #     % (time.time() - start_time_total),
```

```
578                     # )
579                     # self.print_and_write_in_file(file_to_write, "*" * 50)
580                     if file_to_write is not None:
581                         file_to_write.flush()
582
583                     self.best_path[:] = found_path
584                     # print("best1: ", self.best_path)
585                     # print("found1: ", found_path)
586                     self.best_vehicle_num = found_path_used_vehicle_num
587                     self.best_path_distance.value = found_path_distance
588
589                     # If graphics need to be drawn, the best path to be found is sent to the
    drawing program
590                     if self.whether_or_not_to_show_figure:
591                         path_queue_for_figure.put(
592                             PathMessage(self.best_path, self.best_path_distance.value)
593                         )
594
595                     # Notify acs_vehicle and acs_time threads of the currently found best_path and
     best_path_distance
596                     global_path_to_acs_vehicle.put(
597                         PathMessage(self.best_path, self.best_path_distance.value)
598                     )
599                     global_path_to_acs_time.put(
600                         PathMessage(self.best_path, self.best_path_distance.value)
601                     )
602
603                 # If the path found by these two threads uses fewer vehicles, stop these two
    threads and start the next iteration.
604                 # Send stop information to acs_time and acs_vehicle
605                 if found_path_used_vehicle_num < best_vehicle_num:
606                     # Search for better results, update start_time
607                     start_time_found_improved_solution = time.time()
608                     # self.print_and_write_in_file(file_to_write, "*" * 50)
609                     # self.print_and_write_in_file(
610                     #     file_to_write,
611                     #     "[macs]: vehicle num of found path (%d) better than best path's (%d),
    found path distance is %f"
612                     #     % (
613                     #         found_path_used_vehicle_num,
614                     #         best_vehicle_num,
615                     #         found_path_distance,
616                     #     ),
617                     # )
618                     # self.print_and_write_in_file(
619                     #     file_to_write,
620                     #     "it takes %0.3f second multiple_ant_colony_system running"
621                     #     % (time.time() - start_time_total),
622                     # )
623                     # self.print_and_write_in_file(file_to_write, "*" * 50)
624                     if file_to_write is not None:
625                         file_to_write.flush()
626
627                     self.best_path[:] = found_path
628                     # print("best2: ", self.best_path)
629                     # print("found2: ", found_path)
630                     self.best_vehicle_num = found_path_used_vehicle_num
631                     self.best_path_distance.value = found_path_distance
632
633                     if self.whether_or_not_to_show_figure:
634                         path_queue_for_figure.put(
635                             PathMessage(self.best_path, self.best_path_distance.value)
636                         )
637
638                     # Stop the acs_time and acs_vehicle threads
639                     # print("[macs]: send stop info to acs_time and acs_vehicle")
640                     # Notify acs_vehicle and acs_time threads of the currently found best_path and
```

```
     best_path_distance
                    stop_event.set()

    def get_best_route(self):
        if not self.best_path:
            return []
        routes = []
        route = []
        for node in self.best_path:
            if node != 0:
                route.append(node)
            else:
                if route:
                    routes.append(route)
                    route = []
        return routes

    @staticmethod
    def print_and_write_in_file(file_to_write=None, message="default message"):
        if file_to_write is None:
            print(message)
        else:
            print(message)
            file_to_write.write(str(message) + "\n")
```

Script vrptw_base.py for ACO

```
import numpy as np
import copy

class Node:
    def __init__(
        self,
        id: int,
        x: float,
        y: float,
        demand: float,
        ready_time: float,
        due_time: float,
        service_time: float,
    ):
        super()
        self.id = id

        if id == 0:
            self.is_depot = True
        else:
            self.is_depot = False

        self.x = x
        self.y = y
        self.demand = demand
        self.ready_time = ready_time
        self.due_time = due_time
        self.service_time = service_time

class VrptwGraph:
    def __init__(self, file_path, rho=0.1):
        super()
        # node_num node number
        # node_dist_mat distance between nodes (matrix)
        # pheromone_mat Information density on the path between nodes
        (
            self.node_num,
            self.nodes,
            self.node_dist_mat,
            self.vehicle_num,
```

```
            self.vehicle_capacity,
        ) = self.create_from_file(file_path)
        # rho pheromone evaporation rate
        self.rho = rho
        # Create a pheromone matrix

        (
            self.nnh_travel_path,
            self.init_pheromone_val,
            _,
        ) = self.nearest_neighbor_heuristic()
        self.init_pheromone_val = 1 / (self.init_pheromone_val * self.node_num)

        self.pheromone_mat = (
            np.ones((self.node_num, self.node_num)) * self.init_pheromone_val
        )
        # heuristic information matrix
        self.heuristic_info_mat = 1 / self.node_dist_mat

    def copy(self, init_pheromone_val):
        new_graph = copy.deepcopy(self)

        # Pheromones
        new_graph.init_pheromone_val = init_pheromone_val
        new_graph.pheromone_mat = (
            np.ones((new_graph.node_num, new_graph.node_num)) * init_pheromone_val
        )

        return new_graph

    def create_from_file(self, file_path):
        # Read the location of service points and customers from the file
        node_list = []
        with open(file_path, "rt") as f:
            count = 1
            for line in f:
                if count == 5:
                    vehicle_num, vehicle_capacity = line.split()
                    vehicle_num = int(vehicle_num)
                    vehicle_capacity = int(vehicle_capacity)
                elif count >= 10:
                    node_list.append(line.split())
                count += 1
        node_num = len(node_list)
        nodes = list(
            Node(
                int(item[0]),
                float(item[1]),
                float(item[2]),
                float(item[3]),
                float(item[4]),
                float(item[5]),
                float(item[6]),
            )
            for item in node_list
        )

        # Create distance matrix
        node_dist_mat = np.zeros((node_num, node_num))
        for i in range(node_num):
            node_a = nodes[i]
            node_dist_mat[i][i] = 1e-8
            for j in range(i + 1, node_num):
                node_b = nodes[j]
                node_dist_mat[i][j] = VrptwGraph.calculate_dist(node_a, node_b)
                node_dist_mat[j][i] = node_dist_mat[i][j]

```

```
        return node_num, nodes, node_dist_mat, vehicle_num, vehicle_capacity

    @staticmethod
    def calculate_dist(node_a, node_b):
        return np.linalg.norm((node_a.x - node_b.x, node_a.y - node_b.y))

    def local_update_pheromone(self, start_ind, end_ind):
        self.pheromone_mat[start_ind][end_ind] = (1 - self.rho) * self.pheromone_mat[
            start_ind
        ][end_ind] + self.rho * self.init_pheromone_val

    def global_update_pheromone(self, best_path, best_path_distance):
        """
        Update pheromone matrix
        :return:
        """
        self.pheromone_mat = (1 - self.rho) * self.pheromone_mat

        current_ind = best_path[0]
        for next_ind in best_path[1:]:
            self.pheromone_mat[current_ind][next_ind] += self.rho / best_path_distance
            current_ind = next_ind

    def nearest_neighbor_heuristic(self, max_vehicle_num=None):
        index_to_visit = list(range(1, self.node_num))
        current_index = 0
        current_load = 0
        current_time = 0
        travel_distance = 0
        travel_path = [0]

        if max_vehicle_num is None:
            max_vehicle_num = self.node_num

        while len(index_to_visit) > 0 and max_vehicle_num > 0:
            nearest_next_index = self._cal_nearest_next_index(
                index_to_visit, current_index, current_load, current_time
            )

            if nearest_next_index is None:
                travel_distance += self.node_dist_mat[current_index][0]

                current_load = 0
                current_time = 0
                travel_path.append(0)
                current_index = 0

                max_vehicle_num -= 1
            else:
                current_load += self.nodes[nearest_next_index].demand

                dist = self.node_dist_mat[current_index][nearest_next_index]
                wait_time = max(
                    self.nodes[nearest_next_index].ready_time - current_time - dist, 0
                )
                service_time = self.nodes[nearest_next_index].service_time

                current_time += dist + wait_time + service_time
                index_to_visit.remove(nearest_next_index)

                travel_distance += self.node_dist_mat[current_index][nearest_next_index]
                travel_path.append(nearest_next_index)
                current_index = nearest_next_index
        # Finally return to the depot
        travel_distance += self.node_dist_mat[current_index][0]
        travel_path.append(0)

```

```
        vehicle_num = travel_path.count(0) - 1
        return travel_path, travel_distance, vehicle_num

    def _cal_nearest_next_index(
        self, index_to_visit, current_index, current_load, current_time
    ):
        """
        Find the nearest reachable next_index
        :param index_to_visit:
        :return:
        """
        nearest_ind = None
        nearest_distance = None

        for next_index in index_to_visit:
            if current_load + self.nodes[next_index].demand > self.vehicle_capacity:
                continue

            dist = self.node_dist_mat[current_index][next_index]
            wait_time = max(self.nodes[next_index].ready_time - current_time - dist, 0)
            service_time = self.nodes[next_index].service_time
            # Check whether you can return to the service shop after visiting a certain passenger
            if (
                current_time
                + dist
                + wait_time
                + service_time
                + self.node_dist_mat[next_index][0]
                > self.nodes[0].due_time
            ):
                continue

            # Cannot serve passengers outside due time
            if current_time + dist > self.nodes[next_index].due_time:
                continue

            if (
                nearest_distance is None
                or self.node_dist_mat[current_index][next_index] < nearest_distance
            ):
                nearest_distance = self.node_dist_mat[current_index][next_index]
                nearest_ind = next_index

        return nearest_ind

class PathMessage:
    def __init__(self, path, distance):
        if path is not None:
            self.path = copy.deepcopy(path)
            self.distance = copy.deepcopy(distance)
            self.used_vehicle_num = self.path.count(0) - 1
        else:
            self.path = None
            self.distance = None
            self.used_vehicle_num = None

    def get_path_info(self):
        return self.path, self.distance, self.used_vehicle_num
```

Script vrptw_aco_figure.py for ACO

```
import matplotlib.pyplot as plt
from multiprocessing import Queue as MPQueue

class VrptwAcoFigure:
    def __init__(self, nodes: list, path_queue: MPQueue):
        """
```

```
        matplotlib drawing calculations need to be placed on the main thread. It is recommended to open another thread for path finding work.
        When the path-finding thread finds a new path, it puts the path in path_queue, and the graphics drawing thread will automatically draw it.
        The path stored in the queue exists in the form of PathMessage (class)
        The nodes stored in nodes exist in the form of Node (class). Node.x and Node.y are mainly used to obtain the coordinates of the nodes.

        :param nodes: nodes is a list of each node, including depot
        :param path_queue: The queue is used to store the path calculated by the working thread. Each element in the queue is a path, and the path stores the ID of each node.
        """
        self.nodes = nodes
        self.figure = plt.figure(figsize=(10, 10))
        self.figure_ax = self.figure.add_subplot(1, 1, 1)
        self.path_queue = path_queue
        self._depot_color = "k"
        self._customer_color = "steelblue"
        self._line_color = "darksalmon"

    def _draw_point(self):
        # draw depot
        self.figure_ax.scatter(
            [self.nodes[0].x],
            [self.nodes[0].y],
            c=self._depot_color,
            label="depot",
            s=40,
        )

        # draw customer
        self.figure_ax.scatter(
            list(node.x for node in self.nodes[1:]),
            list(node.y for node in self.nodes[1:]),
            c=self._customer_color,
            label="customer",
            s=20,
        )
        # plt.pause(0.5)

    def run(self):
        # First draw each node
        self._draw_point()
        # self.figure.show()

        # Read the new path from the queue and draw it
        # while True:
        #     if not self.path_queue.empty():
        #         # Take the latest path in the queue and discard the other paths.
        #         info = self.path_queue.get()
        #         while not self.path_queue.empty():
        #             info = self.path_queue.get()

        #         path, distance, used_vehicle_num = info.get_path_info()
        #         if path is None:
        #             print("[draw figure]: exit")
        #             break

        #         # You need to record the line to be removed first. You cannot remove it directly in the first loop.
        #         # Otherwise, self.figure_ax.lines will change during the loop, causing some lines to fail to be removed successfully.
        #         remove_obj = []
        #         for line in self.figure_ax.lines:
        #             if line._label == "line":
        #                 remove_obj.append(line)

```

```
        #          for line in remove_obj:
        #              self.figure_ax.lines.remove(line)
        #          remove_obj.clear()

        #          # redraw line
        #          self.figure_ax.set_title(
        #              "travel distance: %0.2f, number of vehicles: %d "
        #              % (distance, used_vehicle_num)
        #          )
        #          self._draw_line(path)
        #      plt.pause(1)

    def _draw_line(self, path):
        # Draw the path according to the index in the path
        for i in range(1, len(path)):
            x_list = [self.nodes[path[i - 1]].x, self.nodes[path[i]].x]
            y_list = [self.nodes[path[i - 1]].y, self.nodes[path[i]].y]
            self.figure_ax.plot(
                x_list, y_list, color=self._line_color, linewidth=1.5, label="line"
            )
            plt.pause(0.2)
```

A.4 GLS (OR-Tools) Methods Implementation

Script solve.py for GLS

```
from gls.base_solver import Solver
from gls.instance_loader import load_instance

time_precision_scaler = 10

def solve_with_gls(input_path, runtime):
    settings = {}
    settings["time_limit"] = runtime

    data = load_instance(input_path, time_precision_scaler)
    solver = Solver(data, time_precision_scaler)
    solver.create_model()
    solver.solve_model(settings)
    routes = solver.get_solution()
    print("GLS cost:", f"{solver.get_solution_travel_time():.1f}")
    print("GLS solution:")
    for i, route in enumerate(routes, start=1):
        print(f"Route #{i}: {' '.join(str(node) for node in route)}")
    print()

    return routes, round(solver.get_solution_travel_time(), 1)
```

Script base_solver.py for GLS

```
from ortools.constraint_solver import pywrapcp, routing_enums_pb2

from gls.data_model import ProblemInstance
from gls.solver_model import SolverSetting

class Solver:
    """
    Solver object that takes a problem instance as input, creates and solves a capacitated vehicle
     routing problem with time
    windows. Objective of the optimization are hierarchical: 1) Minimize number of vehicles 2)
    Minimize total distance.
    Distance is Euclidean, and the value of travel time is equal to the value of distance between
    two nodes.

```

```
    Parameters
    ----------
    data : ProblemInstance
        Problem data according to ProblemInstance model.
    time_precision_scaler : int
        Variable defining the precision of travel and service times, e.g. 100 means precision of
    two decimals.
    """

    def __init__(self, data: ProblemInstance, time_precision_scaler: int):
        self.data = data
        self.time_precision_scaler = time_precision_scaler
        self.manager = None
        self.routing = None
        self.solution = None

    def create_model(self):
        """
        Create vehicle routing model for Solomon instance.
        """
        # Create the routing index manager, i.e. number of nodes, vehicles and depot
        self.manager = pywrapcp.RoutingIndexManager(
            len(self.data["time_matrix"]), self.data["num_vehicles"], self.data["depot"]
        )

        # Create routing model
        self.routing = pywrapcp.RoutingModel(self.manager)

        # Create and register a transit callback
        def time_callback(from_index, to_index):
            """Returns the travel time between the two nodes."""
            # Convert from solver internal routing variable Index to time matrix NodeIndex.
            from_node = self.manager.IndexToNode(from_index)
            to_node = self.manager.IndexToNode(to_index)
            return self.data["time_matrix"][from_node][to_node]

        transit_callback_index = self.routing.RegisterTransitCallback(time_callback)

        # Define cost of each arc and fixed vehicle cost
        self.routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
        # Make sure to first minimize number of vehicles
        self.routing.SetFixedCostOfAllVehicles(100000)

        # Create and register demand callback
        def demand_callback(from_index):
            """Returns the demand of the node."""
            # Convert from routing variable Index to demands NodeIndex.
            from_node = self.manager.IndexToNode(from_index)
            return self.data["demands"][from_node]

        demand_callback_index = self.routing.RegisterUnaryTransitCallback(
            demand_callback
        )

        # Register vehicle capacitites
        self.routing.AddDimensionWithVehicleCapacity(
            demand_callback_index,
            0,  # null capacity slack
            self.data["vehicle_capacities"],  # vehicle maximum capacities
            True,  # start cumul to zero
            "Capacity",
        )

        # Add Time Windows constraint.
        self.routing.AddDimension(
            transit_callback_index,
            10 ** 10,  # allow waiting time at nodes
```

```
            10 ** 10,  # maximum time per vehicle route
            False,  # Don't force start cumul to zero, i.e. vehicles can start after time 0 from depot
            "Time",
        )

         # Allow to drop nodes.
        # penalty = 1000
        # for node in range(1, len(self.data["distance_matrix"])):
        #     self.routing.AddDisjunction([self.manager.NodeToIndex(node)], penalty)
        # end penalty

        time_dimension = self.routing.GetDimensionOrDie("Time")

        # Add time window constraints for each location except depot.
        for location_idx, time_window in enumerate(self.data["time_windows"]):
            if location_idx == self.data["depot"]:
                continue
            index = self.manager.NodeToIndex(location_idx)
            time_dimension.CumulVar(index).SetRange(time_window[0], time_window[1])

        # Add time window constraints for each vehicle start node.
        depot_idx = self.data["depot"]
        for vehicle_id in range(self.data["num_vehicles"]):
            index = self.routing.Start(vehicle_id)
            time_dimension.CumulVar(index).SetRange(
                self.data["time_windows"][depot_idx][0],
                self.data["time_windows"][depot_idx][1],
            )
        # The solution finalizer is called each time a solution is found during search
        # and tries to optimize (min/max) variables values
        for i in range(self.data["num_vehicles"]):
            self.routing.AddVariableMinimizedByFinalizer(
                time_dimension.CumulVar(self.routing.Start(i))
            )
            self.routing.AddVariableMinimizedByFinalizer(
                time_dimension.CumulVar(self.routing.End(i))
            )

    def solve_model(self, settings: SolverSetting):
        """
        Solver model with solver settings.

        Parameters
        ----------
        settings : SolverSetting
            Solver settings according to SolverSetting model.
        """

        # Setting first solution heuristic.
        search_parameters = pywrapcp.DefaultRoutingSearchParameters()
        search_parameters.first_solution_strategy = (
            routing_enums_pb2.FirstSolutionStrategy.PARALLEL_CHEAPEST_INSERTION
        )
        search_parameters.local_search_metaheuristic = (
            routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
        )
        search_parameters.time_limit.seconds = settings["time_limit"]

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

    def print_solution(self):
        """
        Print solution to console.
        """
        print(f"Solution status: {self.routing.status()}\n")
```

```
        if self.routing.status() == 1:
            print(
                f"Objective: {self.solution.ObjectiveValue()/self.time_precision_scaler}\n"
            )
            time_dimension = self.routing.GetDimensionOrDie("Time")
            cap_dimension = self.routing.GetDimensionOrDie("Capacity")
            total_time = 0
            total_vehicles = 0
            for vehicle_id in range(self.data["num_vehicles"]):
                index = self.routing.Start(vehicle_id)
                plan_output = f"Route for vehicle {vehicle_id}:\n"
                while not self.routing.IsEnd(index):
                    time_var = time_dimension.CumulVar(index)
                    cap_var = cap_dimension.CumulVar(index)
                    plan_output += f"{self.manager.IndexToNode(index)} -> "
                    index = self.solution.Value(self.routing.NextVar(index))
                time_var = time_dimension.CumulVar(index)
                plan_output += f"{self.manager.IndexToNode(index)}\n"
                plan_output += f"Time of the route: {self.solution.Min(time_var)/self.
    time_precision_scaler}min\n"
                plan_output += f"Load of vehicle: {self.solution.Min(cap_var)}\n"
                print(plan_output)
                total_time += self.solution.Min(time_var) / self.time_precision_scaler
                if self.solution.Min(time_var) > 0:
                    total_vehicles += 1
            total_travel_time = (
                total_time
                - sum(self.data["service_times"]) / self.time_precision_scaler
            )
            print(f"Total time of all routes: {total_time}min")
            print(f"Total travel time of all routes: {total_travel_time}min")
            print(f"Total vehicles used: {total_vehicles}")

    def get_solution(self):
        """
        Get solution as list of lists of nodes.
        Skip empty routes.
        """
        routes = []
        if self.routing.status() == 1:
            time_dimension = self.routing.GetDimensionOrDie("Time")
            for vehicle_id in range(self.data["num_vehicles"]):
                index = self.routing.Start(vehicle_id)
                route = []
                while not self.routing.IsEnd(index):
                    index = self.solution.Value(self.routing.NextVar(index))
                    node = self.manager.IndexToNode(index)
                    if node != self.data["depot"]:
                        route.append(node)
                time_var = time_dimension.CumulVar(index)
                if self.solution.Min(time_var) > 0:
                    routes.append(route)
        return routes

    def get_solution_time(self):
        """
        Get solution time value.
        """
        if self.routing.status() == 1:
            time_dimension = self.routing.GetDimensionOrDie("Time")
            total_time = 0
            for vehicle_id in range(self.data["num_vehicles"]):
                index = self.routing.Start(vehicle_id)
                while not self.routing.IsEnd(index):
                    time_var = time_dimension.CumulVar(index)
                    index = self.solution.Value(self.routing.NextVar(index))
                time_var = time_dimension.CumulVar(index)
```

```
                total_time += self.solution.Min(time_var) / self.time_precision_scaler
            return total_time
        else:
            return None

    # get total travel time
    def get_solution_travel_time(self):
        """
        Get solution travel time value.
        """
        if self.routing.status() == 1:
            time_dimension = self.routing.GetDimensionOrDie("Time")
            total_travel_time = 0
            for vehicle_id in range(self.data["num_vehicles"]):
                index = self.routing.Start(vehicle_id)
                while not self.routing.IsEnd(index):
                    time_var = time_dimension.CumulVar(index)
                    index = self.solution.Value(self.routing.NextVar(index))
                time_var = time_dimension.CumulVar(index)
                total_travel_time += self.solution.Min(time_var) / self.time_precision_scaler
            total_travel_time = total_travel_time - sum(self.data["service_times"]) / self.
    time_precision_scaler
            return total_travel_time
        else:
            return None
```

Script data_model.py for GLS

```
from pydantic import BaseModel

class ProblemInstance(BaseModel):
    time_matrix: list
    time_windows: list
    demands: list
    depot: int
    num_vehicles: int
    vehicle_capacities: list
    service_times: list
```

Script instance_loader.py for GLS

```
import math
import re

import pandas as pd

from gls.data_model import ProblemInstance

def load_instance(problem_path: str, time_precision_scaler: int) -> ProblemInstance:
    """
    Load instance of Solomon benchmark with defined precision scaler.

    Parameters
    ----------
    time_precision_scaler : int
        Variable defining the precision of travel and service times, e.g. 100 means precision of
    two decimals.
    """

    data = {}
    data["depot"] = 0
    df = pd.read_csv(
        problem_path,
        sep="\s+",
        skiprows=8,
        names=[
            "customer",
            "xcord",
```

```
            "ycord",
            "demand",
            "ready_time",
            "due_date",
            "service_time",
        ],
    )
    df["service_time"] = df["service_time"] * time_precision_scaler
    df["ready_time"] = df["ready_time"] * time_precision_scaler
    df["due_date"] = df["due_date"] * time_precision_scaler

    data["service_times"] = list(df.service_time)

    travel_times = df[["xcord", "ycord", "service_time"]].to_dict()
    time_matrix = []
    for i in df.customer:
        time_vector = []
        for j in df.customer:
            if i == j:
                time_vector.append(0)
            else:
                time = int(
                    time_precision_scaler
                    * math.hypot(
                        (travel_times["xcord"][i] - travel_times["xcord"][j]),
                        (travel_times["ycord"][i] - travel_times["ycord"][j]),
                    )
                )
                time += travel_times["service_time"][j]
                time_vector.append(time)
        time_matrix.append(time_vector)
    data["time_matrix"] = time_matrix

    with open(problem_path) as f:
        lines = f.readlines()
    data["num_vehicles"] = int(re.findall("[0-9]+", lines[4])[0])
    data["vehicle_capacities"] = [int(re.findall("[0-9]+", lines[4])[1])] * data[
        "num_vehicles"
    ]
    data["demands"] = list(df.demand)

    windows = df[["ready_time", "due_date", "service_time"]].to_dict()
    time_windows = []
    for i in df.customer:
        time_windows.append(
            (
                windows["ready_time"][i] + windows["service_time"][i],
                windows["due_date"][i] + windows["service_time"][i],
            )
        )
    data["time_windows"] = time_windows

    data["xcord"] = list(df.xcord)
    data["ycord"] = list(df.ycord)

    return data
```

Script solver_model.py for GLS

```
from pydantic import BaseModel

class SolverSetting(BaseModel):
    time_limit: int
```

A.5 The Whole Project's main.py Files

The main.py Script for All Models

```
import time
from matplotlib import pyplot as plt
from tabulate import tabulate
from aco.solve import solve_with_aco
from bks import bks_solution
from hgs.solve import solve_with_hgs
from gls.solve import solve_with_gls
from plot import plot_my_solution
from sa.solve import solve_using_sa
from pyvrp import read

dataset = "r211"
INPUT_PATH = f"data/{dataset}.txt"
BKS_PATH = f"data/{dataset}.sol"
RUNTIME = 120  # seconds

INSTANCE = read(INPUT_PATH, instance_format="solomon", round_func="trunc1")

result = {
    "bks": {},
    "hgs": {},
    "gls": {},
    "aco": {},
    "sa": {},
}
print("Running Algorithms on dataset:", dataset)
result["bks"]["routes"], result["bks"]["cost"] = bks_solution(BKS_PATH)

start = time.time()
result["hgs"]["routes"], result["hgs"]["cost"] = solve_with_hgs(INPUT_PATH, RUNTIME)
result["hgs"]["runtime"] = time.time() - start

start = time.time()
result["gls"]["routes"], result["gls"]["cost"] = solve_with_gls(INPUT_PATH, RUNTIME)
result["gls"]["runtime"] = time.time() - start

start = time.time()
result["aco"]["routes"], result["aco"]["cost"] = solve_with_aco(INPUT_PATH)
result["aco"]["runtime"] = time.time() - start

start = time.time()
result["sa"]["routes"], result["sa"]["cost"] = solve_using_sa(INPUT_PATH)
result["sa"]["runtime"] = time.time() - start

_, ax = plt.subplots(figsize=(10, 10))
plot_my_solution(result["hgs"], INSTANCE, ax=ax, dataset=dataset, algo="HGS")

_, ax = plt.subplots(figsize=(10, 10))
plot_my_solution(result["gls"], INSTANCE, ax=ax, dataset=dataset, algo="GLS")

_, ax = plt.subplots(figsize=(10, 10))
plot_my_solution(result["aco"], INSTANCE, ax=ax, dataset=dataset, algo="ACO")

_, ax = plt.subplots(figsize=(10, 10))
plot_my_solution(result["sa"], INSTANCE, ax=ax, dataset=dataset, algo="SA")

gap = lambda bks_cost, algo_cost: round(100 * (algo_cost - bks_cost) / bks_cost, 2)
header = ["Algorithms", "No. of Routes", "Costs", "Gap(%)", "Runtime(seconds)"]
rows = [
    ["BKS", len(result["bks"]["routes"]), result["bks"]["cost"], "-", "-"],
    [
        "HGS",
```

```
        len(result["hgs"]["routes"]),
        result["hgs"]["cost"],
        gap(result["bks"]["cost"], result["hgs"]["cost"]),
        result["hgs"]["runtime"],
    ],
    [
        "GLS",
        len(result["gls"]["routes"]),
        result["gls"]["cost"],
        gap(result["bks"]["cost"], result["gls"]["cost"]),
        result["gls"]["runtime"],
    ],
    [
        "ACO",
        len(result["aco"]["routes"]),
        result["aco"]["cost"],
        gap(result["bks"]["cost"], result["aco"]["cost"]),
        result["aco"]["runtime"],
    ],
    [
        "SA",
        len(result["sa"]["routes"]),
        result["sa"]["cost"],
        gap(result["bks"]["cost"], result["sa"]["cost"]),
        result["sa"]["runtime"],
    ],
]
print("Algorithm results on dataset:", dataset)
tabulate(rows, header, tablefmt="html")
```

The bks.py Script for All Models

```
from vrplib import read_solution

def bks_solution(bks_path):
    BKS = read_solution(bks_path)
    BKS_ROUTES = BKS["routes"]
    BKS_COST = BKS["cost"]
    print("BKS cost:", BKS_COST)
    print("BKS solution:")
    for i, route in enumerate(BKS_ROUTES, start=1):
        print(f"Route #{i}: {' '.join(str(node) for node in route)}")
    print()

    return BKS_ROUTES, BKS_COST
```

The plot.py Script for All Models

```
from typing import Optional

import matplotlib.pyplot as plt
import numpy as np

from pyvrp import ProblemData

def plot_my_solution(
    solution: any,
    data: ProblemData,
    ax: Optional[plt.Axes] = None,
    dataset: str = "c101",
    algo: str = "HGS",
):
    """
    Plots the given solution.

    Parameters
    ----------
    solution
```

```
        Solution to plot.
    data
        Data instance underlying the solution.
    plot_clients
        Whether to plot clients as dots. Default False, which plots only the
        solution's routes.
    ax
        Axes object to draw the plot on. One will be created if not provided.
    """
    if not ax:
        _, ax = plt.subplots()

    dim = data.num_clients + 1
    x_coords = np.array([data.client(client).x for client in range(dim)])
    y_coords = np.array([data.client(client).y for client in range(dim)])

    # This is the depot
    kwargs = dict(c="tab:red", marker="*", zorder=3, s=500)
    ax.scatter(x_coords[0], y_coords[0], label="Depot", **kwargs)

    for idx, route in enumerate(solution["routes"], 1):
        x = x_coords[route]
        y = y_coords[route]

        # Coordinates of clients served by this route.
        ax.scatter(x, y, label=f"Route {idx}", zorder=3, s=75)
        ax.plot(x, y)

        # Edges from and to the depot, very thinly dashed.
        kwargs = dict(ls=(0, (5, 15)), linewidth=0.25, color="grey")
        ax.plot([x_coords[0], x[0]], [y_coords[0], y[0]], **kwargs)
        ax.plot([x[-1], x_coords[0]], [y[-1], y_coords[0]], **kwargs)

    ax.grid(color="grey", linestyle="solid", linewidth=0.2)

    ax.set_title(f"Solution - {dataset} - {algo} - {solution['cost']:.1f}")
    ax.set_aspect("equal", "datalim")
    ax.legend(frameon=False, ncol=2)
```

The README.md Script for All Models

````
# Development

## Create venv

```sh
python -m venv .venv
```

## Activate venv

```sh
. .venv/bin/activate
```

## Install requirements

```sh
pip install -r requirements.txt
```
````

The requirments.txt Script for All Models

```
ortools==9.8.3296
pandas==2.1.3
pydantic==2.5.1
pyvrp==0.6.3
tabulate==0.9.0
```

Bibliography

[1] N Siswanto A Adhi B Santosa. *A New Metaheuristics for Solving Vehicle Routing Problem: Partial Comparison Optimization*. Sept. 9, 2019. URL: https://iopscience.iop.org/article/10.1088/1757-899X/598/1/012023 (visited on 08/25/2020).

[2] D. de Werra A. Hertz E. Taillard. *Tabu Search*. vol. 5, pg. 121-136. 1997. URL: https://infoscience.epfl.ch/record/88614.

[3] Herbert Kopfer A. L. Kok C. Manuel Meyer. *A Dynamic Programming Heuristic for the Vehicle Routing Problem with Time Windows and European Community Social Legislation*. vol. 44, pg. 429-553. Nov. 2010. URL: https://pubsonline.informs.org/doi/10.1287/trsc.1100.0331.

[4] H.C. Watson A. Ratnaweera S.K. Halgamuge. *Self-organizing Hierarchical Particle Swarm Optimizer with Time-varying Acceleration Coefficients*. vol. 8, pg. 240-255. June 14, 2004. URL: https://ieeexplore.ieee.org/document/1304846.

[5] Antoine Harfouche Abbas Tarhini Kassem Danach. *Swarm Intelligence-based Hyper-heuristic for the Vehicle Routing Problem with Prioritized Customers*. vol. 308, pg. 549–570. 2022. URL: https://link.springer.com/article/10.1007/s10479-020-03625-5.

[6] John Bullinaria Abel García Nájera. *An Improved Multi-objective Evolutionary Algorithm For The Vehicle Routing Problem With Time Windows*. vol. 38, pg. 287-300. 2011. URL: https://www.sciencedirect.com/science/article/abs/pii/S0305054810001176?via%3Dihub.

[7] Sheldon H. Jacobson Alexander G. Nikolaev. *Simulated Annealing*. vol. 1, pg. 1-40. Jan. 1, 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_1.

[8] Khaled Ghedira Amine Dhahri Kamel Zidi. *A Variable Neighborhood Search for the Vehicle Routing Problem with Time Windows and Preventive Maintenance Activities*. vol. 47, pg. 229-236. Feb. 2015. URL: https://doi.org/10.1016/j.endm.2014.11.030.

[9] Alan Holliday Anthony Wren. *Computer Scheduling of Vehicles from One or More Depots to a Number of Delivery Points*. 1972. URL: https://www.tandfonline.com/doi/abs/10.1057/jors.1972.53.

[10] Niaz A. Wassan Arif Imran Said Salhi. *A Variable Neighborhood-based Heuristic for the Heterogeneous Fleet Vehicle Routing Problem*. vol. 197, pg. 509-518. Sept. 2009. URL: https://doi.org/10.1016/j.ejor.2008.07.022.

[11] Volker Gruhn Asvin Goel. *A General Vehicle Routing Problem*. vol. 191, pg. 650-660. 2008. URL: https://d-nb.info/1249798817/34.

[12] Mingyuan Chen Attahiru Sule Alfa Sundresh S. Heragu. *A 3-OPT Based Simulated Annealing Algorithm for Vehicle Routing Problems*. vol. 21, pg. 635-639. 1991. URL: https://doi.org/10.1016/0360-8352(91)90165-3.

[13] B. Z. Yao B. Yu zhong zhen Yang. *A Hybrid Algorithm For Vehicle Routing Problem With Time Windows*. vol. 38, pg. 435-441. Jan. 2011. URL: https://www.sciencedirect.com/science/article/abs/pii/S0957417410005865?via%3Dihub.

[14] Annisa Kesy Garside Baiq Nurul Izzah Farida Ramadhani. *Particle Swarm Optimization Algorithm to Solve Vehicle Routing Problem with Fuel Consumption Minimization*. 2021. URL: https://josi.ft.unand.ac.id/index.php/josi/article/view/475/233.

[15] J. E. Beasle. *Route First—Cluster Second Methods for Vehicle Routing*. vol. 11, pg. 403-408. 1983. URL: https://doi.org/10.1016/0305-0483(83)90033-6.

[16] John E Beasley. *Route First—Cluster Second Methods for Vehicle Routing*. vol. 11, pg. 403-408. 1983. URL: https://doi.org/10.1016/0305-0483(83)90033-6.

[17] Richard Bellman. *The Theory of Dynamic Programming*. vol. 60, pg. 503–515. 1954. URL: https://www.ams.org/journals/bull/1954-60-06/S0002-9904-1954-09848-8/S0002-9904-1954-09848-8.pdf.

[18] Christine Strauss Bernd Bullnheimer Richard F. Hartl. *An Improved Ant System Algorithm for the Vehicle Routing Problem*. 1997. URL: https://www.researchgate.net/publication/274732340_An_improved_Ant_System_algorithm_for_the_Vehicle_Routing_Problem.

[19] Christine Strauss Bernd Bullnheimer Richard F. Hartl. *Applying the Ant System to the Vehicle Routing Problem*. 1999. URL: https://www.researchgate.net/publication/259687430_Applying_the_Ant_System_to_the_Vehicle_Routing_Problem.

[20] Leland R. Miller Billy E. Gillett. *A Heuristic Algorithm for the Vehicle-Dispatch Problem*. vol. 22, pg. 205-451. Apr. 1, 1974. URL: https://doi.org/10.1287/opre.22.2.340.

[21] Zhong Zhen Yang Bin Yu. *An Ant Colony Optimization Model: The Period Vehicle Routing Problem with Time Windows*. vol. 47, pg. 166-181. Mar. 2011. URL: https://doi.org/10.1016/j.tre.2010.09.010.

[22] Ivona Brajevic. *Artificial Bee Colony Algorithm for the Capacitated Vehicle Routing Problem*. 2011. URL: https://www.researchgate.net/publication/229051516_Artificial_bee_colony_algorithm_for_the_capacitated_vehicle_routing_problem.

[23] José Brandão. *A Tabu Search Algorithm for the Heterogeneous Fixed Fleet Vehicle Routing Problem*. vol. 38, pg. 140-151. Jan. 2011. URL: https://doi.org/10.1016/j.cor.2010.04.008.

[24] Olli Bräysy. *A Reactive Variable Neighborhood Search for the Vehicle-Routing Problem with Time Windows*. vol. 15, pg. 347-368. 2003. URL: https://pubsonline.informs.org/doi/10.1287/ijoc.15.4.347.24896.

[25] Paul Willy Budi Santosa. *Metode Metaheuristik Konsep dan Implementasi*. 2011. URL: https://onesearch.id/Record/IOS3201.slims-1049.

[26] Arnold Reisman Burak Eksioglu Arif Volkan Vural. *The Vehicle Routing Problem: A Taxonomic Review*. vol. 57, pg. 1472-1483. Nov. 11, 2008. URL: https://doi.org/10.1016/j.cie.2009.05.009 (visited on 05/25/2009).

[27] Guy Desaulniers C. Archetti Mathieu Bouchard. *Enhanced Branch and Price and Cut for Vehicle Routing with Split Deliveries and Time Windows*. vol. 45, pg. 285-298. Aug. 2011. URL: https://www.researchgate.net/publication/236158473_Enhanced_Branch_and_Price_and_Cut_for_Vehicle_Routing_with_Split_Deliveries_and_Time_Windows.

[28] G.T.S. Ho Canhong Lin K.L. Choy. *Survey of Green Vehicle Routing Problem: Past and Future Trends*. vol. 41, pg. 1118-1138. Aug. 13, 2013. URL: https://doi.org/10.1016/j.eswa.2013.07.107 (visited on 03/2014).

[29] Bo Söderberg Carsten Peterson. *Artiflcial Neural Networks*. vol. 7, pg. 173-214. 1997. URL: http://home.thep.lu.se/~carsten/pubs/lu_tp_97_38.pdf.

[30] V. Cerny. *Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm*. vol. 45, pg. 41-51. 1985. URL: https://www.webpages.uidaho.edu/~stevel/565/literature/tsp.pdf.

[31] Alain Chabrier. *Vehicle Routing Problem with Elementary Shortest Path Based Column Generation*. vol. 33, pg. 2972-2990. Mar. 26, 2005. URL: https://doi.org/10.1016/j.cor.2005.02.029.

[32] Lance D. Chambers. *The Practical Handbook of Genetic Algorithms*. 2000. URL: https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Handbook%20Genetic%20Algorithms.pdf.

[33] *Check VRP Instance Feasibility*. URL: https://or.stackexchange.com/questions/6627/check-vrp-instance-is-feasibility.

[34] Andrea Roli Christian Blum. *Hybrid Metaheuristics: An Introduction*. vol. 1, pg. 1-30. 2008. URL: https://www.researchgate.net/profile/Andrea-Roli/publication/226963067_Hybrid_Metaheuristics_An_Introduction/links/09e415107aa86978f2000000/Hybrid-Metaheuristics-An-Introduction.pdf.

[35] Antonio J. Fernández Christian Blum Carlos Cotta. *Hybridizations of Metaheuristics With Branch Bound Derivates*. vol. 4, pg. 85-116. 2008. URL: https://link.springer.com/chapter/10.1007/978-3-540-78295-7_4.

[36] Nicos Christofides. *Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem*. 1976. URL: https://link.springer.com/article/10.1007/s43069-021-00101-z.

[37] Abdullah Alsheddy Christos Voudouris Edward P.K. Tsang. *Guided Local Search*. vol. 11, pg. 321-361. 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_11.

[38] Hadi Gökçen Cihan Çetinkaya Ismail Karaoglan. *Two-stage Vehicle Routing Problem With Arc Time Windows: A Mixed Integer Programming Formulation And A Heuristic Approach*. vol. 230, pg. 539-550. Jan. 2013. URL: https://www.sciencedirect.com/science/article/abs/pii/S0377221713003858.

[39] G. B. Dantzig and J. H. Ramser. *The Truck Dispatching Problem*. vol. 6, pg. 80-91. Oct. 1959. URL: https://www.jstor.org/stable/2627477 (visited on 09/28/2008).

[40] Oliver Hinder David Applegate Mateo Diaz. *Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient*. 2021. URL: https://openreview.net/forum?id=_eXwwWOyqT_.

[41] Vašek Chvátal David Applegate Robert E. Bixby. *The Traveling Salesman Problem: A Computational Study*. 2011. URL: https://www.researchgate.net/publication/268645053_The_Traveling_Salesman_Problem_A_Computational_Study.

[42] L. Davis. *Handbook of Genetic Algorithms*. 1991. URL: https://www.semanticscholar.org/paper/Handbook-Of-Genetic-Algorithms-Davis/54acdb67ca083326c34eabdeb59bfdc01c748df0.

[43] Giovanni Rinaldi Denis Naddef. *Branch-And-Cut Algorithms for the Capacitated VRP*. vol. 3, pg. 53-84. 2002. URL: https://doi.org/10.1137/1.9780898718515.ch3.

[44] Ashish Girdhar Divya Aggarwal Vijay Chahar. *Lagrangian Relaxation for the Vehicle Routing Problem with Time Windows*. 2017. URL: https://ieeexplore.ieee.org/document/8342810.

[45] Fabien Lehuédé Dominique Feillet Thierry Garaix. *A New Consistent Vehicle Routing Problem for the Transportation of People with Disabilities*. vol. 63, pg. 211-224. Feb. 24, 2014. URL: https://doi.org/10.1002/net.21538 (visited on 05/2014).

[46] Nenad Mladenović Dragan Urošević Aleksandar Ilić. *A General Variable Neighborhood Search for Solving the Uncapacitated Single Allocation p-hub Median Problem*. vol. 206, pg. 289-300. 2010. URL: https://www.academia.edu/19470127/A_general_variable_neighborhood_search_for_solving_the_uncapacitated_single_allocation_p_hub_median_problem.

[47] P. Toth E. Balas. *Branch and Bound Methods for the Traveling Salesman Problem*. vol. 10, pg. 361-401. Mar. 1, 1983. URL: https://www.semanticscholar.org/paper/Branch-and-Bound-Methods-for-the-Traveling-Salesman-Balas-Toth/3446fc85faacecd1ef6ad5d1a3f85cb6fff8cb36.

[48] P. J. M. van Laarhoven E. H. L. Aarts J. H. M. Korst. *Simulated Annealing*. vol. 4, pg. 91-120. 1997. URL: https://research.tue.nl/en/publications/simulated-annealing.

[49] A. H. G. Rinnooy Kan E. L. Lawler Jan Karel Lenstra. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. 1985. URL: https://doi.org/10.1002/net.3230180309.

[50] J.K. Lenstra E.H.L. Aarts. *Local Search in Combinatorial Optimization*. 1997. URL: https://research.tue.nl/en/publications/local-search-in-combinatorial-optimization.

[51] Graham Kendall Edmund K. Burke. *Search Methodologies Introductory Tutorials in Optimization and Decision Support Techniques*. vol. 2, pg. 19-68. 2005. URL: https://www.nibmehub.com/opac-service/pdf/read/Search%20Methodologies-%202nd%20edition-%202014.pdf.

[52] Yuri Bykov Edmund K. Burke. *A Late Acceptance Strategy in Hill Climbing for Exam Timetabling Problems*. 2008. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.180.8129&rep=rep1&type=pdf.

[53] Chris T. Kiranoudis Emmanouil E. Zachariadis Christos D. Tarantilis. *The Vehicle Routing Problem with Simultaneous Pick-ups and Deliveries and Two-Dimensional Loading Constraints*. vol. 251, pg. 369-386. Oct. 1, 2014. URL: https://www.sciencedirect.com/science/article/abs/pii/S037722171501067X (visited on 06/01/2016).

[54] Chris Kiranoudis Emmanouil Zachariadis. *An Open Vehicle Routing Problem Metaheuristic for Examining Wide Solution Neighborhoods*. vol. 37, pg. 712-723. June 21, 2009. URL: https://doi.org/10.1016/j.cor.2009.06.021 (visited on 04/2010).

[55] Günes Erdögan. *VRP Spreadsheet Solver*. URL: https://people.bath.ac.uk/ge277/vrp-spreadsheet-solver/.

[56] Luca Maria Gambardella Éric D. Taillard. *Adaptive Memories for the Quadratic Assignment Problem*. 1997. URL: https://www.researchgate.net/publication/277289470_Adaptive_Memories_for_the_Quadratic_Assignment_Problem.

[57] Louis-Martin Rousseau Eric Prescott-Gagnon Guy Desaulniers. *A branch-and-price-based Large Neighborhood Search Algorithm For The Vehicle Routing Problem With Time Windows*. vol. 54, pg. 190-204. Dec. 2009. URL: https://onlinelibrary.wiley.com/doi/10.1002/net.20332.

[58] Miguel Andres Figliozzi. *The Time Dependent Vehicle Routing Problem with Time Windows: Benchmark Problems, an Efficient Solution Algorithm, and Solution Characteristics*. vol. 48, pg. 616-636. Aug. 15, 2011. URL: https://doi.org/10.1016/j.tre.2011.11.006 (visited on 12/24/2011).

[59] Miguel Andres Figliozzi. *The Time Dependent Vehicle Routing Problem With Time Windows: Benchmark Problems, An Efficient Solution Algorithm, And Solution Characteristics*. vol. 48, pg. 616-636. 2012. URL: https://www.sciencedirect.com/science/article/abs/pii/S1366554511001426.

[60] Merrill M. Flood. *The Traveling Salesman Problem*. vol. 4, pg. 1-137. 1956. URL: https://pubsonline.informs.org/doi/10.1287/opre.4.1.61.

[61] Manuel Laguna Fred Glover. *Tabu Search*. 1997. URL: https://link.springer.com/book/10.1007/978-1-4615-6089-0.

[62] Rafael Marti Fred Glover Manuel Laguna. *Scatter Search and Path Relinking: Advances and Applications*. vol. 1, pg. 1-35. 2003. URL: https://link.springer.com/chapter/10.1007/0-306-48056-5_1.

[63] Gerald J. Lieberman Frederick S. Hillier. *Introduction To Operations Research*. Jan. 1969. URL: DOI:10.2307/2345190.

[64] J. W. Wright G. Clarke. *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points*. vol. 12, pg. 568–582. Aug. 1, 1964. URL: https://doi.org/10.1287/opre.12.4.568 (visited on 1964).

[65] J. W. Wright G. Clarke. *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points*. vol. 12, pg. 519-643. Aug. 1964. URL: https://doi.org/10.1287/opre.12.4.568.

[66] I. H. Osman G. Laporte. *Routing Problems: A Bibliography*. vol. 61, pg. 227-262. Jan. 1995. URL: DOI:10.1007/BF02098290.

[67] J-Y. Potvin G. Laporte M. Gendreau. *Classical and Modern Heuristics for the Vehicle Routing Problem*. vol. 7, pg. 285-300. Sept. 2000. URL: https://doi.org/10.1111/j.1475-3995.2000.tb00200.x.

[68] Y. Nobert G. Laporte. *Exact Algorithms for the Vehicle Routing Problem*. vol. 31, pg. 147-184. 1987. URL: DOI:10.1016/S0304-0208(08)73235-3.

[69] H. Paul Williams Gautam Appa Leonidas Pitsoulis. *Handbook on Modelling for Discrete Optimization*. vol. 1, pg. 3-60. Jan. 2006. URL: https://www.researchgate.net/publication/48909727_Handbook_on_Modelling_for_Discrete_Optimization (visited on 2006).

[70] Michel Gendreau. *An Introduction to Tabu Search*. vol. 2, pg. 37-54. Apr. 2006. URL: https://www.researchgate.net/publication/227008055_An_Introduction_to_Tabu_Search.

[71] Gregory Prastacos George Ioannou Manolis Kritikos. *A Greedy Look-ahead Heuristic for the Vehicle Routing Problem with Time Windows*. vol. 52, pg. 523-537. 2001. URL: https://www.tandfonline.com/doi/abs/10.1057/palgrave.jors.2601113.

[72] Jonathan F. Bard George Kontoravdis. *A GRASP for the Vehicle Routing Problem with Time Windows*. vol. 7, pg. 1-116. 1995. URL: https://pubsonline.informs.org/doi/10.1287/ijoc.7.1.10.

[73] Martin W. P. Savelsbergh Gerard A. P. Kindervater. *Vehicle Routing: Handling Edge Exchanges*. vol. 10, pg. 337-360. 1997. URL: https://doi.org/10.1515/9780691187563-013.

[74] Gilbert Laporte Gerardo Berbeglia Jean-François Cordeau. *A Hybrid Tabu Search and Constraint Programming Algorithm for the Dynamic Dial-a-Ride Problem*. vol. 24, pg. 343-355. 2012. URL: https://pubsonline.informs.org/doi/10.1287/ijoc.1110.0454.

[75] Frederic Semet Gilbert Laporte. *Classical Heuristics for the Capacitated VRP*. vol. 5, pg. 109-128. 2002. URL: https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/002-The-Vehicle-Routing-Problem-Monograf-on-discrete-mathematics-and-applications-Paolo-Toth-Daniele-Vigo-Edisi-1-2002.pdf.

[76] Chen Kim Heng Gitae Kim Yew Soon Ong. *City Vehicle Routing Problem (City VRP): A Review*. vol. 16, pg. 1654-1666. Aug. 4, 2015. URL: https://www.memetic-computing.org/publication/journal/city-vrp-survey.pdf (visited on 2015).

[77] Fred Glover. *Future Paths for Integer Programming and Links to Artificial Intelligence*. vol. 13, pg. 533–549. 1986. URL: https://doi.org/10.1016/0305-0548(86)90048-1 (visited on 05/16/2003).

[78] Fred Glover. *Future Paths for Integer Programming and Links to Artificial Intelligence*. vol. 13, pg. 533-549. Jan. 1986. URL: https://www.researchgate.net/publication/222721339_Future_Paths_for_Integer_Programming_and_Links_to_Artificial_Intelligence_Computers_Operations_Research_13_533-549.

[79] B. Goldengorin. *Synergy Effects in Branch-and-Bound Algorithms for the Asymmetric Capacitated Vehicle Routing Problem*. Technical report. 2006.

[80] Ralph E. Gomory. *Outline of an Algorithm for Integer Solutions to Linear Programs*. vol. 64, pg. 275-278. Sept. 1958. URL: https://projecteuclid.org/journals/bulletin-of-the-american-mathematical-society-new-series/volume-64/issue-5/Outline-of-an-algorithm-for-integer-solutions-to-linear-programs/bams/1183522679.full.

[81] *Google OR-Tools*. URL: https://developers.google.com/optimization/introduction.

[82] Abraham P. Punnen Gregory Gutin. *The Traveling Salesman Problem and Its Variations*. 2007. URL: http://doi.org/10.1007/b101971.

[83] Chris Groer. *VRPH*. URL: https://github.com/coin-or/VRPH.

[84] *Guided Local Search (GLS)*. URL: https://acrogenesis.com/or-tools/documentation/user_manual/manual/metaheuristics/GLS.html.

[85] Christian Blum Günther R. Raidl Jakob Puchinger. *Metaheuristic Hybrids*. vol. 16, pg. 469-496. 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_16.

[86] Jakob Puchinger Günther R. Raidl. *Combining (Integer) Linear Programming Techniques and Metaheuristics for Combinatorial Optimization*. vol. 2, pg. 31-62. 2008. URL: https://link.springer.com/chapter/10.1007/978-3-540-78295-7_2.

[87] Marius M. Solomon Guy Desaulniers Jacques Desrosiers. *Column Generation*. 2005. URL: https://link.springer.com/book/10.1007/b135457.

[88] Toshihide Ibaraki Hideki Hashimoto Mutsunori Yagiura. *An Iterated Local Search Algorithm for the Time-Dependent Vehicle Routing Problem with Time Windows — Detailed Computational Results*. vol. 5, pg. 434-456. May 2008. URL: https://doi.org/10.1016/j.disopt.2007.05.004.

[89] John H. Holland. *Adaptation in Natural and Artificial Systems*. 1975. URL: https://mitpress.mit.edu/9780262581110/adaptation-in-natural-and-artificial-systems/.

[90] Kwong Meng Teo Hoong Chuin Lau Melvyn Sim. *Discrete Optimization Vehicle Routing Problem with Time Windows and a Limited Number of Vehicles*. vol. 148, pg. 559–569. 2003. URL: https://www.academia.edu/2887526/Vehicle_routing_problem_with_time_windows_and_a_limited_number_of_vehicles (visited on 2003).

[91] Germano Crispim Vasconcelos Humberto César Brandão de Oliveira. *A Hybrid Search Method for the Vehicle Routing Problem with Time Windows*. vol. 180, pg. 125–144. 2010. URL: https://link.springer.com/article/10.1007/s10479-008-0487-y.

[92] Eduardo Uchoa Humberto Longo Marcus Poggi de Aragão. *Solving Capacitated Arc Routing Problems Using a Transformation to the CVRP*. vol. 33, pg. 1823-1837. 2006. URL: https://doi.org/10.1016/j.cor.2004.11.020.

[93] J. R. C. Holland I. M. Oliver D. J. Smith. *A Study of Permutation Crossover Operators on the Traveling Salesman Problem*. 1987. URL: https://www.researchgate.net/publication/201976411_A_Study_of_Permutation_Crossover_Operators_on_the_Traveling_Salesman_Problem.

[94] James P. Kelly Ibrahim H Osman. *Meta-heuristics: Theory and Application*. vol. 1, pg. 1-22. Jan. 1996. URL: https://www.researchgate.net/publication/268626261_Meta-Heuristics_An_Overview (visited on 2008).

[95] Nursel Öztürk İlker Küçükoğlu. *An Advanced Hybrid Meta-heuristic Algorithm for the Vehicle Routing Problem with Backhauls and Time Windows*. vol. 86, pg. 60-68. Aug. 2015. URL: https://doi.org/10.1016/j.cie.2014.10.014.

[96] Stefan Irnich. *A Unified Modeling and Solution Framework for Vehicle Routing and Local Search-Based Metaheuristics*. vol. 20, pg. 270-287. May 2008. URL: https://www.researchgate.net/publication/220668857_A_Unified_Modeling_and_Solution_Framework_for_Vehicle_Routing_and_Local_Search-Based_Metaheuristics.

[97] G. Laporte J-F Cordeau M. Gendreau. *A Guide to Vehicle Routing Heuristics*. vol. 53, pg. 512-522. Feb. 2002. URL: https://www.jstor.org/stable/823019.

[98] A. H. G. Rinnooy Kan J. K. Lenstra. *Complexity of Vehicle Routing and Scheduling Problems*. vol. 11, pg. 221-227. June 1981. URL: https://doi.org/10.1002/net.3230110211 (visited on 1981).

[99] M. W. P. Savelsbergh J. T. Linderoth. *A Computational Study of Search Strategies for Mixed Integer Programming*. vol. 11, pg. 137-187. May 1, 1999. URL: https://doi.org/10.1287/ijoc.11.2.173.

[100] Pedro Munari Jacek Gondzio Pablo González-Brevis. *New Developments in the Primal–Dual Column Generation Technique*. vol. 224, pg. 41-51. Jan. 1, 2013. URL: https://doi.org/10.1016/j.ejor.2012.07.024.

[101] Dragan Urošević Jack Brimberg Nenad Mladenović. *Variable Neighborhood Search for the Heaviest k-subgraph*. vol. 36, pg. 2885-2891. Nov. 2009. URL: https://doi.org/10.1016/j.cor.2008.12.020.

[102] Russell Eberhart James Kennedy. *Particle Swarm Optimization*. vol. 4, pg. 1942-1948. Nov. 1995. URL: https://ieeexplore.ieee.org/document/488968/authors#authors.

[103] Nenad Mladenović Jasmina Lazić Saïd Hanafi. *Variable Neighbourhood Decomposition Search for 0–1 Mixed Integer Programs*. vol. 37, pg. 1055-1067. June 2010. URL: https://doi.org/10.1016/j.cor.2009.09.010.

[104] Gilbert Laporte Jean-Francois Cordeau. *Modeling and Optimization of Vehicle Routing and Arc Routing Problems*. vol. 6, pg. 151-191. 2006. URL: https://link.springer.com/chapter/10.1007/0-387-32942-0_6.

[105] Mirko Maischberger Jean-François Cordeau. *A Parallel Iterated Tabu Search Heuristic For Vehicle Routing Problems*. vol. 39, pg. 2033-2050. Sept. 2012. URL: https://www.sciencedirect.com/science/article/abs/pii/S0305054811002826?via%3Dihub.

[106] Patrick R. McMullen John E. Bell. *Ant Colony Optimization Techniques for the Vehicle Routing Problem*. vol. 18, pg. 41-48. Nov. 5, 2004. URL: https://doi.org/10.1016/j.aei.2004.07.001.

[107] Bruce Golden John Silberholz. *Comparison of Metaheuristics*. vol. 21, pg. 625-640. 2010. URL: https://ideas.repec.org/h/spr/isochp/978-1-4419-1665-5_21.html.

[108] Paolo Toth John Willmer Escobar Rodrigo Linfati. *A Hybrid Granular Tabu Search Algorithm for the Multi-Depot Vehicle Routing Problem*. vol. 20, pg. 483–509. 2014. URL: https://link.springer.com/article/10.1007/s10732-014-9247-0.

[109] Nenad Mladenović José Andrés Moreno Pérez Pierre Hansen. *Variable Neighborhood Search: VNS for Training Neural Networks*. URL: https://jamoreno.webs.ull.es/www/papers/VNS2TNN.pdf.

[110] Christian Prins Jose-manuel Belenguer Enrique Benavent. *A Branch-and-Cut Method for the Capacitated Location-Routing Problem*. vol. 38, pg. 931-941. June 2011. URL: https://doi.org/10.1016/j.cor.2010.09.019.

[111] Fei-Fei Li Juan Carlos Niebles Hongcheng Wang. *Unsupervised Learning of Human Action Categories Using Spatial-Temporal Words*. vol. 79, pg. 299-318. Sept. 2008. URL: https://www.researchgate.net/publication/220659676_Unsupervised_Learning_of_Human_Action_Categories_Using_Spatial-Temporal_Words.

[112] Lionel Amodeo Julien Michallet Christian Prins. *Multi-start Iterated Local Search for the Periodic Vehicle Routing Problem with Time Windows and Time Spread Constraints on Services*. vol. 41, pg. 196-207. Jan. 2014. URL: https://doi.org/10.1016/j.cor.2013.07.025.

[113] Manfred Padberg Karla L. Hoffman. *Improving LP-Representations of Zero-One Linear Programs for Branch-and-Cut*. vol. 3, pg. 85-176. May 1, 1991. URL: https://doi.org/10.1287/ijoc.3.2.121.

[114] Katrien Ramaekers Kris Braekers Inneke Van Nieuwenhuyse. *The Vehicle Routing Problem: State of the Art Classification and Review*. Dec. 7, 2015. URL: https://www.researchgate.net/publication/287796502 (visited on 09/17/2018).

[115] Gilbert Laporte. *The Vehicle Routing Problem: An Overview of Exact and Approximate Algorithms*. vol. 59, pg. 345-358. June 25, 1992. URL: https://doi.org/10.1016/0377-2217(92)90192-C (visited on 01/13/2011).

[116] Gilbert Laporte. *Fifty Years of Vehicle Routing*. vol. 43, pg. 408–416. Oct. 21, 2009. URL: https://doi.org/10.1287/trsc.1090.0301 (visited on 11/2009).

[117] M.A. Lejeune. *A Variable Neighborhood Decomposition Search Method for Supply Chain Management Planning Problems*. vol. 175, pg. 959-976. Dec. 2006. URL: https://doi.org/10.1016/j.ejor.2005.05.021.

[118] Lan Leon. *VRPLIB*. URL: https://github.com/leonlan/VRPLIB.

[119] Shen Lin. *Computer Solutions of the Traveling Salesman Problem: Bell System Technical Journal*. vol. 44, pg. 2245–2269. Dec. 1965. URL: http://doi.org/10.1002/j.1538-7305.1965.tb04146.x (visited on 12/1965).

[120] Marco E. Lübbecke. *Column Generation*. Jan. 14, 2011. URL: https://doi.org/10.1002/9780470400531.eorms0158.

[121] Giovanni Agazzi Luca Maria Gambardella Ric Taillard. *MACS-VRPTW: A Multiple Ant Colony System for Vehicle Routing Problems with Time Windows*. 1999. URL: https://people.idsia.ch/~luca/tr-idsia-06-99.pdf.

[122] Marco Dorigo Luca Maria Gambardella. *Ant-Q: a Reinforcement Learning Approach to the Traveling Salesman Problem*. 1995. URL: https://www.researchgate.net/publication/221346170_Ant-Q_A_Reinforcement_Learning_Approach_to_the_Traveling_Salesman_Problem.

[123] Marco Dorigo Luca Maria Gambardella. *Solving Symmetric and Asymmetric TSPs by Ant Colonies*. 1996. URL: https://www.researchgate.net/publication/2581867_Solving_Symmetric_and_Asymmetric_TSPs_by_Ant_Colonies.

[124] C.L. Monma M.O. Ball T.L. Magnanti. *Network Routing, Handbooks in Operations Research and Management Science*. vol. 1, pg. 1-33. 1995.

[125] Stefan Ropke Mads Jepsen Simon Spoorendonk. *A Branch-and-Cut Algorithm for the Symmetric Two-Echelon Capacitated Vehicle Routing Problem*. vol. 47, pg. 23-37. Feb. 2013. URL: https://www.researchgate.net/publication/260140723_A_Branch-and-Cut_Algorithm_for_the_Symmetric_Two-Echelon_Capacitated_Vehicle_Routing_Problem.

[126] NeriFerrante Neri Marco A. Montes de Oca Carlos Cotta. *Local Search*. vol. 3, pg. 29-41. Jan. 2012. URL: https://www.researchgate.net/publication/278705669_Local_Search.

[127] Alberto Colorni Marco Dorigo Vittorio Maniezzo. *Positive Feedback as a Search Strategy*. 1991. URL: https://www.researchgate.net/publication/2573263_Positive_Feedback_as_a_Search_Strategy.

[128] Alberto Colorni Marco Dorigo Vittorio Maniezzo. *Ant System: Optimization by a Colony of Cooperating Agents*. vol. 26, pg. 29-41. 1996. URL: https://ieeexplore.ieee.org/document/484436.

[129] Luca Maria Gambardella Marco Dorigo. *Ant Colonies for the Traveling Salesman Problem*. vol. 43, pg. 73-81. 1997. URL: https://doi.org/10.1016/S0303-2647(97)01708-5.

[130] Luca Maria Gambardella Marco Dorigo. *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem*. vol. 1, pg. 53-56. 1997. URL: https://www.researchgate.net/publication/201977138_Ant_Colony_System_A_Cooperative_Learning_Approach_to_the_Traveling_Salesman_Problem.

[131] Thomas Stützle Marco Dorigo. *Ant Colony Optimization*. 2004. URL: https://web2.qatar.cmu.edu/~gdicaro/15382/additional/aco-book.pdf.

[132] Thomas Stützle Marco Dorigo. *Ant Colony Optimization: Overview and Recent Advances*. vol. 8, pg. 227-263. 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_8.

[133] Maria Zrikem Marie-Christine Costa Francois-Regis Monclar. *Variable Neighborhood Decomposition Search for the Optimization of Power Plant Cable Layout*. vol. 13, pg. 353-365. 2002. URL: https://www.academia.edu/17369142/Variable_neighborhood_decomposition_search_for_the_optimization_of_power_plant_cable_layout.

[134] Ramchandran Jaikumar Marshall L. Fisher. *A Generalized Assignment Heuristic for Vehicle Routing*. vol. 11, pg. 109-124. 1981. URL: https://doi.org/10.1002/net.3230110205.

[135] Rafael Martí. *Multi-Start Methods*. vol. 12, pg. 355-368. 2003. URL: https://www.uv.es/~rmarti/paper/docs/multi2.pdf.

[136] Martin Macho. *Vehicle Routing Problem with Time Windows and its Solving Methods*. 2017. URL: https://dspace.cvut.cz/bitstream/handle/10467/72934/F8-BP-2017-Macho-Martin-thesis.pdf?sequence=1&isAllowed=y (visited on 2017).

[137] Ilaria Vacca Matteo Salani. *Branch and Price for the Vehicle Routing Problem with Discrete Split Deliveries and Time Windows*. vol. 213, pg. 470–477. Mar. 15, 2011. URL: https://doi.org/10.1016/j.ejor.2011.03.023 (visited on 09/16/2011).

[138] Pedro Martins Mauricio C. de Souza. *Skewed VNS Enclosing Second Order Algorithm for the Degree Constrained Minimum Spanning Tree Problem*. vol. 191, pg. 677-690. Dec. 2008. URL: https://doi.org/10.1016/j.ejor.2006.12.061.

[139] Celso C. Ribeiro Mauricio G.C. Resende. *Greedy Randomized Adaptive Search Procedures*. vol. 8, pg. 219-249. 2003. URL: https://link.springer.com/chapter/10.1007/0-306-48056-5_8.

[140] Celso C. Ribeiro Mauricio G.C. Resende. *Greedy Randomized Adaptive Search Procedures: Advances, Hybridizations, and Applications*. vol. 10, pg. 283-319. 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_10.

[141] Fred Glover Mauricio G.C. Resende Celso C. Ribeiro. *Scatter Search and Path-Relinking: Fundamentals, Advances, and Applications*. vol. 4, pg. 87-107. 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_4.

[142] Stephan Winkler Michael Affenzeller Stefan Wagner. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*. 2009. URL: https://doi.org/10.1201/9781420011326.

[143] Michal Polivka. *Vehicle Routing Problem, its Variants and Solving Methods*. May 10, 2015. URL: https://dspace.cvut.cz/bitstream/handle/10467/63209/F8-BP-2015-Polivka-Michal-thesis.pdf?sequence=3&isAllowed=y (visited on 2015).

[144] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. 2004. URL: https://paginas.fe.up.pt/~mac/ensino/docs/OR/HowToSolveIt/Chapters_1-6_How_to_Solve_It_Modern_Heuristics.pdf.

[145] Jean-Yves Potvin Michel Gendreau. *Metaheuristics in Combinatorial Optimization*. vol. 140, pg. 189-213. Nov. 2005. URL: https://link.springer.com/article/10.1007/s10479-005-3971-7.

[146] Jean-Yves Potvin Michel Gendreau. *Tabu Search*. vol. 2, pg. 41-59. Jan. 1, 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_2.

[147] Jean-Yves Potvin Michel Gendreau Gilbert Laporte. *Vehicle Routing: Modern Heuristics*. vol. 9, pg. 311-336. 1997. URL: https://doi.org/10.1515/9780691187563-012.

[148] Jean-Yves Potvin Michel Gendreau Gilbert Laporte. *Metaheuristics for the Capacitated VRP*. vol. 6, pg. 129-154. 2002. URL: https://epubs.siam.org/doi/10.1137/1.9780898718515.ch6.

[149] John E. Mitchell. *Branch and Cut*. Jan. 14, 2011. URL: https://doi.org/10.1002/9780470400531.eorms0117.

[150] Enrique Alba Mostepha R. Khouadjia Briseida Sarasola. *A Comparative Study between Dynamic Adapted PSO and VNS for the Vehicle Routing Problem with Dynamic Requests*. vol. 12, pg. 1426-1439. Apr. 2012. URL: https://doi.org/10.1016/j.asoc.2011.10.023.

[151] P. Toth N. Christofides A. Mingozzi. *The vehicle routing problem: Christofides et al. 1979*. 1984. URL: http://www.vrp-rep.org/references/item/christofides-et-al-1979.html.

[152] M. N. Rosenbluth N. Metropolis A. W. Rosenbluth. *Equation of State Calculations by Fast Computing Machines*. vol. 21, pg. 1087-1092. 1953. URL: https://www.researchgate.net/publication/200622065_Equation_of_State_Calculations_by_Fast_Computing_Machines.

[153] P. Hansen N. Mladenović. *Variable Neighborhood Search*. vol. 24, pg. 1097-1100. Nov. 1997. URL: https://doi.org/10.1016/S0305-0548(97)00031-2.

[154] Jean-Yves Potvin Nabila Azi Michel Gendreau. *An Exact Algorithm For A Vehicle Routing Problem With Time Windows And Multiple Use of Vehicles*. vol. 202, pg. 756-763. May 2010. URL: https://www.sciencedirect.com/science/article/abs/pii/S0377221709004950?via%3Dihub.

[155] Aleksandar Ilić Nenad Mladenović Dragan Urošević. *A General Variable Neighborhood Search for the One-commodity Pickup-and-Delivery Travelling Salesman Problem*. vol. 220, pg. 270-285. July 1, 2012. URL: https://doi.org/10.1016/j.ejor.2012.01.036.

[156] Dragan Urošević Nenad Mladenović. *Variable Neighborhood Search for the K-Cardinality Tree*. vol. 86, pg. 481-500. 2003. URL: https://link.springer.com/chapter/10.1007/978-1-4757-4137-7_23.

[157] Mirjana M. Čangalović Nenad Mladenović Vera Kovačevic-Vujčić. *Solving Spread Spectrum Radar Polyphase Code Design Problem by Tabu Search and Variable Neighbourhood Search*. vol. 151, pg. 389-399. Dec. 1, 2003. URL: https://doi.org/10.1016/S0377-2217(02)00833-0.

[158] Pierre Hansen Nenad Mladenović Martine Labbé. *Solving the p-Center Problem with Tabu Search and Variable Neighborhood Search*. vol. 42, pg. 48-64. July 8, 2003. URL: https://doi.org/10.1002/net.10081.

[159] Vera Kovačevic-Vujčić Nenad Mladenović Milan Dražić. *General Variable Neighborhood Search for the Continuous Optimization*. vol. 191, pg. 753-770. Dec. 2008. URL: https://doi.org/10.1016/j.ejor.2006.12.064.

[160] Siti Meriam Zahari Nor Edayu Abdul Ghani S. Sarifah Radiah Shariff. *An Alternative Algorithm for Vehicle Routing Problem with Time Windows for Daily Deliveries*. vol. 6, pg. 342-350. Dec. 30, 2015. URL: https://file.scirp.org/pdf/APM_2016042214394769.pdf (visited on 04/22/2016).

[161] Wout Dullaert Olli Bräysy Geir Hasle. *A Multi-Start Local Search Algorithm for the Vehicle Routing Problem with Time Windows*. vol. 159, pg. 586-605. Dec. 2004. URL: https://doi.org/10.1016/S0377-2217(03)00435-1.

[162] *Open-VRP*. URL: https://github.com/mck-/Open-VRP.

[163] *or-tools User's Manual: Constraint Programming (CP)*. URL: https://acrogenesis.com/or-tools/documentation/user_manual/manual/introduction/what_is_cp.html.

[164] *or-tools User's Manual: Local Search*. URL: https://acrogenesis.com/or-tools/documentation/user_manual/manual/ls/local_search.html.

[165] Ibrahim H Osman. *Meta-strategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routine Problem*. vol. 41, pg. 421-451. Dec. 1993. URL: https://www.researchgate.net/publication/226358362_Meta-strategy_simulated_annealing_and_Tabu_search_algorithms_for_the_vehicle_routine_problem.

[166] D. Vigo P. Toth. *The vehicle routing problem*. 2002. URL: https://scholar.google.de/scholar?q=P.+Toth+and+D.+Vigo.+The+vehicle+routing+problem.+SIAM,+2002&hl=en&as_sdt=0&as_vis=1&oi=scholart.

[167] George Ioannou Panagiotis P. Repoussis Christos D. Tarantilis. *Arc-Guided Evolutionary Algorithm for the Vehicle Routing Problem With Time Windows*. vol. 13, pg. 624–647. 2009. URL: https://ieeexplore.ieee.org/document/5089893.

[168] Daniele Vigo Paolo Toth. *Branch-And-Bound Algorithms for the Capacitated VRP*. vol. 2, pg. 29-52. 2002. URL: https://doi.org/10.1137/1.9780898718515.ch2.

[169] Daniele Vigo Paolo Toth. *The Vehicle Routing Problem*. vol. 1, pg. 1-26. 2002. URL: https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/002-The-Vehicle-Routing-Problem-Monograf-on-discrete-mathematics-and-applications-Paolo-Toth-Daniele-Vigo-Edisi-1-2002.pdf (visited on 2002).

[170] Daniele Vigo Paolo Toth. *The Vehicle Routing Problem*. vol. 7, pg. 157-194. 2002. URL: https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/002-The-Vehicle-Routing-Problem-Monograf-on-discrete-mathematics-and-applications-Paolo-Toth-Daniele-Vigo-Edisi-1-2002.pdf (visited on 2002).

[171] Daniele Vigo Paolo Toth. *The Vehicle Routing Problem*. vol. 8, pg. 195-224. 2002. URL: https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/002-The-Vehicle-Routing-Problem-Monograf-on-discrete-mathematics-and-applications-Paolo-Toth-Daniele-Vigo-Edisi-1-2002.pdf (visited on 2002).

[172] Daniele Vigo Paolo Toth. *The Vehicle Routing Problem*. vol. 9, pg. 225-242. 2002. URL: https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/002-The-Vehicle-Routing-Problem-Monograf-on-discrete-mathematics-and-applications-Paolo-Toth-Daniele-Vigo-Edisi-1-2002.pdf (visited on 2002).

[173] Andreas Schutt Peter J. Stuckey Thibaut Feydy. *The MiniZinc Challenge 2008–2013*. 2014. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2539.

[174] Dionisio Perez-Britos Pierre Hansen Nenad Mladenović. *Variable Neighborhood Decomposition Search*. vol. 7, pg. 335-350. July 2001. URL: https://link.springer.com/article/10.1023/A:1011336210885.

[175] Dragan Urošević Pierre Hansen Nenad Mladenović. *Variable Neighborhood Search and Local Branching*. vol. 33, pg. 3034-3045. Oct. 2006. URL: https://doi.org/10.1016/j.cor.2005.02.033.

[176] José A. Moreno Pérez Pierre Hansen Nenad Mladenović. *Variable Neighbourhood Search: Methods and Applications*. vol. 6, pg. 319-360. 2008. URL: https://link.springer.com/article/10.1007/s10288-008-0089-1.

[177] José A. Moreno Pérez Pierre Hansen Nenad Mladenović. *Variable Neighborhood Search*. vol. 3, pg. 61-86. Jan. 1, 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_3.

[178] José A. Moreno Pérez Pierre Hansen Nenad Mladenović. *Variable Neighbourhood Search: Methods and Applications*. vol. 175, pg. 367-407. 2010. URL: https://link.springer.com/article/10.1007/s10479-009-0657-6.

[179] Nenad Mladenović Pierre Hansen. *An Introduction to Variable Neighborhood Search*. vol. 30, pg. 433-458. 1999. URL: https://link.springer.com/chapter/10.1007/978-1-4615-5775-3_30.

[180] Nenad Mladenović Pierre Hansen. *Developments of Variable Neighborhood Search*. vol. 19, pg. 415-440. 2000. URL: https://www.researchgate.net/publication/232786029_Essays_and_Surveys_in_Metaheuristics.

[181] Nenad Mladenović Pierre Hansen. *Variable Neighborhood Search*. vol. 6, pg. 145-184. 2003. URL: https://link.springer.com/chapter/10.1007/0-306-48056-5_6.

[182] Nenad Mladenović Pierre Hansen. *First vs. Best Improvement: An Empirical Study*. vol. 154, pg. 802-817. Apr. 2006. URL: https://doi.org/10.1016/j.dam.2005.05.020.

[183] Nenad Mladenović Pierre Hansen. *Variable Neighborhood Search*. URL: https://www.cs.uleth.ca/~benkoczi/OR/read/vns-tutorial.pdf.

[184] Nenad Mladenović Pierre Hansen Jasmina Lazić. *Variable Neighbourhood Search for Colour Image Quantization*. vol. 18, pg. 207-221. Apr. 2007. URL: https://doi.org/10.1093/imaman/dpm008.

[185] Victor Pillac. *Vroom*. URL: https://github.com/VROOM-Project/vroom/tree/master.

[186] *PyVRP*. URL: https://github.com/PyVRP/PyVRP.

[187] A. Duarte Rafael Martí J. Marcos Moreno-Vega. *Advanced Multi-Start Methods*. vol. 9, pg. 265-282. 2010. URL: https://ideas.repec.org/h/spr/isochp/978-1-4419-1665-5_9.html.

[188] Frédéric Semet Rahma Lahyani Mahdi Khemakhem. *Rich Vehicle Routing Problems: From a Taxonomy to a Definition*. vol. 241, pg. 1-14. Mar. 12, 2014. URL: https://doi.org/10.1016/j.ejor.2014.07.048 (visited on 02/16/2015).

[189] Colin R. Reeves. *Genetic Algorithms*. vol. 5, pg. 109-139. 2010. URL: https://link.springer.com/chapter/10.1007/978-1-4419-1665-5_5.

[190] Jens Lysgaard Ricardo Fukasawa Humberto Longo. *Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem*. vol. 106, pg. 491–511. Oct. 12, 2005. URL: https://link.springer.com/article/10.1007/s10107-005-0644-x.

[191] Marcus Poggi de Aragão Ricardo Fukasawa Humberto Longo. *Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem*. vol. 106, pg. 491-511. Oct. 12, 2005. URL: DOI:10.1007/s10107-005-0644-x (visited on 07/2006).

[192] Aristide Mingozzi Roberto Baldacci. *An Unified Exact Method for Solving Different Classes of Vehicle Routing Problems*. vol. 120, pg. 347-380. Sept. 2009. URL: https://www.researchgate.net/publication/220588884_Mingozzi_A_An_unified_exact_method_for_solving_different_classes_of_vehicle_routing_problems_Math_Program_Ser_A_1202_347-380.

[193] Aristide Mingozzi Roberto Baldacci Eleni Hadjiconstantinou. *An Exact Algorithm for the Capacitated Vehicle Routing Problem Based on a Two-Commodity Network Flow Formulation*. vol. 52, pg. 723-738. Oct. 2004. URL: https://www.researchgate.net/publication/220243696_An_Exact_Algorithm_for_the_Capacitated_Vehicle_Routing_Problem_Based_on_a_Two-Commodity_Network_Flow_Formulation.

[194] Eric A. Hansen Rong Zhou. *Breadth-first Heuristic Search*. vol. 170, pg. 385-408. Dec. 13, 2005. URL: doi:10.1016/j.artint.2005.12.002.

[195] Pascal Van Hentenryck Russell Bent. *A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows*. vol. 38, pg. 395-534. 2004. URL: https://pubsonline.informs.org/doi/10.1287/trsc.1030.0049.

[196] M. P. Vecchi S. Kirkpatrick C. D. Gelatt. *Optimization by Simulated Annealing*. vol. 220, pg. 671–680. May 13, 1983. URL: http://doi.org/10.1126/science.220.4598.671 (visited on 1983).

[197] S.C. Lenny Koh S.P. Anbuudayasankar K. Ganesh. *Modified Savings Heuristics And Genetic Algorithm For Bi-Objective Vehicle Routing Problem With Forced Backhauls*. vol. 39, pg. 2296-2305. Feb. 2012. URL: https://www.sciencedirect.com/science/article/abs/pii/S0957417411011201.

[198] J. Ramonet S.R. Balseiro I. Loiseau. *An Ant Colony algorithm hybridized with insertion heuristics for the Time Dependent Vehicle Routing Problem with Time Windows*. vol. 38, pg. 954-966. June 2011. URL: https://doi.org/10.1016/j.cor.2010.10.011.

[199] Samira Almoustafa. *Distance-Constrained Vehicle Routing Problem: Exact and Approximate Solution (Mathematical Programming)*. 2013. URL: http://bura.brunel.ac.uk/handle/2438/7640 (visited on 05/14/2018).

[200] Stefan Schröder. *Jsprit*. URL: https://jsprit.github.io/.

[201] M. P. Jr. Vecchi Scott Kirkpatrick C. D. Jr. Gelatt. *Optimization by Simulated Annealing*. vol. 220, pg. 671-680. May 13, 1983. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[202] Elise Miller-Hooks Sevgi Erdoğan. *A Green Vehicle Routing Problem*. vol. 48, pg. 100-114. Jan. 2012. URL: https://doi.org/10.1016/j.tre.2011.08.001.

[203] Weixiong Zhang Sharlee Climer. *Cut-and-Solve: An Iterative Search Strategy for Combinatorial Optimization Problems*. vol. 170, pg. 714-738. Apr. 17, 2006. URL: https://doi.org/10.1016/j.artint.2006.02.005.

[204] Fu-Hao Zhang Sheng-Hua Xu Ji-Ping Liu. *A Combination of Genetic Algorithm and Particle Swarm Optimization for Vehicle Routing Problem with Time Windows*. vol. 15, pg. 21033-21053. Aug. 2015. URL: https://www.mdpi.com/1424-8220/15/9/21033.

[205] Nasser El-Sherbeny. *Vehicle Routing with Time Windows: An Overview of Exact, Heuristic and Metaheuristic Methods*. vol. 22, pg. 123-131. July 2010. URL: https://doi.org/10.1016/j.jksus.2010.03.002.

[206] Wei-Chang Yeh Shi-Yi Tan. *The Vehicle Routing Problem: State-of-the-Art Classification and Review*. Sept. 28, 2021. URL: https://www.mdpi.com/2076-3417/11/21/10295 (visited on 11/02/2021).

[207] Shuo-Yan Chou Shih-Wei Lin Vincent F. Yu. *A Note on the Truck and Trailer Routing Problem*. vol. 37, pg. 899-903. July 5, 2009. URL: https://doi.org/10.1016/j.eswa.2009.06.077 (visited on 01/2010).

[208] Zne-Jung Lee Shih-Wei Lin Kuo-Ching Ying. *Vehicle Routing Problems with Time Windows Using Simulated Annealing*. 2006. URL: https://ieeexplore.ieee.org/document/4273905.

[209] Geoffrey De Smet. *Optaplanner*. URL: https://www.optaplanner.org/.

[210] Frits C. R. Spieksma Sofie Coene A Arnout. *On a Periodic Vehicle Routing Problem*. vol. 61, pg. 1719-1728. Oct. 1, 2008. URL: https://doi.org/10.1057/jors.2009.154 (visited on 12/21/2017).

[211] Marius M. Solomon. *Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints*. vol. 35, pg. 254-265. 1987. URL: https://www.iro.umontreal.ca/~dift6751/paper_solomon.pdf.

[212] *Solomon's, Homberger and Gehring's Benchmarks*. URL: http://vrp.galgos.inf.puc-rio.br/index.php/en.

[213] Keshav Dahal Stephen Remde Peter Cowling. *Exact/Heuristic Hybrids Using rVNS and Hyperheuristics for Workforce Scheduling*. pg. 188-197. Apr. 2007. URL: https://dl.acm.org/doi/10.5555/1761927.1761944.

[214] *SYMPHONY*. URL: https://github.com/coin-or/SYMPHONY.

[215] A M S Asih T Iswari. *Comparing Genetic Algorithm and Particle Swarm Optimization for Solving Capacitated Vehicle Routing Problem*. 2018. URL: https://iopscience.iop.org/article/10.1088/1757-899X/337/1/012004.

[216] Voratas Kachitvichyanukul The Jin Ai. *A Particle Swarm Optimization for the Vehicle Routing Problem with Simultaneous Pickup and Delivery*. vol. 36, pg. 1693-1702. May 2009. URL: https://doi.org/10.1016/j.cor.2008.04.003.

[217] Voratas Kachitvichyanukul The Jin Ai. *A Particle Swarm Optimization for the Vehicle Routing Problem with Simultaneous Pickup and Delivery*. vol. 36, pg. 1693-1702. May 2009. URL: https://doi.org/10.1016/j.cor.2008.04.003.

[218] Michel Gendreau Thibaut Vidal Teodor Gabriel Crainic. *A Hybrid Genetic Algorithm with Adaptive Diversity Management for a Large Class of Vehicle Routing Problems with Time-Windows*. vol. 40, pg. 475-489. 2013. URL: https://doi.org/10.1016/j.cor.2012.07.018.

[219] Michel Gendreau Thibaut Vidal Teodor Gabriel Crainic. *Heuristics for Multi-Attribute Vehicle Routing Problems : A Survey and Synthesis*. vol. 231, pg. 1-21. Nov. 2013. URL: https://www.sciencedirect.com/science/article/abs/pii/S0377221713002026?via%3Dihub.

[220] Mauricio G. C. Resende Thomas A. Feo. *Greedy Randomized Adaptive Search Procedures*. vol. 6, pg. 109-133. 1995. URL: https://link.springer.com/article/10.1007/BF01096763.

[221] Holger H Hoos Thomas Stützle. *Stochastic Local Search-Foundations and Applications*. pg. 149-202. 2005. URL: https://doi.org/10.1016/B978-155860872-6/50021-4.

[222] Alexander Martin Tobias Achterberga Thorsten Koch. *Branching Rules Revisited*. vol. 33, pg. 42-54. June 26, 2004. URL: doi:10.1016/j.orl.2004.04.002.

[223] Ante Galic Tonci Caric Juraj Fosin. *Empirical Analysis of Two Different Metaheuristics for Real-World Vehicle Routing Problems*. vol. 4771, pg. 31–44. 2007. URL: https://link.springer.com/chapter/10.1007/978-3-540-75514-2_3.

[224] Izzatdin Abdul Aziz Tung Son Ngo Jafreezal Jaafar. *Metaheuristic Algorithms Based on Compromise Programming for the Multi-Objective Urban Shipment Problem*. Nov. 30, 2021. URL: https://doi.org/10.3390/e24030388 (visited on 03/09/2022).

[225] Soheil Fathi Vahid Mahdavi Asl Seyed Amir Sadeghi. *A Mathematical Model and Solving Method for Multi-Depot and Multi-level Vehicle Routing Problem with Fuzzy Time Windows*. vol. 1, pg. 19–24. Jan. 2012. URL: https://www.researchgate.net/publication/268435368_A_mathematical_Model_and_Solving_Method_for_Multi-Depot_and_Multi-level_Vehicle_Routing_Problem_with_Fuzzy_Time_Windows.

[226] Christelle Guéret Victor Pillac Michel Gendreau. *A Review of Dynamic Vehicle Routing Problems*. vol. 225, pg. 1–11. 2013. URL: https://www.sciencedirect.com/science/article/abs/pii/S0377221712006388?via%3Dihub.

[227] Thibaut Vidal. *Technical Note: Split Algorithm in O(n) for the Capacitated Vehicle Routing Problem*. vol. 69, pg. 40-47. May 2016. URL: https://doi.org/10.1016/j.cor.2015.11.012.

[228] Thibaut Vidal. *HGS-CVRP: A Modern Implementation of the Hybrid Genetic Search for the CVRP*. 2020. URL: https://github.com/vidalt/HGS-CVRP.

[229] Thibaut Vidal. *Hybrid Genetic Search for the CVRP: Open-Source Implementation and SWAP* Neighborhood*. Apr. 2022. URL: https://www.sciencedirect.com/science/article/abs/pii/S030505482100349X?via%3Dihub.

[230] Chao-Lung Yang Vincent F. Yu A. A. N. Perwira Redi. *Symbiotic Organisms Search and Two Solution Representations for Solving the Capacitated Vehicle Routing Problem*. vol. 52, pg. 657–672. Mar. 1, 2017. URL: https://doi.org/10.1016/j.asoc.2016.10.006 (visited on 03/01/2017).

[231] Marc Reimann Vitória Pureza Reinaldo Morabito. *Vehicle Routing with Multiple Deliverymen: Modeling and Heuristic Approaches for the VRPTW*. vol. 218, pg. 636–647. May 1, 2012. URL: https://doi.org/10.1016/j.ejor.2011.12.005 (visited on 2012).

[232] Sin C. Ho W.Y. Szeto Yongzhong Wu. *An Artificial Bee Colony Algorithm for the Capacitated Vehicle Routing Problem*. vol. 215, pg. 126-135. Nov. 2011. URL: https://doi.org/10.1016/j.ejor.2011.06.006.

[233] Mohammed Ouali Walid Mahdi Seyyid Ahmed Medjahed. *Performance Analysis of Simulated Annealing Cooling Schedules in the Context of Dense Image Matching*. 2017. URL: https://www.scielo.org.mx/pdf/cys/v21n3/1405-5546-cys-21-03-00493.pdf.

[234] Chao Peng Wenbin Hu Huanle Liang. *A Hybrid Chaos-Particle Swarm Optimization Algorithm for the Vehicle Routing Problem with Time Window*. vol. 15, pg. 1247–1270. Apr. 2013. URL: https://www.mdpi.com/1099-4300/15/4/1247.

[235] David L. Woodruff. *A Chunking Based Selection Strategy for Integrating Meta-Heuristics with Branch and Bound*. vol. 34, pg. 499-511. 1999. URL: https://link.springer.com/chapter/10.1007/978-1-4615-5775-3_34.

[236] Stephen C.H. Leung Xiangyong Li Peng Tian. *Vehicle Routing Problems with Time Windows and Stochastic Travel and Service Times: Models and Algorithm*. vol. 125, pg. 137-145. May 2010. URL: https://doi.org/10.1016/j.ijpe.2010.01.013.

[237] Ertan Yakici. *A Heuristic Approach for Solving a Rich Min-Max Vehicle Routing Problem with Mixed Fleet and Mixed Demand*. vol. 109, pg. 288-294. July 23, 2015. URL: https://doi.org/10.1016/j.cie.2017.05.001 (visited on 05/08/2017).

[238] Magdalene Marinaki Yannis Marinakis. *A Hybrid Genetic – Particle Swarm Optimization Algorithm for the Vehicle Routing Problem*. vol. 37, pg. 1446-1455. Mar. 2010. URL: https://doi.org/10.1016/j.eswa.2009.06.085.

[239] Wee-Chong Oon Yongquan Li Andrew Lim. *The Tree Representation for the Pickup and Delivery Traveling Salesman Problem with LIFO Loading*. vol. 212, pg. 482-496. 2021. URL: https://doi.org/10.1016/j.ejor.2011.02.008.

[240] Russell C. Eberhart Yuhui Shi. *Parameter Selection in Particle Swarm Optimization*. vol. 1447, pg. 591–600. 1998. URL: https://link.springer.com/chapter/10.1007/BFb0040810.

[241] Shigenobu Kobayashi Yuichi Nagata. *A Memetic Algorithm for the Pickup and Delivery Problem with Time Windows Using Selective Route Exchange Crossover*. Sept. 2010. URL: https://link.springer.com/chapter/10.1007/978-3-642-15844-5_54.

[242] Wout Dullaert Yuichi Nagata Olli Bräysy. *A Penalty-based Edge Assembly Memetic Algorithm for the Vehicle Routing Problem with Time Windows*. vol. 37, pg. 724-737. Apr. 2010. URL: https://www.researchgate.net/publication/220470593_A_penalty-based_edge_assembly_memetic_algorithm_for_the_vehicle_routing_problem_with_time_windows.

[243] Sanja Petrovic Yuri Bykov. *A Step Counting Hill Climbing Algorithm*. Nov. 2013. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2361694.