A Generalized Vehicle Routing Problem Optimization Study

A semester project for the Combinatorial Optimization course - ECE157, under the supervision of Prof. N. Ploskas.

Christos - Chrysovalantis Paraschakis
Department of Electrical and Computer Engineering
University of Western Macedonia
Kozani, Greece
ece02010@uowm.gr

Dimitrios Charistes
Department of Electrical and Computer Engineering
University of Western Macedonia
Kozani, Greece
ece02031@uowm.gr

***Abstract*—The Generalized Vehicle Routing problem investigated in this project is a Single Commodity Formulation [citation]. This is a study on model optimization for the Single Commodity Formulation problem proposing a custom Branch and Cut algorithm, as well as heuristics. The final architecture of the custom Branch and Cut algorithm is then compared to the simple Branch and Bound algorithm and the state-of-the-art Gurobi solver for a computational analysis on execution time and nodes traversed in increasing problem order.**

I. Introduction

Generalized Vehicle Routing Problems (GVRP) are a set of challenging optimization problems which have been widely studied for nearly two decades. They are applied across logistics and transportation to minimize costs by determining optimal, constraint-compliant routes for fleets. Key applications include last-mile package delivery, food/supply distribution, school bus routing, maintenance technician scheduling, waste collection, and to some extent in telecommunications network design.

The GVRP is a generalization of the Capacitated Vehicle Routing Problem (CVRP) which lies at the heart of distribution management. The literature on the CVRP is quite rich and we refer the reader to Cordeau and Laporte (2006), Cordeau et al. (2007), Laporte (2007) for overviews of the recent progress on this problem.

In this project, we are concerned with the Generalized Vehicle Routing Problem (GVRP) that consists of finding a set of routes for a number of vehicles with limited capacities on a graph with the vertices (customers) partitioned into clusters. Customers are assigned with given demands -through a problem instance generation algorithm-. The total cost of the travel has to be minimized and all demands to be met.

A. Project Objectives and Outcomes

The project's objective is to examine various techniques on heuristic methods, cutting planes and different formulations to find what works best in Generalized Vehicle Routing Problems. To do so the problem will be modeled and optimized in Pyomo to get a solution reference and a computational footprint from the Gurobi solver and then dive deeper into the actual solving process. The problem will be remodeled in gurobiPy in order to import the lower and upper bounds in the custom branch and bound algorithm that will be implemented. Then the custom optimization process using efficient data structures, heuristics and cutting planes will take place. The computational reference to this optimization will be the simple branch and bound algorithm.

II. Literature Review

A. Hernández and Palacios (2025) [1]

Problem: This study addresses the Generalized Vehicle Routing Problem (GVRP) with a specific focus on the complexities introduced by *heterogeneous fleets*. Unlike standard models that often assume identical vehicles, this paper considers practical logistics scenarios where the fleet consists of vehicles with varying capacities and operational characteristics, significantly increasing the difficulty of the routing and cluster-assignment decisions.

Methodology: Acknowledging that exact methods are computationally prohibitive for large-scale instances of this NP-hard problem, the authors proposed the use of advanced metaheuristics. Specifically, they implemented and analyzed **Ant Colony Optimization (ACO)** and **Genetic Algorithms (GA)**. These bio-inspired algorithms were selected for their ability to efficiently navigate the vast search space, effectively handling the dual constraints of selecting exactly one node per cluster while optimizing routes for a diverse fleet.

Contribution: The research broadens the applicability of GVRP models by validating the effectiveness of evolutionary and swarm intelligence heuristics in heterogeneous fleet contexts. The study demonstrates that these metaheuristics provide robust, high-quality solutions that significantly reduce operational costs and environmental impact in complex transportation systems.

B. Pop, Matei, and Sitar (2013) [2]

Problem: The authors noticed that standard Genetic Algorithms (GA) have a specific weakness when solving the

GVRP. While GAs are good at finding a general order of clusters to visit, they struggle to decide exactly which specific node inside the cluster is the best one to pick. **Methodology:** To fix this, they created a "Hybrid" algorithm, where they combined a standard Genetic Algorithm with a local search technique. The Genetic Algorithm handles the "big picture" (the sequence of clusters), while the local search acts as a repair mechanism, constantly refining the route to pick better nodes and shorten the travel distance. **Contribution:** This paper is important because it proved that combining two different methods is more effective than using just one as their hybrid approach found better solutions for many standard test problems compared to previous algorithms that used only Genetic Algorithms or Ant Colony Optimization.

C. VRP in Stochastic Traffic Networks [3]

The paper addresses the Vehicle Routing Problem with Time Windows (VRPTW) within a stochastic traffic network, acknowledging that travel times are often random due to external factors like accidents or congestion. The study models travel time as a random variable following a normal distribution and establishes a multi-objective chance-constrained model.

Proposed Algorithm: The three conflicting objectives are minimizing total delay (maximizing reliability), minimizing expected travel time, and minimizing transportation costs. To solve this, Gao proposes a modified Genetic Algorithm (GA) using natural number coding and a relative priority method for crossover and mutation.

Outcomes: Numerical experiments on a small-scale instance demonstrated that the algorithm could effectively find Pareto optimal solutions, highlighting the trade-off where increasing travel time reliability inevitably increases the expected total travel time.

D. Electric Vehicle Routing Problem [4]

In this paper, the Electric Vehicle Routing Problem with Time Windows (EVRPTW) is investigated. Unlike standard VRPs, this variant must account for battery capacity limits, the selection of charging stations, and charging duration.

Proposed Algorithm: The authors introduce a Mixed Integer Linear Programming (MILP) model that specifically incorporates a "linear charging" policy, where the charging time is proportional to the amount of electricity required, rather than a fixed duration or full battery swap. The problem is solved exactly using the AMPL/CPLEX solver.

Outcomes: Computational experiments using modified Solomon's VRPTW instances (converting some customers to charging stations) validated the model. The results confirmed that the linear MILP formulation could successfully optimize charging schedules and routing to maximize vehicle utilization while adhering to time window constraints.

E. Open Vehicle Routing Problem [5]

This paper explores the Open Vehicle Routing Problem with Time Windows (OVRPTW), a variant relevant to third-party logistics where vehicles do not return to the depot after the final delivery.

Proposed Algorithm: To solve this NP-hard problem, the authors propose an Improved Variable Neighborhood Search (VNS) algorithm. The method initializes solutions using a route construction heuristic that prioritizes customers with earlier time windows. The VNS employs three neighborhood operators (Swap, Insertion, and 2-Opt) and integrates a Variable Neighborhood Descent (VND) procedure for local search. The algorithm was tested on Solomon's R1 and RC1 datasets (100 nodes) and compared against the existing IMPACT algorithm.

Outcomes: The results indicated that the improved VNS significantly outperformed the benchmark in terms of total travel distance (reducing it by $\simeq 33\%$), although it utilized a slightly higher number of vehicles.

III. Problem Analysis

The GVRP is defined on a directed graph $G = (V, A)$, where the vertices are grouped into disjoint subsets, the clusters. The defining characteristic of this formulation is the 'generalized' visitation rule, dictating that the demand of an entire cluster is satisfied by servicing a single node within it. This transforms the problem into a combined location-routing challenge, which means that there are two simultaneous processes.

The first process is the selection of the node (customer) to be visited for each cluster, and the second process is the routing structure configuration between the chosen nodes of each cluster. The objective is to minimize total traversal costs across the set of arcs A, while satisfying the topological constraints of the cluster partition, for exactly one node visitation per cluster.

A. Single Commodity Problem Statement

The formulation of the GVRP that will be examined and optimized in this project is the Single Commodity Formulation [6] proposed by Gavish and Graves (1978). The problem is modeled as a pickup situation, in order to examine the strengthened capacity bound that will be further discussed in I.C. This formulation is set upon cluster demands but with each node (customer) of the cluster having a distinct cost. All vehicles have the same capacity Q and the demand of any cluster is at most Q. The exact problem instance generation on number of customers, clusters, vehicles and capacity will be discussed in section IV.

B. Mathematical Formulation

The mathematical formulation is based on a directed graph and uses a binary variable x_{ij} defined for every $(i, j) \in A$, which equals 1 if arc $(i, j) \in A$ where A is the set of arcs, is traversed by a vehicle, and 0 otherwise. A continuous variable $f_{ij} \geq 0, \forall (i, j) \in A$ indicates the amount that the vehicle carries from vertex i to vertex j. Its definition is crucial for the commodity balance and to constraint the demand serving. In this formulation it is very important to define the arcs that are entering and leaving a cluster. For that purpose the notation $\delta(S)$ is defined as $\delta(S) = (i, j) \in A; (i \in S \ \& \ j \notin S) \vee (i \notin S \ \& \ j \in S)$. It corresponds to the set of arcs entering or leaving the subset

$$
\begin{aligned}
(\mathcal{F}_1)\quad & \text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} && && (1)\\
& \text{subject to} \\
& x(\delta^+(C_k)) = 1 && \forall k \in M \setminus \{0\} && (2)\\
& x(\delta^-(C_k)) = 1 && \forall k \in M \setminus \{0\} && (3)\\
& x(\delta^+(C_0)) = K && && (4)\\
& x(\delta^-(C_0)) = K && && (5)\\
& x(\delta^+(i)) = x(\delta^-(i)) && \forall i \in V && (6)\\
& f(\delta^+(i)) - f(\delta^-(i)) = \frac{1}{2}q_{\alpha(i)}(x(\delta^-(i)) + x(\delta^+(i))) && \forall i \in V \setminus \{0\} && (7)\\
& q_{\alpha(i)} \leq f_{ij} \leq (Q - q_{\alpha(i)})x_{ij} && \forall (i,j) \in A && (8)\\
& x_{ij} \in \{0,1\} && \forall (i,j) \in A. && (9)
\end{aligned}
$$

S, whereas in the case of the problem this is for every cluster C_k in the set of clusters M, except the cluster 0 of the depot. The formulation is presented above.

C. Variables and Constraints Analysis

In the formulation below (1), is the objective function corresponding to the minimization problem. The c_{ij} is the cost of the arc (i,j) and it is counted only if the variable x_{ij} arc is 1 thus traversed. The purpose of the solver is to minimize that sum as close possible to the relaxed problem optimal solution objective value (will be discussed later).

Based on the analysis of the notation $\delta(S)$, constraints (2) and (3) define the requirement of the vehicle to visit a cluster C_k exactly one time, thus having one entering arc (2) and a one leaving arc (3).

Constraint (4) and (5) determine that exactly K vehicles will be used. This is achieved by forcing the leaving and entering arcs of C_0 depot cluster to be exactly K each.

Constraint (6) imposes the conservation of the physical route at the level of each individual node (customer). Ideally, it is the rule that ensures vehicles do not get stuck at a node or break the flow of casuality within a cluster. Mathematically, this equation states that the number of vehicles entering node i, $x(\delta^-(i))$ must equal the number of vehicles leaving node i, $x(\delta^+(i))$. Since x is a binary variable, there are only two valid states for any node i.

It might be assumed that Constraints (2) and (3) are sufficient. However, without Constraint (6), the model allows for a physical impossibility known as "Inter-Cluster Teleportation". Constraint (6) prevents this by forcing the specific node that receives the vehicle to be the exact same node that sends the vehicle to the next destination.

Constraint (7) serves as the commodity flow balance equation, effectively linking the binary routing variables (x) with the continuous cargo variables (f). In this pickup scenario, it mandates that the load leaving a customer node must exceed the load entering it by exactly the amount picked up. This constraint is non-redundant to the degree constraints (2) and (3); while the latter ensure that clusters are entered and exited, they fail to prevent isolated subtours (disconnected loops such as $C_1 \to C_2 \to C_1$). By enforcing flow accumulation from a source (the depot) to a sink, Constraint (7) renders such isolated loops mathematically impossible. Furthermore, it works in tandem with Constraint (8) to explicitly couple routing decisions with capacity limits, ensuring the vehicle's accumulated load never exceeds Q.

While Constraint (7) determines the flow for customer nodes, the depot (Node 0), is excluded from this flow balance formulation. It requires a distinct formulation to satisfy the conservation of mass across the entire network. For any customer node $i \in V \setminus \{0\}$, the vehicle arrives with a specific load, collects the demand $q_{\alpha(i)}$, and departs. This relationship is expressed as:

$$f(\delta^+(i)) - f(\delta^-(i)) = q_{\alpha(i)} \quad \text{(if visited)} \qquad (10)$$

This implies a net positive flow generation at every visited customer.

However, applying this same logic to the depot leads to a contradiction. The vehicle departs empty from the depot (load = 0) and returns carrying the cumulative demand of all visited clusters. If the standard constraint (Eq. 7) were applied to the depot, where the local demand is zero, it would enforce:

$$
\begin{aligned}
f(\delta^+(0)) - f(\delta^-(0)) &= \frac{1}{2}q_{\alpha(0)}\big(x(\delta^-(0)) + x(\delta^+(0))\big)\\
&\implies f(\delta^+(0)) - f(\delta^-(0)) = 0\\
&\implies f(\delta^+(0)) = f(\delta^-(0))
\end{aligned}
\qquad (11)
$$

Since the vehicle must depart the depot empty ($f(\delta^+(0)) = 0$), this condition would force $f(\delta^-(0)) = 0$, implying that the vehicle returns empty. This is not in line with the fundamental pickup objective of the problem.

To resolve this, the flow balance equation for the depot must account for the absorption of the total system load. The sum of flow differences across all nodes in the network must equal zero:

$$\sum_{i\in V\setminus\{0\}} \underbrace{(f(\delta^+(i)) - f(\delta^-(i)))}_{+\text{Total Demand}} + \underbrace{(f(\delta^+(0)) - f(\delta^-(0)))}_{\text{Depot Net Flow}} = 0 \tag{12}$$

To balance the positive flow generated by the customers, the depot must exhibit a negative net flow exactly equal to the total collected demand:

$$f(\delta^+(0)) - f(\delta^-(0)) = -\sum_{k\in M} q_k \tag{13}$$

Given that $f(\delta^+(0)) = 0$, this simplifies to $f(\delta^-(0)) = \sum q_k$, correctly ensuring that the vehicle returns to the depot fully loaded. Equation 13 will be added as a constraint in the model.

Constraint (8) is the Strengthened Validity Inequality. It physically links the routing and flow variables, and it "tightens" the mathematical model to help the solver run faster. The lower bound represents the minimum load condition for the pickup problem. If a vehicle traverses arc (i,j), it implies the vehicle has just serviced (visited) node i. Since this is a Pickup problem, the vehicle must have picked up the demand $q_{\alpha(i)}$ at node i. Therefore, the load on the vehicle immediately after leaving node i (f_{ij}) must be at least equal to that pickup amount. The upper bound represents the Strengthened Capacity Limit relative to the vehicle. If the vehicle leaves node i carrying a load, and it just used up $q_\alpha(i)$ of its "virtual space", we can mathematically tighten the upper bound. This strengthened bound is theoretically better than the simple $0 \leq f_{ij} \leq Q$ because it tightens the LP relaxation solution making the resolution to complete faster and also prunes a tree earlier in the process.

IV. Problem Instance Generator

To evaluate the performance of the proposed algorithms, a custom instance generator was developed in Python following the instructions that were given during the labs of the course. The generator constructs problem instances based on a grid topology, ensuring diverse spatial distributions and cluster configurations. The generation process follows a four-step sequential logic: grid initialization, cluster formation, demand assignment, and graph construction.

A. Topology Initialization

The problem space is defined as a square grid of size $grid\_size \times grid\_size$. Also a set of V vertices is generated, where:

- The Depot (Node 0) is placed at a random coordinate (x_0, y_0) within the grid.
- $V - 1$ customer nodes are placed at random unique coordinates (x_i, y_i) on the grid, ensuring that they do not overlap with the depot or other customers.

B. Cluster Formation Strategy

A key feature of this generator is the "Strategic Proximity" clustering method, it is designed so that the clusters make sense topologically imitating real life scenarios. The algorithm creates $num\_clusters$ or as is referenced in the paper above M clusters using a seed-based approach:

1) Seed Selection: The algorithm first selects up to 5 strategic "anchor" points to ensure spatial spread. These points correspond to the four corners of the grid and its center. After these 5 seeds, it keeps selecting seed points randomly until the $num\_clusters$ is matched.

- If $M \leq 5$, the first M strategic points are used as cluster seeds.
- If $M > 5$, all 5 strategic points are used, and the remaining $M - 5$ seeds are placed randomly on the grid.

2) Customer Assignment: Once all seeds are established, customers are assigned to clusters based on proximity (Manhattan distance). To ensure valid GVRP instances where no cluster is empty, a priority assignment is performed: 1. **Guaranteed Assignment:** For each cluster seed, the closest unassigned customer is identified and locked to that cluster. 2. **Proximity Filling:** The remaining unassigned customers are then assigned to the cluster of their nearest seed.

C. Demand and Connectivity

Demand Generation: In alignment with the Single Commodity formulation, demand is assigned at the cluster level instead of at the individual node level. Each cluster C_k (where $k \in M \setminus \{0\}$) is assigned a random integer demand d_k such that $1 \leq d_k \leq Q$, where Q is the total vehicle capacity.

Graph Construction: The generator constructs the edge set A by calculating the traversal cost between all pairs of nodes belonging to *different* clusters. The cost c_{ij} between two nodes i and j is calculated using the Manhattan (L_1) distance:

$$c_{ij} = |x_i - x_j| + |y_i - y_j| \tag{14}$$

Edges between nodes within the same cluster are excluded, as the problem definition requires exactly one visit per cluster.

V. Optimization in Pyomo

A. Overview

To model the GVR problem and solve it using Gurobi, the Pyomo library is utilized. Pyomo (Python Optimization Modeling Objects) is an open-source Python library used to formulate optimization models. It acts as the interface that interprets the mathematical formulation defined in III in a form the solver will understand. Using the Pyomo library, the sets, parameters, variables, constraints, and the objective function are defined to build the problem's model.

To build the model a function *build_gvrp_model* is defined that takes as arguments the data extracted from the problem instance file (see section for data extraction). These arguments is presented in the table below.

Argument	Definition
`N` (set)	Total nodes in the graph.
`K` (scalar)	Total number of vehicles at the depot.
`Q` (scalar)	Maximum capacity of a single vehicle.
`M` (set)	Total number of customer clusters.
q_k (parameter)	A list of demands for every k.
a_i (parameter)	Node-to-Cluster mapping, where i is the cluster.
`arc_list` (set)	A list of tuples (u, v) of all valid directed arcs.
`cost_param` (parameter)	The cost of each tuple(u, v).
`depot_id` (parameter)	ID assigned to the depot node (0).
`cluster_nodes` (parameter)	A dictionary mapping cluster k to nodes.

TABLE I
`BUILD_GVRP_MODEL` FUNCTION ARGUMENTS.

B. Sets

From these arguments, sets are the first to be defined, as they define the dimensions of the instance problem, using $pyo.Set()$. The set of all vertices in the graph, indexed from 0 to $N-1$, where 0 represents the depot is defined in Pyomo as `model.V`. A subset of V containing only the customer locations, defined as $V \setminus \{0\}$. This is defined to be later used in the customer specific constraints, such as flow conservation. It is defined as `model.V_cust`.

The set of valid directed edges (i, j) connecting nodes in the graph is defined as `model.A` with a dimension of 2 through the arc_list.

The set of all cluster indices $\{0, 1, \ldots, M\}$, where 0 corresponds to the depot's cluster. Defined as `model.C`. The set of clusters requiring service, defined as $\{1, \ldots, M\}$.

C. Parameters

The static input data is stored in Pyomo parameters, which can be scalars or indexed arrays. They are defined using the object $pyo.Param()$

A parameter indexed by the set A, representing the travel cost or distance associated with traversing arc (i, j). Defined as `model.c`.

A parameter indexed by V, mapping each node i to its respective cluster ID. Defined as `model.a`. A parameter indexed by C, representing the demand quantity required by cluster k. Defined as `model.q_cluster`.

A scalar parameter defining the maximum load capacity of a vehicle. Defined as `model.Q`. A scalar parameter defining the fixed number of available vehicles at the depot. Defined as `model.K`. A calculated scalar parameter representing the sum of demands across all customer clusters is defined. This parameter is crucial for formulating the constraint 13 that was extracted in section IIIto account for the depot flow balance. Defined as `model.D_total`.

D. Variables

The Pyomo model explicitly defines the decision variables using strict mathematical domains. These definitions guide the Gurobi solver in distinguishing between discrete combinatorial choices and continuous system states.

1) Routing Variables (x_{ij}): The primary decision variable represents the selection of arcs in the transport network.

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \tag{15}$$

Mathematically, x_{ij} equals 1 if a vehicle travels directly from node i to node j, and 0 otherwise. This variable is declared with the domain `pyo.Binary`.

When Pyomo interfaces with Gurobi, these variables are flagged as binary integers. Gurobi treats the problem as a Mixed-Integer Linear Program (MILP). It enforces integrality on x_{ij} using its internal Branch-and-Cut algorithms, ensuring that in the final solution, every arc is either fully selected or fully ignored.

2) Commodity Flow Variables (f_{ij}): The secondary variable tracks the accumulated demand (load) carried by the vehicle along a traversed arc.

$$f_{ij} \in \mathbb{R}_{\geq 0} \quad \forall (i, j) \in A \tag{16}$$

This variable represents the flow the vehicle is carrying after visiting node i but before reaching node j. This variable is declared with the domain `pyo.NonNegativeReals`. Pyomo translates these as standard continuous variables bounded by non-negativity constraints (Linear Programming variables). Gurobi processes them using the Simplex method (or Barrier method) within each node of the search tree. Unlike x_{ij}, these variables are never subjected to branching; their values are determined purely by the linear constraints (flow balance and capacity) and the optimal settings of the routing variables.

E. Constraints

The mathematical constraints governing the GVRP are translated into Python functions (rules) and attached to the Pyomo model. To define the constraints the $Pyo.Constraint()$ object is utilized. This section details how each constraint is translated to specific code structures in Pyomo.

1) Cluster Visit Constraints (Eq. 2): The problem requires that exactly one vehicle enters and leaves each customer cluster C_k (where $k \in \{1, \ldots, M\}$).

$$Eq.2 \implies \sum_{i \in V_k} \sum_{j \in V} x_{ji} = 1 \quad \forall k \in C_{cust} \tag{17}$$

The model iterates over the set of customer clusters (*model.C_cust*). For each cluster k, it retrieves the specific list of nodes belonging to that cluster using `model.cluster_nodes[k]`. The constraint sums the binary variables x_{ji} for all arcs entering any node i within that cluster.

2) Depot Flow Constraints (Eq. 4): The fleet size constraint ensures that exactly K vehicles depart from and return to the depot (node 0).

$$Eq.4 \implies \sum_{j \in V_{cust}} x_{0j} = K \tag{18}$$

The rules sum the binary variables associated with arcs leaving or entering the depot. This sum is forced to equal the parameter *model.K.*

3) Routing Flow Conservation (Eq. 6): To ensure route continuity, the number of arcs entering a node must equal the number of arcs leaving it.

$$Eq.6 \implies \sum_{j \in V} x_{ji} = \sum_{j \in V} x_{ij} \quad \forall i \in V \tag{19}$$

The flow conservation rule calculates *sum_in* and *sum_out* for every node i in the graph.

4) Commodity Flow Balance (Eq. 7): For any customer node i, the flow leaving minus the flow entering must equal the demand picked up at that node (if the node is visited).

$$Eq.7 \implies \sum_{j} f_{ij} - \sum_{j} f_{ji} = q_{a(i)} \sum_{j} x_{ij} \tag{20}$$

The rule calculates the demand term dynamically. It sets the Right-Hand Side (RHS) to $0.5 \cdot q_{a(i)} \cdot (\sum x_{in} + \sum x_{out})$. Since a visited node has $\sum x_{in} + \sum x_{out} = 2$, this simplifies to exactly $1 \cdot q_{a(i)}$. If the node is not visited ($x = 0$), the demand term becomes 0, ensuring the equation holds trivially ($0 = 0$).

The net flow out is fixed to $-D_{total}$ (total system demand). In case the node i is the depot (0) the flow conservation constraint has to be met differently through equation 13 as proved in section III. To achieve this an if-else statement is set.

5) Capacity Constraints (Eq. 8): These constraints (eq.8) link the continuous flow variables f_{ij} to the binary routing variables x_{ij}. They ensure that flow is zero on unused arcs and bounded by capacity Q on used arcs.

$$Eq.8 \implies q_i x_{ij} \leq f_{ij} \leq (Q - q_j) x_{ij} \tag{21}$$

In the code they are implemented as two separate constraints (lower and upper bound) for every arc (i, j). If $x_{ij} = 1$, the flow must be at least the demand collected at the starting node i. If $x_{ij} = 1$, the flow cannot exceed the vehicle capacity minus the demand of the destination node j. If $x_{ij} = 0$, the flow is forced to 0.

VI. Read Problem Instance File

To import a problem instance there has to be a function that can read the instance file generated. It has to return all the arguments mentioned in the above section for the model to be build properly.

The *read_data_gvrp* function serves as the data extraction layer for the problem model to be built. Its primary role is to parse the raw instance file and transform it into the specific data structures (dictionaries, lists, and scalar parameters) required by the Pyomo model build function, *build_gvrp_model*.

A. Operational Logic

The function processes the input file in three distinct steps to handle the header, node definitions, and graph topology.

The function first reads the global parameters defining the problem scope. It extracts the number of nodes (V), clusters (M), and vehicles (K), as well as the vehicle capacity (Q) from the first line. These scalars establish the bounds for the model's sets and in extend the problem's dimensionality later to be gradually upscaled in the computational analysis.

The raw data identifies nodes by grid coordinates (x, y). However, as we saw earlier the optimization model requires sequential integer indices $(0, \ldots, N-1)$. The function utilizes a hash map (*node_id_map*) to assign a unique integer ID to every coordinate pair read from the file.

Simultaneously, it builds the *cluster_nodes* dictionary. As nodes are parsed, their new integer IDs are appended to the list corresponding to their cluster ID. It explicitly checks for the cluster ID 0 to locate and store the *depot_node_id*, ensuring the depot is correctly distinguished from customer nodes later.

After detecting the "Arcs:" keyword, the function parses the network topology. It translates the raw start/end coordinates $(x_1, y_1) \rightarrow (x_2, y_2)$ into the integer indices established in the previous phase. The valid connections and their costs are stored in *arc_list* and *cost_param*, respectively.

B. Integration with Model Construction

The output of *read_data_gvrp* is engineered to map directly to the arguments of *build_gvrp_model*, effectively decoupling data parsing from mathematical modeling.

The function abstracts away the spatial geometry of the grid. By converting (x, y) coordinates to simple integers, it allows *build_gvrp_model* to define the set V as a simple range, simplifying the indexing of variables x_{ij} and f_{ij}.

The constraints *visit_in* and *visit_out* require summing variables over all nodes within a specific cluster. The *read_data_gvrp* pre-calculates this grouping in the *cluster_nodes* dictionary. This allows the model to iterate directly over pre-grouped lists rather than searching the entire node set for cluster members during constraint generation.

The returned lists *q_cluster* (demands) and *a* (cluster assignments) are formatted to be passed directly into *pyo.Param* constructors, ensuring O(1) access time during the definition of flow balance constraints.

VII. GurobiPy Model

A. Model Overview

To implement a custom Branch-and-Bound (B&B) algorithm, we transitioned from the high-level modeling library `Pyomo` to the native solver interface `gurobipy`. This decision was driven by the specific control requirements of the algorithm compared to the abstraction provided by Pyomo.

Pyomo is an algebraic modeling language designed to decouple the model formulation from the solver. Its standard workflow is "Define Model $\rightarrow$ Send to Solver $\rightarrow$ Receive Results". While excellent for solving static instances, it poses significant limitations for a custom B&B algorithm:

1) **Low-Level State Access:** The B&B algorithm relies on *Warm Starts*—using the Simplex basis (VBasis/CBasis) of a parent node to speed up the child node's solution. `gurobipy` provides native access to these attributes (`model.getAttr("VBasis")`), whereas retrieving

and injecting basis information in Pyomo is complex and inefficient.

2) **Dynamic Bound Updates:** In our algorithm, branching is implemented by changing the variable bounds (e.g., $LB = 1$ or $UB = 0$). `gurobipy` allows these updates in $O(1)$ time followed by a `model.update()`, whereas Pyomo would require rebuilding parts of the model instance.

The function `gvrp_model_gurobi` constructs the problem specifically for the B&B process. The implementation logic is analytically distinct from a standard MILP definition.

B. Variables

The algorithm operates on linear indices rather than semantic tuples (e.g., (u, v)). The variables are flattened into a single list, `vars_list`, of size $2|A|$:

- Indices 0 to $|A| - 1$: The routing variables x_{ij}.
- Indices $|A|$ to $2|A| - 1$: The commodity flow variables f_{ij}.

A critical aspect of the custom B&B is that the solver must not enforce integrality itself.

All variables are initialized with `vtype = GRB.CONTINUOUS`. If we defined x_{ij} as `GRB.BINARY`, Gurobi would treat the `model.optimize()` call as a black-box instruction to solve the entire integer problem using its own internal algorithms (similar to the PyOmo modeling).

By setting them to continuous, we restrict Gurobi to solving the LP Relaxation (Simplex Method) at each step. This gives our Python script full control over the branching tree and cutting planes.

Since the variables are defined as continuous, the solver may return fractional values (e.g., $x_{ij} = 0.6$). The *integer_var* is set to true for arc variables and false for flow as we want to branch only on arc variables.

In the GVR problem we are solving the customer demands (q_k) are generated as integers. Since the flow is simply the accumulation of these integer demands along a valid path, if $x_{ij} \in \{0, 1\}$, then f_{ij} is guaranteed to be an integer value. Specifically, for any customer node i visited by the vehicle, the flow conservation constraint is given by equation 20

For any customer node i visited by the vehicle, the flow conservation constraint is given by equation 20. Since the path is simple and fixed, there is exactly one incoming arc (u, i) and one outgoing arc (i, v). The equation simplifies to:

$$f_{iv} - f_{ui} = q_{a(i)} \implies f_{iv} = f_{ui} + q_{a(i)} \tag{22}$$

When the vehicle departs from the depot the accumulated load is 0.

$$f_{v_0,v_1} = 0 \tag{23}$$

So the flow on the first arc is an integer.

For the n-th node in the sequence, the incoming flow f_{v_{n-1},v_n} is an integer ($f_{in} \in \mathbb{Z}$) based on the chain reaction the base case initiated. Using equation 22, the flow on the outgoing arc (v_n, v_{n+1}) is:

$$f_{v_n,v_{n+1}} = f_{v_{n-1},v_n} + q_{a(v_n)} \tag{24}$$

We are given that the demand parameter $q_{a(v_n)}$ is an integer input ($q \in \mathbb{Z}$). By the closure property of integers under addition, $f_{v_n,v_{n+1}} \in \mathbb{Z}$.

By mathematical induction, since the flow starts at an integer value (0) and changes only by integer increments (demands) at each node, f_{ij} must remain an integer for all arcs (i, j) in the solution. Therefore, enforcing integrality on x implicitly enforces valid discrete values for f, making branching on f redundant and especially computationally wasteful. The flow on (i, j) is fractional only if the arc (i, j) is fractional instead of binary as implied by the flow balance rule. If the arc (i, j) becomes integer the flow will be too. This implication supports the task of only checking the arch variable on integrality.

The integrality is enforced externally by the custom Python loop:

1) After solving a node, the algorithm scans `integer_var`.
2) It calculates the fractionality using $|x_{ij} - \text{round}(x_{ij})|$. If a violation is found for the x variables, the algorithm splits the problem space by modifying the variable's bounds in the child nodes:
   - **Left Child:** Forces x_{ij} to 0 by setting $UB_{ij} = 0$.
   - **Right Child:** Forces x_{ij} to 1 by setting $LB_{ij} = 1$.

This process repeats until the LP relaxation naturally yields integer values (0.0 or 1.0) for all x_{ij}, at which point the solution is accepted.

The flow variables are explicitly excluded from the branching process (`integer_var[f_idx] = False`) and remain continuous throughout the entire search. The flow variables are mathematically dependent on the routing variables. Once the topology (the binary x values) is fixed, the constraints $f_{ij} = f_{ji} + q_j$ determine the exact flow values.

Since the decision variables are stored in a single list (`vars_list`) to facilitate the Branch-and-Bound indexing, we had to define the helper functions, `get_x(u, v)` and `get_f(u, v)`. These functions utilize the `arc_to_idx` hash map to retrieve the specific continuous variable of arc (u, v) from the linear array.

C. Constraints

Instead of defining the constraints as abstract rules applied to sets as in Pyomo, gurobiPy forces the construction of linear constraints. This requires a specific mechanism to map graph topology to the 1-dimensional variable structure as explained in the above section. The constraints are added to the model instance using `model.addLConstr()`.

VIII. Custom Branch and Bound Algorithm

A. Overview

A custom Branch and Cut algorithm was developed to solve the GVRP. While Gurobi is very powerful as a solver and it gave as the optimal solution of our problem in very little time, it functions as a "black box" hiding the internal decision-making process. By building our own algorithm, we gain full control over the optimization strategy, in this implementation Gurobi is used only to solve the simple linear equations (LP relaxation) at each step. All high-level decisions such as which variable to branch on, how to manage the search tree, and when to stop searching are handled by our custom Python framework.

The original script of the algorithm that we were given utilized a simple Depth-First Search (DFS) strategy. However, to handle the complexity of the GVRP and get better times when running it, the final architecture evolved into a Best-First Search (BFS) approach equipped with warm-start capabilities and dynamic cut generation.

B. Solving Process

The core solving loop iteratively processes nodes from the search tree until the optimality gap closes (basically until the Global Lower Bound meets the Global Upper Bound). The process for each node is defined as follows:

1) **Node Selection:** A node is extracted from the open set (Heap) based on the Best-First strategy. To ensure deterministic behavior and handle ties in objective values, nodes are selected using a tuple of `(Objective, CreationCounter)`.
2) **Relaxation with Warm Start:** The Linear Programming (LP) relaxation of the node is solved. To accelerate this step, we utilize *Warm Starts*. By extracting the Simplex basis vectors (`VBasis` and `CBasis`) from the parent node and loading them into the child node's model attributes, the solver avoids restarting the Simplex algorithm from scratch, significantly reducing computational time per node.
3) **Dynamic Cut Separation:** If the relaxed solution is feasible but fractional, the algorithm checks for violated valid inequalities (specifically Capacity Cuts). If cuts are found, they are added to the model, and the relaxation is *re-optimized* immediately to tighten the bound before branching (See Section IX).
4) **Pruning:** The node is discarded (pruned) if:
   - *Infeasibility:* The LP relaxation has no solution.
   - *Bound Pruning:* The objective value of the relaxed solution (Z_{LP}) is greater than or equal to the current Global Upper Bound (UB): $Z_{LP} \geq UB - \epsilon$.
5) **Integrality Check:** If the solution satisfies all integrality constraints, it is treated as a candidate solution. If its objective value is better than the current Global Upper Bound ($Z_{LP} < UB$), the Global UB is updated, and the incumbent solution is stored.
6) **Branching:** If the solution remains fractional and cannot be pruned, the algorithm selects a branching variable using the "Most Fractional" strategy:

   $$x_{branch} = \arg\max_i |x_i - 0.5| \tag{25}$$

   Two child nodes are created by imposing standard binary branching constraints:
   - Left Child: $x_{branch} \leq \lfloor x^*_{val} \rfloor$ (Fix to 0)
   - Right Child: $x_{branch} \geq \lceil x^*_{val} \rceil$ (Fix to 1)

C. Best First Search Technique

1) Initial Approach: Depth-First Search (DFS): Our initial implementation utilized a `deque` (stack) data structure, enforcing a LIFO (Last-In-First-Out) traversal. While DFS is memory efficient and often discovers *feasible* integer solutions quickly by diving deep into the tree, it suffers from "getting lost" in suboptimal sub-trees. It frequently expends computational resources exploring nodes with poor lower bounds that are far from the optimal solution.

2) Improved Approach: Best-First Search (BFS): To minimize the total number of nodes explored (N), we transitioned to a Best-First Search strategy [7]. This method prioritizes the exploration of the node with the most promising *potential*—specifically, the lowest LP objective value in a minimization context. To achieve that a min-heap data structure is utilized.

After the creation of the left and right child nodes explained in section VIII-B they are pushed in the heap structure. The *heappush* process re-arranges the *minHeap* so the nodes are inserted correctly. The decision variable for the *minHeap* sorting is the objective value of each node. This objective value is what the LP relaxation promises for each child's search tree (Simplex solution). Basically, the process begins by inserting the node at the top of the tree and compares its objective value with the nodes already in the heap. If the value is higher it pushes the node down then tree until the *minHeap* condition is met again.

So every time a node is popped from the *minHeap* data structure to be explored it will always be the one with the lowest objective value. At every instance of the problem's solving process, the nodes stored in the *minHeap* are the best case scenario (bound -till then) for the objective value of each node's search tree exploration. So the lowest objective value among these nodes will be the new global lower bound for the problem's current solving step. This ensures that the algorithm always expends computational effort on the branch that mathematically holds the potential for the best solution, rather than just following a depth path.

This approach will speed up the gap minimization between the lower and upper bound and might terminate the solving process before exploring each node down the line, as the next promised objective value of the *heappop* process will hit the upper bound of the current best integer solution concluding that this solution contains the optimal integer objective value.

IX. Cutting Planes

A. Overview

In the context of MILP, cuts are linear inequalities used to tighten the relaxation of the problem space. A *cut* is a constraint added to the model that satisfies two properties:

1) It is violated by the current fractional solution x^*.
2) It is valid for all feasible integer solutions.

By iteratively adding these cuts and re-optimizing, the LP relaxation bound is tightened (the lower bound increases in minimization problems), pruning the search before branching is required.

B. Theoretical Formulation

The cut implemented in this algorithm is the *Rounded Capacity Inequality* [9], which is a fundamental valid inequality for Generalized VRPs.

The cut is based on the logic that if a subset of customers S has a total demand $D(S)$, a minimum number of vehicles $r(S)$ is required to serve them. Since every vehicle entering the set S must also leave it, the number of arcs crossing the boundary of set S must be at least twice the number of these vehicles.

In section III the notation $\delta(S)$ was thoroughly explained as the set of arcs with exactly one endpoint in S (the cut set). The mathematical formulation of the constraint for a set $\delta(S)$ is:

$$\sum_{(i,j)\in\delta(S)} x_{ij} \geq 2 \cdot r(S) \tag{26}$$

Where $r(S)$ is the minimum number of vehicles required, calculated as:

$$r(S) = \left\lceil \frac{\sum_{k\in S} q_k}{Q} \right\rceil \tag{27}$$

Here, q_k represents the demand of cluster k and Q is the vehicle capacity. The ceiling function $\lceil\cdot\rceil$ ensures that if for example the demand is $1.1\times Q$, at least 2 vehicles are required.

C. The Problem of Exponential Constraints

The limitation of Capacity Cuts is that they are defined for every possible subset of customers $S \subseteq C \setminus \{0\}$. For a problem with $|C|$ clusters, there are $2^{|C|} - 2$ possible subsets. It is computationally hard to add all these constraints to the model build section even for small problems.

To address this, the algorithm imposes the cut dynamically *(Dynamic Cut Separation)*. The constraints are initially omitted. After solving the LP relaxation at a node, the algorithm searches for a specific set S where the constraint is violated—i.e., where the current fractional flow into S is insufficient to support its demand.

D. Code Implementation

The separation logic is implemented in the function `separate_capacity_cuts` and called inside the branch and bound function's main loop. Instead of solving the exact separation problem, the implementation utilizes a heuristic that checks for violations on subsets of single clusters and pairs of clusters.

1) Weight Aggregation: The algorithm first extracts the current fractional solution from the Gurobi model. It builds an adjacency matrix `cluster_edge_vehicles` representing the flow between clusters. This simplifies the graph, as the flow between individual nodes u and v is now a flow between their respective clusters $a(u)$ and $a(v)$.

2) Separation of Subsets: The code iterates through specific subsets S to check for violations.

Singleton Cuts ($S = 1$)**:** The algorithm iterates through every individual cluster k. It calculates the flow crossing the boundary of k, defined as `lhs`, and the required capacity `rhs`.

- **LHS Calculation:** The function `x_delta_S` sums the variable values x_{ij} where one node is inside S and the other is outside.
- **RHS Calculation:** It computes $r(S)$ as $\lceil q_S/Q \rceil$ multiplied by 2.
- **Violation Check:** If $LHS < RHS - \epsilon$, the cut is violated and added to the list.

Pairwise Cuts ($S = 2$)**:** The algorithm then repeats this process for all pairs of clusters $\{k, l\}$. It sums their combined demand to check if the flow entering/leaving the pair is sufficient. This detects situations where two clusters individually satisfy the capacity rules but, when grouped, require more vehicles than the current fractional solution provides.

3) Cut Injection: The violated cuts are sorted by the magnitude of their violation to prioritize the most effective constraints. The top cuts (limited by `MAX_CUTS_PER_NODE`) are defined as Gurobi linear expressions (`gp.LinExpr`) and inserted into the model using `model.addLConstr`. The solver is then updated with `model.update()` to resolve the node with the tightened bounds.

X. Heuristics

A. Utilization

To speed up the solving process of the branch and bound algorithm, heuristics methods are implemented. The branch and bound algorithm analyzed in section VIII initializes the upper bound (for minimization problem) to infinite. So, it is possible for the branch and bound algorithm to find an integer solution that is far from the optimal integer one, taking longer for the solving process to complete. This is because the upper bound will not be as strict relative to the optimal integer objective value.

The heuristic methods have to improve the gap between the initial upper bound and the final integer solution, under a set of rules for its termination. This is derived after the optimal integer solution is found and compared to the heuristic's objective value to determine its performance. Mathematically, it is expressed as the below percentage minimization. The minimization is not **global** and only applies if the conditions R for the heuristic termination are not met yet.

$$\min_k \frac{UB_i - UB_f}{UB_i},\ while\ k \in R \tag{28}$$

The heuristic methods are rarely consistent for different instances as they rely on randomness. That said, the above percentage is pointless to optimize as the more the gap closes, the longer the time it takes for the heuristic to terminate. So the rules defining the termination of the heuristic will be strict.

The purpose of this project is to utilize a solving method and not over-rely on heuristics that will take time to give a solution.

In our problem a relatively good integer solution is accepted for the upper bound of the branch and bound algorithm.

In more detail, the purpose of heuristic methods is to find an integer solution that passes the constraints limitations, so that the branch and bound algorithm can start solving with a known integer objective value. That way every integer solution between this and infinite is cut by bound during branching by comparing the best case objective value of the relaxed problem to this upper bound. Even if a solution is contained at a path it will be cut because a better solution is already found.

The heuristics architecture proposed in this project consists of a myopic heuristic and a meta-heuristic algorithm of neighborhood search [8]. The metaheuristic takes the objective value of the myopic result as input and improves on it.

B. Objective Function Evaluation

The evaluation of a solution's quality is handled by the `getAssignmentCost` function. This function calculates the total routing cost for a given set of routes, which corresponds to the objective value to be minimized in the GVRP model.

The function iterates through every active route in the solution. For each route, the cost is computed as the sum of the traversal costs between consecutive nodes, including the connections to and from the depot.

C. Feasibility Verification

The `check_feasibility` function ensures that a solution complies to the problem's constraints before it is accepted. This step is important during both the construction phase (Myopic heuristic) and the improvement phase (VNS) as to not accept an integer objective value that will direct the branch and bound solving process to infeasibility.

The function validates the solution against two primary constraints.

1) **Vehicle Capacity:** For every route in the solution, the algorithm sums the demands of the clusters associated with the visited nodes. It verifies that the total load does not exceed the vehicle capacity Q.
2) **Cluster Visit Rule:** The function enforces that every cluster is visited exactly once. It maintains a set of visited clusters while traversing the routes. If a cluster is encountered more than once, or if the final count of unique visited clusters does not equal the total number of clusters M, the solution is not accepted.

If any of these conditions are violated, the function returns `False`, effectively rejecting the move or solution instance.

D. Myopic Heuristic

To generate an initial feasible solution (Upper Bound) for the Branch and Bound algorithm, a constructive heuristic based on the Nearest Neighbor principle is employed. This method is "myopic" (greedy) because it makes the locally optimal choice at each step without considering the global impact on future decisions.

The solution is extracted by filling one vehicle route at a time until all clusters are visited or all vehicles are utilized. U is the set of unvisited clusters, containing all M clusters initially. The algorithm proceeds as follows:

1) **Route Initialization:** A new route is opened from depot (Node 0).
2) **Candidate Evaluation:** The algorithm iterates through every unvisited cluster $C_k \in U$. For each cluster, it checks the capacity constraint:
$$L + q_k \leq Q \tag{29}$$
If the cluster fits, the algorithm calculates the distance c_{ij} to every node $j \in C_k$.
3) **Selection Step:** The algorithm selects the specific node j in cluster C that minimizes the distance from the current node i.
4) **Update:** Node j is added to the route. The current position updates to $i \leftarrow j$, the load updates to $L \leftarrow L + q_{C_k}$, and cluster C is removed from U.
5) **Termination of Route:** The above steps are repeated (except the first) until no remaining cluster satisfies the capacity constraint. The vehicle returns to depot, and the next route begins.

This method effectively handles the two-layered complexity of the GVRP simultaneously: it decides which cluster to visit next *and* exactly which node within that cluster to visit, based purely on proximity. While computationally fast, the lack of look-ahead often results in the final few clusters being geographically isolated, necessitating the subsequent improvement phase.

E. Metaheuristic: Variable Neighborhood Search

Following the derivation of an initial feasible solution with the myopic heuristic, a Variable Neighborhood Search (VNS) metaheuristic is implemented to refine the solution and improve the percentage equation 28 in the branch and bound algorithm. The VNS algorithm explores utilizing two processes: the stochastic shaking, and the deterministic improvement with the local search to escape local optimals.

The algorithm is constrained by two control loops: the outer loop constrained by a maximum number of global iterations, and the inner loop controlled by the neighborhood intensity k. These decision variables will be the termination rules R for the heuristic. In this algorithm S denotes the current solution and $C(S)$ objective value calculated by the assignment cost function explained in section X-B.

1) Neighborhood and Shaking: In the VNS algorithm the first to be defined is the neighborhood based on perturbation intensity. In this implementation, the k neighborhood $N_k(S)$ is defined by the number of random relocation moves applied to solution S. The shaking procedure generates a solution S' according to the current intensity k.

$$S' \leftarrow \text{Shaking}(S, k) \tag{30}$$

The algorithm makes exactly k random moves. A single move consists of:

1) Selecting a random non-empty route and a random node within it.
2) Removes the node and inserts it into a random position in a different target route, provided the `check_feasibility` is true.

As k increases, the difference between S and S' grows, making the exploration of more distant solutions regions possible.

2) Local Search Phase: After the shaking phase, a local search is applied to S' to reach a local optimum S'':

$$S'' \leftarrow \text{LocalSearch}(S') \tag{31}$$

The `local_search_vns` function iterates through all clusters $m \in \{1, \ldots, M\}$. For each cluster, the algorithm evaluates: the nodes in that cluster to choose from, and then their insertion positions across all routes.

3) Acceptance and Update: The improved solution S'' is compared to the current global best S. The update rules for the solution and the neighborhood iterator k are defined as follows:

$$(S, k) \leftarrow \begin{cases} (S'', 1) & \text{if } C(S'') < C(S) - \epsilon \\ (S, k+1) & \text{otherwise} \end{cases} \tag{32}$$

If S'' improves the objective value, it is accepted as the new solution, and the search intensity is reset to ($k = 1$) to intensify the search around the new optimum for a better solution. If no improvement is found, the solution reverts to S, and k is incremented to apply a stronger relocation in the next step.

4) Termination Rules: To ensure the heuristic provides a warm start to the branch and bound algorithm without consuming excessive computational time as mentioned before, strict termination rules are implemented:

- **Neighborhood Exhaustion (k_{max}):** The inner loop terminates if k exceeds the maximum defined intensity k_{max}. This indicates that the current solution S cannot be improved even after applying the strongest defined perturbation.
- **Maximum Iterations:** The outer loop terminates when the global counter reaches `max_iterations`. This acts as a hard limit on the runtime, ensuring the solver transitions to the exact method regardless of whether local improvements are still possible.

XI. Computational Analysis

A. Computational Process

The finalized structure of the optimization model is presented in the Algorithm 1. The model for the problem instance generated is built through the `gvrp_model_gurobi` function that uses the gurobiPy library. The model, upper and lower bounds and all the variables are extracted. Then the `read_data_gvrp` function is called on its own to extract the problem's parameters that are needed for the heuristics and the cost_matrix dictionary is created through the cost_param and arc_list.

Algorithm 1 GVRP Solving Framework

```
Require: Problem Instance (Data file)
Ensure: Optimal routes and objective value
 1: Step 1: Initialization
 2: $N, K, Q, M, q_k, cost = read\_data\_gvrp(filename)$
 3: $bb, ub, lb, arcs, vars = gvrp\_model(filename)$
 4: $Data = N, K, Q, M, q_k, cost$
 5: Step 2: Heuristics
 6: $S_{myopic}, C_S = myopic(Data)$
 7: if $S_{myopic}$ is feasible then
 8:     $(S_{vns}, C_{vns}) = VNS(S_{myopic}, k_{max}, iter_{max})$
 9:     $UB \leftarrow C_{vns}$ {Set Upper Bound for B&B}
10: else
11:     $UB \leftarrow \infty$
12: end if
13: Step 3: Search
14: $(S_{opt}, C_{opt}) = branch\_and\_bound(UB, Data, ...)$
15: Step 4: Results
16: Report $S_{opt}$, $C_{opt}$, and total runtime
```

The first step of the optimization algorithm is to run the myopic heuristic to find a solution to then be imported to VNS. If a solution is found, the objective value is used for the VNS algorithm as a starting point. The VNS parameter k_max is set to 4 meaning that the maximum number of relocation the algorithm will try is 4. Then the max_iterations parameter is set to 100 as the hard limit for the termination of VNS. Even if the objective value does not improve the algorithm will terminate and the best objective value it holds will be returned in variable best_cost.

A significant note here is that as can be implied from the check_feasibility function, no check happens on how many vehicles are being used. The VNS algorithm might find a solution that uses less than the K vehicles set in the problem instance. In case that happens it means that the problem can be solved with less than the K available vehicles. In that situation the vehicle parameter in the problem's instance file has to be adjusted to the number of vehicles the VNS algorithm found a solution. This is crucial because the constraints 4 and 5 will not permit an objective value like the imputed to branch and bound as upper bound. Even if the number of vehicles needed is less than K, the problem's formulation states that exactly K vehicles must be used.

So the branch and bound algorithm will find only infeasible solution as the objective value it finds will be higher than the upper bound set by VNS and in some cases the lower bound will reach a point where it has a higher value than the upper bound leading to the algorithm's termination. To prevent that the number of vehicles used in VNS is extracted after its execution concluded, by counting the leaving arcs for the depot (pseudo-code XI-A. If it is less than the original K, the variable and the file is modified.

```
1: $K_{active} \leftarrow 0$
2: for all route $r$ in $S_{best}$ do
3:     if length($r$) $> 0$ then
4:         $K_{active} \leftarrow K_{active} + 1$
5:     end if
6: end for
```

B. Instances order generation

The proposed algorithm will be compared with the simple branch and bound without utilizing any heuristics as well as with the state-of-the-art gurobi solver. The problems instances that will be tested will be 5. Sequentially, the grid size and number of customers will be increased starting from 10x10 up to 45x45 and the number of customers and clusters will be linearly transformed based on the grid size. The performance metric will be the solving time and the nodes explored for each problem.

XII. Results

The results are presented in the below figures as well as in the table ref in more detail.

A. Time Comparison

The time graph in Fig. 1 of the custom branch and bound identifies a lack of correlation between the order of the problem and the solving time. It indicates that the order of the problem is not entirely correlated to the solving time but instead the problem's difficulty is what drives the solving time over a number of problems. The same goes for the simple branch and bound although it is observed that the time is almost half the time of branch and cut outlining a major performance outcome for the implementation of cuts. It will be explained more on the Nodes Explored section as that performance metric is removes the instability of the computer's processing time due to other processes. The objective value of each problem that the branch and cut-bound algorithms and Gurobi find are the same respectively.

B. Nodes Explored

The nodes explored was a driving performance metric to determine the performance of the custom branch and bound. Here the branch and bound algorithm with the the cut method implemented shows its strength. As we can see the nodes explored of the branch and cut algorithm are 50% less than the nodes the branch and bound explored without implementing any cuts, but only for the first 3 problems. In the last two problems, the simple branch and bound explored more than twice the nodes of the branch and cut. This shows that the strength of the cut is not linear, but polynomial.

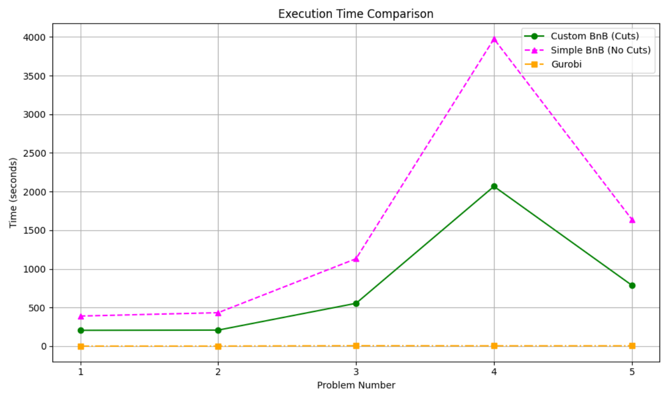

Fig. 1. Time Comparison of Gurobi Solver and Custom Branch and Bound. For 5 problems.

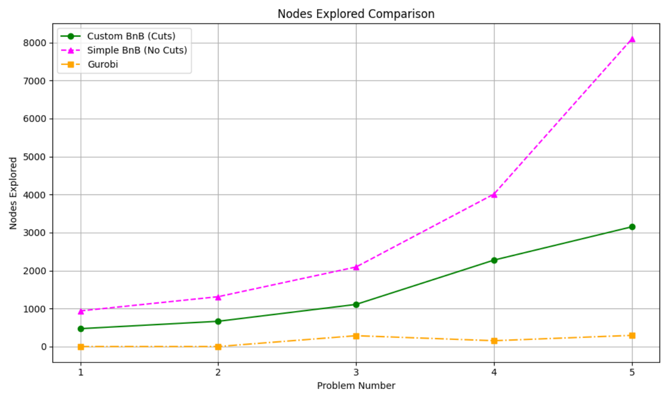

Fig. 2. Nodes explored Comparison between Gurobi Solver and Custom Branch and Bound. For 5 problems.

TABLE II
RESULTS OF SOLVING METHODS

Pr.	Time (s)			Nodes Expl.		
#	BnC	BnB	Gur.	BnC	BnB	Gur.
1	205.11	389.70	1.08	471	938	1
2	207.94	433.22	1.22	665	1312	1
3	554.59	1131.82	5.55	1109	2094	286
4	2067.19	3975.36	4.09	2275	4012	154
5	787.97	1639.58	4.21	3151	8094	295

References

[1] F. Hernández and W. Palacios, "Heuristic algorithms for generalized vehicle routing problem (H-GVRP)," *Applied and Computational Mathematics*, vol. 11, 2025.
[2] P. C. Pop, O. Matei, and C. P. Sitar, "An improved hybrid algorithm for solving the generalized vehicle routing problem," *Neurocomputing*, vol. 109, pp. 76–83, 2013.
[3] G. Jie, "Model and algorithm of vehicle routing problem with time windows in stochastic traffic network," 2010 International Conference on Logistics Systems and Intelligent Management (ICLSIM), Harbin, China, 2010, pp. 848-851, doi: 10.1109/ICLSIM.2010.5461065.
[4] X. Zuo, Y. Xiao, C. Zhu and M. You, "A Linear MIP Model for the Electric Vehicle Routing Problem with Time Windows Considering Linear Charging," 2018 Annual Reliability and Maintainability Symposium (RAMS), Reno, NV, USA, 2018, pp. 1-5, doi: 10.1109/RAM.2018.8463021.
[5] A. A. N. P. Redi, M. F. N. Maghfiroh and V. F. Yu, "An improved variable neighborhood search for the open vehicle routing problem

with time windows," 2013 IEEE International Conference on Industrial Engineering and Engineering Management, Bangkok, Thailand, 2013, pp. 1641-1645, doi: 10.1109/IEEM.2013.6962688.

[6] Tolga Bektaş, Güneş Erdoğan, Stefan Røpke, (2011) Formulations and Branch-and-Cut Algorithms for the Generalized Vehicle Routing Problem. Transportation Science 45(3):299-316.

[7] Mhand Hifi, Rachid Ouafi, A best-first branch-and-bound algorithm for orthogonal rectangular packing problems, International Transactions in Operational Research, Volume 5, Issue 5, 1998, Pages 345-356, ISSN 0969-6016.

[8] P.C. Pop, C. Pop Sitar, I. Zelina, V. Lupşe, C. Chira, Heuristic Algorithms for Solving the Generalized Vehicle Routing

[9] Pessoa, A., de Aragão, M., Uchoa, E. (2008). Robust Branch-Cut-and-Price Algorithms for Vehicle Routing Problems. In: Golden, B., Raghavan, S., Wasil, E. (eds) The Vehicle Routing Problem: Latest Advances and New Challenges. Operations Research/Computer Science Interfaces, vol 43. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-77778-8_14, Problem, Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844 Vol. VI (2011), No. 1 (March), pp. 158-165

[10] N. Ploskas, "Lectures on Combinatorial Optimization", Electrical and Computer Engineering Department, University of Western Macedonia, source: E-Class, 2025