A heuristic framework for a class of vehicle routing problems

Bruno Oliveira[a,*], Artur Pessoa[b], Marcos Roboredo[b]

[a] *Programa de Pós-Graduação em Engenharia de Produção, Universidade Federal Fluminense, Niterói, RJ, Brazil*
[b] *Departamento de Engenharia de Produção, Universidade Federal Fluminense, Niterói, RJ, Brazil*

Abstract

We propose GenericILS, a general-purpose heuristic framework for generating high-quality upper bounds for Vehicle Routing Problems whose pricing subproblems are modeled, in column-generation-based exact algorithms, as resource-constrained shortest path problems described through Resource Extension Functions (REFs), the standard device for encoding constraints in branch-and-price algorithms. GenericILS reuses REFs implemented for the exact method, without a problem-specific heuristic, illustrated on risk-constrained, simultaneous pickup and delivery, multi-depot, and heterogeneous-fleet variants. The search engine, a Nested Iterated Local Search, handles infeasibility via a lightweight, problem-agnostic penalty estimated from the number of infeasible arcs, evaluated in constant time using forward and backward label structures built in linear time and space, improving on prior quadratic schemes. Experiments on four VRP variants and twelve benchmark sets show best-solution gaps below 0.9% on Branch-Cut-and-Price-relevant instances under a matched time budget, falling to at or below 0.12% under a tenfold larger budget. An open-source implementation is available at `https://github.com/brunomattos1/GenericILS`.

Keywords: Routing, Metaheuristics, Column generation, Resource

*Corresponding author
Email addresses: `bruno_mattos@id.uff.br` (Bruno Oliveira), `arturpessoa@id.uff.br` (Artur Pessoa), `mcroboredo@id.uff.br` (Marcos Roboredo)

extension functions, Iterated local search

1. Introduction

Vehicle Routing Problems (VRPs) are a core class of combinatorial optimization problems arising throughout logistics and distribution planning. State-of-the-art exact algorithms for VRPs rely on Column Generation (CG) and Branch-Cut-and-Price (BCP), in which the pricing subproblem is solved as a resource-constrained shortest path problem via a labeling algorithm Irnich (2008a). Complex side constraints such as capacity, time windows, multiple depots, heterogeneous fleets, and cumulative or pickup-and-delivery structures are encoded through *Resource Extension Functions (REFs)*, which specify how a partial path's resources are updated as it is extended and how two partial paths can be concatenated. This modeling device is central to modern generic branch-and-price solvers Pessoa et al. (2020) and allows a single algorithmic core to handle a wide range of VRP variants without problem-specific re-engineering of the pricing algorithm.

Producing strong upper bounds is a critical component of any BCP algorithm, both to accelerate convergence and to support branching decisions. In practice, however, this typically requires developing a dedicated metaheuristic for each new VRP variant, duplicating much of the problem-specific feasibility logic that has already been implemented through the REFs used in the pricing algorithm; for problems with elaborate side constraints, such as multiple time windows or resource-dependent costs, this REF logic can itself represent a substantial share of the development and validation effort behind the exact algorithm, effort that is discarded when a separate heuristic is built from scratch.

A natural candidate for avoiding this duplication is the existing line of work on *unified* or *multi-attribute* VRP heuristics, which aim to solve broad classes of routing problems within a single metaheuristic engine rather than one bespoke algorithm per variant. The most prominent example is the concatenation-based framework of Vidal et al. (2014), which unifies split and local search procedures across variants by expressing route feasibility

and cost through problem-specific *bounds* on route attributes, evaluated in constant time via a precomputed concatenation scheme. Vidal et al. (2013) present a survey on heuristics for multi-attribute VRPs, and Subramanian et al. (2013) propose a hybrid ILS-RVND with Set Partitioning that solves, within a single algorithm, several VRP variants. These frameworks are effective and have been validated on a wide range of variants, but their generality operates at the level of the *search engine*: the metaheuristic itself is reusable across problems, while the attribute bounds that make a route's feasibility and cost evaluable in constant time still have to be derived and implemented anew, independently, for every variant, in a formalism specific to the heuristic rather than shared with any exact algorithm available for the same problem. As a result, even where such a unified heuristic exists for a given VRP variant, it does not by itself let a researcher reuse the feasibility and cost-propagation logic already encoded in that variant's REFs, the two formalisms are developed independently for every variant.

The framework closest to ours is Irnich (2008b), who also builds on the REF formalism of Irnich (2008a) but generalizes it from depot-anchored paths to arbitrary segments of a single *giant tour* representing all routes at once, so that a local-search move, however it cuts and reconnects the tour, can be evaluated by combining segment REFs; making this evaluation efficient requires precomputing, through a hierarchical scheme of seed points, both segment REFs and their inverses for the entire giant tour. GenericILS pursues the same unification of the heuristic and exact-algorithm formalisms without this generalization: labels remain anchored at the depot, exactly as used by the pricing algorithm, so a variant's exact-algorithm REFs can be reused inside GenericILS exactly as implemented for branch-and-price, with no reformulation to segments required.

This paper addresses this gap by proposing GenericILS, a heuristic framework that operates directly on REFs originally developed for column-generation-based exact algorithms, requiring no additional problem-specific implementation to produce high-quality upper bounds. The user only needs to define a set of resources, following the *Standard Resource* and *Custom Resource* abstractions detailed in Section 2, along with their extension and

concatenation functions, exactly as required by a labeling-based pricing algorithm; we adopt this resource formalism, and its associated notation, directly from the generic Resource Constrained Shortest Path Problem solver of Sadykov et al. (2026). GenericILS then explores the solution space using a *Nested Iterated Local Search (NILS)* metaheuristic, in which an outer ILS controls solution perturbation and acceptance while the local search applied to each candidate is itself structured as an inner ILS, enabling deeper intensification around promising solutions while retaining the simplicity of the classical ILS paradigm. Neighborhood exploration is based on classical λ-interchange moves Osman (1993). At the end of the search, the pool of routes generated along the trajectory is used to build a *Set Partitioning (SP)* model, whose solution refines the final result; we refer to this hybrid algorithm as *NILS-SP*.

Realizing this idea, however, surfaces a challenge intrinsic to reusing REFs designed for an exact algorithm within a local search: most REFs are designed only to certify feasibility, returning a binary feasible/infeasible outcome, without quantifying the extent of a violation. This is sufficient for exact pricing algorithms, which only need to discard infeasible labels, but it is not enough for a local search, which must be able to compare and rank infeasible solutions to escape infeasible regions gradually. To address this without requiring any problem-specific infeasibility measure, we introduce a generic technique that estimates an upper bound on the number of infeasible arcs in a route from the position, within the sequence of REF evaluations along that route, at which infeasibility is first detected. To support the evaluation of neighborhood moves in constant time under this scheme, we further introduce forward and backward label structures that can be built and updated in linear time and space per solution, improving on the quadratic effort required by prior general concatenation schemes Vidal et al. (2014) and yielding substantial runtime gains on instances with long routes.

The framework is evaluated on four VRP variants modeled solely through their REFs: the Risk Constrained Cash-in-Transit VRP (RCTVRP), the Vehicle Routing Problem with Simultaneous Pickup and Delivery (VRP-SPD), the Multi-Depot VRP (MDVRP), and the Heterogeneous Fleet VRP

(HFVRP). Computational experiments show that NILS-SP produces high-quality feasible solutions across all four variants using the same generic search engine, with no variant-specific modification beyond the REF definitions themselves.

The main contributions of this paper are as follows. (i) We propose GenericILS, a heuristic framework that operates directly on Resource Extension Functions, allowing researchers who have implemented REFs for a branch-and-price algorithm to obtain high-quality upper bounds without developing a dedicated heuristic. (ii) We propose NILS-SP, a hybrid algorithm combining a Nested Iterated Local Search with a Set Partitioning post-processing step, instantiated within GenericILS. (iii) We introduce a generic technique for estimating an upper bound on the number of infeasible arcs in a route after a local change, requiring no problem-specific infeasibility measure from the REFs, applicable to classical VRP neighborhoods. (iv) We introduce forward and backward label structures that support constant-time evaluation of neighborhood moves and can be built and updated in linear time and space, improving on the quadratic effort of prior general concatenation schemes. (v) We demonstrate the framework on four VRP variants (RCTVRP, VRPSPD, MDVRP, HFVRP) modeled solely through their REFs, and report computational results against reference heuristic algorithms.

An open-source implementation of GenericILS is available at `https://github.com/brunomattos1/GenericILS`.

The remainder of this paper is organized as follows. Section 2 defines the class of problems the framework targets and introduces the REF notation used throughout, including how infeasibility is quantified from otherwise boolean REFs (Section 2.1). Section 3 describes the NILS-SP algorithm in full: its initial solution procedure and neighborhood structures (Sections 3.2 and 3.3), the perturbation mechanism (Section 3.4; speed-up mechanisms that make neighborhood evaluation efficient are detailed in Section S2 of the Supplementary Material), the adaptive penalty scheme that guides the search between feasible and infeasible regions (Section 3.5), and the Set Partitioning post-processing step that completes NILS-SP (Sec-

tion 3.6). Section 4 then details the generic technique for estimating the number of infeasible arcs introduced by a neighborhood move, applied to each of the neighborhoods introduced in Section 3 in turn. Section 5 reports the computational experiments and discusses the results across the four VRP variants introduced above. Finally, Section 6 concludes the paper and outlines directions for future research.

2. Problem definition and notation

The problems addressed by the proposed framework must satisfy the following two characteristics: all routes can be represented as paths in a single graph with a unique source and a unique sink, and all customers must be visited exactly once.

Let $G = (V, A)$ be a complete directed graph, where V is the set of vertices, including the depot 0 and the customers $v_1, v_2, \ldots, v_n$. Let P be a path in G, represented as $P = (0, v_1, \ldots, v_k)$, and let $A_P = ((0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$ denote the arcs of P. For an arc $a \in A$, let $v^-(a)$ and $v^+(a)$ denote its tail and head, respectively, and let c_a be its cost.

The problem consists of finding a set of feasible routes minimizing total cost, where the cost of a route is the sum of an *arc-cost* and a *resource-cost*. The arc-cost of a path P is $\sum_{a \in A_P} c_a$; its resource-cost, which depends on the REFs associated with the path, is defined below. A path may be subject to multiple, nonlinear, and interdependent constraints, provided that each can be expressed through REFs.

Sadykov et al. (2026) proposed a generic solver for the Resource Constrained Shortest Path Problem in which the user specifies a collection of problem-dependent functions, following the REF formalism originally introduced by Irnich (2008a), that drive a labeling algorithm. We adopt their notation to describe how our framework evaluates the cost and feasibility of a neighborhood move, since the same REFs used in their meta-solver for exact branch-and-price algorithms can be reused, unmodified, within our heuristic framework.

Let $\mathcal{R}$ denote the set of *resources* that any feasible path must respect

(e.g., load, elapsed time, route duration, number of visits). For a path P, we define its associated *label* as the tuple $L(P) = \big(v(P),\ c(P),\ \{\, s_R(P) \,\}_{R\in\mathcal{R}}\big)$. or simply $L = (v,\, c,\, \{s_R\}_{R\in\mathcal{R}})$ when the underlying path is clear from context, where $v(P) \in V$ is the last vertex visited along P, $c(P) \in \mathbb{R}$ is the accumulated cost of P, and $s_R(P)$ is the state of resource $R \in \mathcal{R}$ at the end of P. With a slight abuse of notation, we write either $v(L)$, $c(L)$, $s_R(L)$ or $v(P)$, $c(P)$, $s_R(P)$, depending on whether it is more convenient to refer to the label or to the underlying path.

To compute the label of a path, following Sadykov et al. (2026), the user provides, for each resource $R \in \mathcal{R}$, the following functions. `initStateForward()` returns the initial forward state s_R for a path containing only the depot; `extendForward`(s_R, a) returns, given a state s_R, the updated state after traversing arc a and the cost $c_R \in \mathbb{R} \cup \{\infty\}$ to be added to the path cost, where $c_R = \infty$ signals infeasibility; `initStateBackward()`, and `extendBackward`(s_R, a) are the analogous functions for backward paths; `concatenate`(s_R^f, s_R^b) returns, given a forward state s_R^f and a backward state s_R^b meeting at the same vertex, the cost resulting from joining the two partial paths with respect to resource R, where the cost is $+\infty$ if the concatenation is not feasible. This is exactly the interface required by a labeling-based pricing algorithm; no additional function needs to be implemented to use the proposed framework.

The initial label, corresponding to a path containing only vertex 0, is $(0, 0, \{\texttt{initStateForward}(R)\}_{R\in\mathcal{R}})$. The label of a path $P = (0, v_1, \ldots, v_k)$ is obtained by applying `extendAlongArcForward` (Algorithm 1) to the label of the subpath $P' = (0, v_1, \ldots, v_{k-1})$, using arc (v_{k-1}, v_k) as the second argument.

Algorithm 2 computes the cost of a path P, with $c(P) = \infty$ signaling infeasibility.

Following Sadykov et al. (2026), we call $P = (0, v_1, \ldots, v_k)$ a *forward path*, since it starts at the depot and ends at an arbitrary vertex. These definitions extend naturally to a *backward path*, starting at an arbitrary vertex and ending at the depot. Instead of traversing an entire path to obtain its cost and feasibility, we can traverse a prefix in the forward direction,

Algorithm 1 extendAlongArcForward procedure

procedure EXTENDALONGARCFORWARD(L, a)
  $v(L) \leftarrow v^+(a)$
  $c(L) \leftarrow c(L) + c_a$
  for $R \in \mathcal{R}$ **do**
    $(s'_R, c'_R) \leftarrow$ `extendForward`$(s_R(L), a)$
    $s_R(L) \leftarrow s'_R$
    $c(L) \leftarrow c(L) + c'_R$
  end for
  return L
end procedure

Algorithm 2 Computing the cost of a path (feasible iff finite)

procedure EVALUATEPATH(P)
  $L \leftarrow (0, 0, \{$`initStateForward`$(R)\}_{R \in \mathcal{R}})$
  for $a \in A_P$ **do**
    $L \leftarrow$ `extendAlongArcForward`(L, a)
  end for
  return $c(L)$
end procedure

producing a *forward label* $\overrightarrow{L}$, and the complementary suffix in the backward direction, producing a *backward label* $\overleftarrow{L}$. Given $\overrightarrow{L}$ and $\overleftarrow{L}$ meeting at the same vertex, we obtain a *complete label* representing the entire path by concatenation (Algorithm 3), using the `concatenate` function defined for each resource $R \in \mathcal{R}$. The consistency of this forward/backward decomposition, i.e., that concatenating $\overrightarrow{L}$ and $\overleftarrow{L}$ at any vertex yields the same cost and feasibility as evaluating the whole path directly, follows from Sadykov et al. (2026).

Algorithm 3 Concatenating two labels

procedure CONCATENATEONVERTEX($\overrightarrow{L}, \overleftarrow{L}$)
  $concatCost \leftarrow c(\overrightarrow{L}) + c(\overleftarrow{L})$
  for $R \in \mathcal{R}$ **do**
    $c'_R \leftarrow$ `concatenate`$(s_R(\overrightarrow{L}), s_R(\overleftarrow{L}))$
    $concatCost \leftarrow concatCost + c'_R$
  end for
  return $concatCost$
end procedure

In practice, when building, perturbing, or improving a solution, inter-

mediate labels associated with partial paths are kept in memory to avoid recomputation, so that combinations of known partial paths can be evaluated efficiently.

We classify resources into two categories: standard and customized.

A resource is *standard* if it satisfies the following property. For each arc a, let q_a denote its resource consumption and $[l_a, u_a]$ the feasible resource interval. The accumulated consumption along any feasible path must respect these bounds, assuming it is allowed to consume an arbitrary additional amount of resource on each arc to meet the lower bounds (e.g., waiting time in time-window constraints).

The REFs for standard resources, where state s represents the minimum (maximum) accumulated consumption in a forward (backward) path, are the following. `initStateForward()`$= (0, 0)$; `extendForward`$(s, a) = (\max\{s + q_a,\, l_a\},\, 0)$ if $s + q_a \leq u_a$, and $(_,\, \infty)$, otherwise; `initStateBackward()`$= (\infty, 0)$; `extendBackward`$(s, a) = (\min\{s,\, u_a\} - q_a,\, 0)$ if $s \geq l_a$, and $(_,\, \infty)$, otherwise; `concatenate`$(s_f, s_b) = 0$ if $s_f \leq s_b$, and ∞, otherwise. Capacity and time-window constraints are classical examples of standard resources. Any resource that cannot be modeled as standard is instead cast as a *customized* resource, with problem-specific initialization, extension, and concatenation functions.

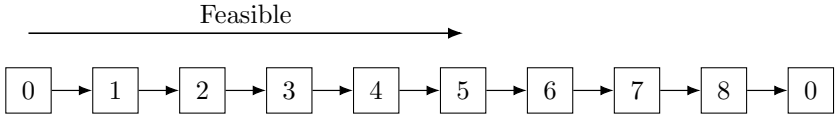
2.1. Dealing with infeasibility

A central feature of the proposed framework is its ability to explore infeasible solutions during the search without requiring REFs to report any measure of infeasibility magnitude, the boolean feasible/infeasible outcome above is all that is required and it is all that most REFs originally written for a pricing algorithm provide.

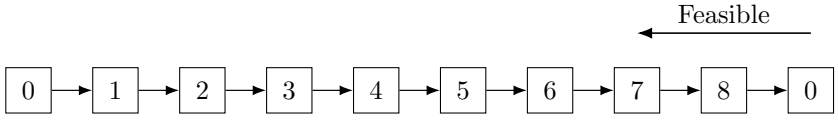
For a path P, let σ_P denote an *upper-bound estimate* of the number of arcs of P that violate at least one resource constraint. This estimate is obtained by evaluating P independently in the forward and backward directions: once an extension is found infeasible in a given direction, every subsequent arc in that direction is also considered infeasible (Algorithm 1), so each direction yields a count of feasible arcs from its starting endpoint.

Taking the larger of the two counts as the number of feasible arcs, σ_P is defined as the path length minus this value. Note that an arc may be counted in σ_P due to a violation of *either* a standard or a customized resource; both resource types contribute to this same estimate.

To illustrate the computation of σ_P, consider a path $P = (0, 1, 2, 3, 4, 5, 6, 7, 8, 0)$, where feasibility is checked in both directions independently.



(a) Path feasibility in forward direction.



(b) Path feasibility in backward direction.

As shown in Figures 1a and 1b, the forward traversal identifies infeasibility starting at arc $(5 \to 6)$ in the forward direction and $(6 \to 7)$ in the backward direction, yielding four infeasible arcs. The backward traversal, in turn, detects infeasibility earlier in the reversed sequence, resulting in a higher count. Since σ_P is defined as the minimum over both directions, we conclude that $\sigma_P = 4$.

For standard resources specifically, infeasibility can be quantified more precisely. We therefore define a second family of REFs for standard resources, automatically derived by the framework from the boolean REFs above, requiring no additional implementation effort from the user, in which the cost returned corresponds to the *magnitude* of the violation. Here, `initStateForward()` returns $(0, 0)$, and `extendForward`(s, a) returns

$$\Big(\min\{\max\{s + q_a,\, l_a\},\, u_a\},\ \max\{0,\, s + q_a - u_a\}\Big);$$

symmetrically, `initStateBackward()` returns $(\infty, 0)$ and `extendBackward`(s, a) returns

$$\Big(\max\{\min\{s,\, u_a\},\, l_a\} - q_a,\ \max\{l_a - s,\, q_a - s,\, 0\}\Big);$$

and `concatenate`(s_f, s_b) returns $\max\{s_f - s_b, 0\}$. These REFs follow directly from Sadykov et al. (2026). We denote by σ'_P the total violation magnitude along P, summed over all standard resources. As with σ_P, an arc violating a standard resource contributes to σ'_P regardless of whether it also violates a customized resource and is therefore already counted in σ_P; the two estimates are intentionally allowed to overlap, as they penalize infeasibility along different, complementary scales (arc count vs. violation magnitude).

With these two estimates, we define the objective function guiding the search. Let S be a solution composed of a set of routes (paths). Its cost is

$$f(S) \;=\; \sum_{P \in \text{paths}(S)} c(P) \;+\; \alpha \sum_{P \in \text{paths}(S)} \sigma_P \;+\; \beta \sum_{P \in \text{paths}(S)} \sigma'_P. \qquad (1)$$

Note that α and β are the *only* two penalty coefficients in the model, associated respectively with the arc-count estimate σ_P and the magnitude estimate σ'_P; both are adjusted dynamically during the search, following Vidal et al. (2012), to balance cost minimization against feasibility enforcement.

3. Nested Iterated Local Search Algorithm

The *Iterated Local Search* (ILS), Lourenço et al. (2003), metaheuristic is widely regarded as one of the most effective general-purpose frameworks for solving Vehicle Routing Problems (VRPs). Its success is largely attributed to the combination of powerful local improvement mechanisms with strategic diversification, enabling the search to escape local optima while preserving high-quality solution structures.

At its core, ILS alternates between a local search phase and a perturbation phase. Starting from an initial solution, a local search procedure is first applied in order to reach a local optimum. Subsequently, a perturbation is introduced to modify the current solution, after which the local search is applied again. An acceptance criterion determines whether the resulting solution replaces the incumbent best solution. This process is repeated until a termination condition, typically defined as a maximum number of consecu-

tive iterations without improvement, is met. Algorithm S1 (Supplementary Material) summarizes the classical ILS framework.

A particularly successful variant for VRPs is the *ILS with Randomized Variable Neighborhood Descent and Set Partitioning* (ILS-RVND-SP), introduced by Subramanian et al. (2013), in which the local search phase is performed using a Randomized Variable Neighborhood Descent (RVND). During the search, feasible routes generated by intermediate solutions are stored in a route pool. After the iterative phase terminates, an SP formulation is solved using the collected routes as columns, enabling a powerful combinatorial recombination step that often yields substantial improvements in solution quality. Algorithm S2 (Supplementary Material) presents this approach.

Rather than directly employing the ILS-RVND-SP scheme, we adopt an approach based on the *Nested Iterated Local Search* (NILS), originally proposed by Hussin and Stützle (2009). The key distinction between NILS and classical ILS lies in the structure of the local search phase: in NILS, the local search itself consists of an ILS procedure.

By replacing the RVND procedure with an internal ILS-RVND cycle, the ILS-RVND-SP framework is naturally extended to a nested scheme, denoted here as NILS-SP. This hierarchical structure enables a deeper exploitation of promising regions of the search space while preserving the diversification capabilities of the outer ILS. Algorithm S3 (Supplementary Material) details the resulting NILS-SP procedure; the stopping and acceptance criteria of its two nested loops are detailed in Section 3.1.

3.1. Stopping and Acceptance Criteria

Algorithm S3 (Supplementary Material) involves two nested loops, the outer NILS level and the inner `ILS-RVND` call, each governed by its own stopping and acceptance criteria.

At the outer (NILS) level, the stopping criterion is a time budget: the outer **while** loop terminates once a prescribed running time is reached, regardless of the number of iterations performed. The acceptance criterion follows a Simulated Annealing scheme, Kirkpatrick et al. (1983), in which

the temperature is a function of elapsed time rather than of the iteration count, so that the probability of accepting a worse solution S as the new reference solution S' decreases monotonically as the remaining time budget is consumed.

At the inner (`ILS-RVND`) level, the stopping criterion is instead a maximum number of consecutive iterations, $innerIter_{max}$, without improvement of the incumbent penalized cost; and the acceptance criterion is purely elitist, i.e., a candidate solution is accepted as the new reference solution only if it improves the best penalized cost $f(S^*)$ found so far within that call. Unlike the outer level, no worse solution is ever accepted at the inner level.

3.2. Initial Solution

The initial solution is constructed by means of a *Best Parallel Insertion* (BPI) heuristic, Potvin and Rousseau (1993), which builds all routes concurrently rather than sequentially. Let $N = \{1, \ldots, n\}$ denote the set of customers to be served and let 0 denote the depot. The procedure is initialized with K seed routes of the form $(0, s_k, 0)$, $k = 1, \ldots, K$, where each seed customer $s_k \in N$ is randomly selected. The set of unrouted customers is then defined as $U = N \setminus \{s_1, \ldots, s_K\}$. At each iteration, the heuristic evaluates every insertion of every customer $i \in U$ into every position p of every current route $r \in \mathcal{R}$. The cost of each candidate insertion is measured by the penalized objective function and the triple (i^*, p^*, r^*) that leads to the smallest cost is then committed to the incumbent solution, and customer i^* is removed from U. The process is repeated until $U = \emptyset$.

3.3. Neighborhood Structures

In the literature, specialized data structures are often used to preprocess subsequences of the current solution in such a way that the objective function associated with a path composed of a fixed number of such subsequences can be computed in $O(1)$ time. Here, because of the way the generic problem is defined, it is only possible to precompute labels that correspond to subsequences that either start or end at the depot. Evaluating a path that contains a long middle subsequence of the current solution would

therefore require propagating labels across that subsequence, which is too time-consuming to be performed at every move evaluation. We mitigate this issue by restricting attention to neighborhood moves whose evaluation decomposes each affected route into at most two contiguous segments: one originating at the depot and one terminating at the depot. This structural property enables the direct application of the label propagation mechanism, allowing move costs to be evaluated in $O(1)$ time per candidate.

We consider the following seven classical neighborhoods derived from the λ-exchange framework (Osman, 1993): `InterShift(1,0)`, removal of a single customer from one route and insertion into another; `InterShift(2,0)`, removal of two consecutive customers from one route and insertion into another, preserving their relative order; `InterSwap(1,1)`, exchange of a single customer between two distinct routes; `InterSwap(2,1)`, exchange of two consecutive customers from one route with a single customer from another; `InterSwap(2,2)`, exchange of two consecutive customers between two distinct routes; `2-opt`*, exchange of the tails of two routes at given cut positions; and `Relocate`, reinsertion of a single customer at a different position within the same route.
To illustrate why these neighborhoods are compatible with the labeling scheme, consider the `InterShift(1,0)` move applied to the routes

$$r_1 = (0, v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, 0), \qquad r_2 = (0, w_1, \ldots, w_j, w_{j+1}, \ldots, 0).$$

Removing v_i from r_1 and inserting it between w_j and w_{j+1} in r_2 yields four segments:

$$(0, v_1, \ldots, v_{i-1}), \quad (v_{i+1}, \ldots, 0), \quad (0, w_1, \ldots, w_j, v_i), \quad (w_{j+1}, \ldots, 0).$$

Each segment either originates at or terminates at the depot, matching precisely the structure of the precomputed forward and backward labels. The same decomposition holds, by construction, for all seven neighborhoods listed above.

These neighborhoods are exhaustively explored in an RVND algorithm

and only the best move, with regard to the evaluation function described before, for each pair of routes is accepted.

Additional speed-up mechanisms are implemented within the local search, as described in Section S2 of the Supplementary Material.

3.4. Perturbation

Perturbation plays a central role in the Iterated Local Search framework by driving the search away from local optima and enabling the exploration of distant regions of the solution space. In the proposed algorithm, perturbation is implemented by applying a sequence of k random moves drawn from two neighborhood structures: `InterShift(1,0)` and `InterSwap(1,1)`. At each step, one of the two move types is selected uniformly at random, and a move is then chosen uniformly at random among all candidate moves of the selected type. The process is repeated k times, yielding a perturbed solution.

The perturbation strength k controls the trade-off between diversification and the ability to recover a good solution during the subsequent local search: small values of k produce mild perturbations that are easily repaired, while large values drive the search further but may destroy too much of the incumbent structure.

Algorithm S4 (Supplementary Material) details the perturbation procedure.

The choice of `InterShift(1,0)` and `InterSwap(1,1)` as the perturbation operators is deliberate. Both moves modify two routes simultaneously, ensuring that the perturbation introduces structural changes that are unlikely to be immediately reversed by the subsequent local search phase. Intra-route operators, by contrast, tend to produce milder changes that the local search can undo more easily, offering weaker diversification.

3.5. Adaptive penalty management

The penalty coefficients α and β are adjusted so as to keep the proportion of feasible solutions close to a target rate $\rho^\star$, similar to the adaptive-penalty principle of Vidal et al. (2012). Two sliding windows record, over the last

W evaluated solutions, whether each was feasible with respect to capacity and with respect to time, respectively. Every K iterations, the observed feasible fractions ρ^{α} and ρ^{β} are compared to $\rho^{\star}$: whenever a fraction falls below $\rho^{\star}$, the corresponding penalty is multiplied by a factor $1+\eta^{+}$ to push the search toward feasibility; whenever it exceeds $\rho^{\star}$, the penalty is divided by $1+\eta^{-}$ to encourage a controlled exploration of the infeasible region. The two penalties evolve independently, so that the search can drive capacity and time feasibility at different paces.

3.6. Set Partitioning

After the main loop of Algorithm S3 (Supplementary Material) terminates, the best solution found is further refined by means of a *Set Partitioning* (SP) formulation, following the ILS-RVND-SP scheme of Subramanian et al. (2013). Throughout the execution of NILS-SP, every feasible route belonging to any solution accepted by the outer ILS is collected into a route pool $\mathcal{P}$ (lines 4, 6, and 11 of Algorithm S3, Supplementary Material). Duplicate routes are discarded so that each route appears at most once in $\mathcal{P}$. The SP formulation then selects, from this pool, a minimum-cost subset of routes that covers every customer exactly once while respecting the maximum number of vehicles available.

Let $V^{+} = \{v_1, \ldots, v_n\}$ denote the set of customers, K the maximum number of vehicles, and c_r the cost of route $r \in \mathcal{P}$. For each route $r \in \mathcal{P}$ and customer $i \in V^{+}$, let $a_{ir} \in \{0,1\}$ indicate whether customer i is served by route r. The formulation is:

$$\min \quad \sum_{r \in \mathcal{P}} c_r x_r \tag{2}$$

$$\text{s.t.} \quad \sum_{r \in \mathcal{P}} a_{ir}\, x_r = 1, \quad \forall\, i \in V^{+}, \tag{3}$$

$$\sum_{r \in \mathcal{P}} x_r \leq K, \tag{4}$$

$$x_r \in \{0,1\}, \quad \forall\, r \in \mathcal{P}. \tag{5}$$

The objective (2) minimizes the total routing cost. Constraints (3) ensure that every customer is covered by exactly one selected route. Constraint (4) limits the number of routes selected to at most K. The formulation (2)–(5) is solved to optimality by a general-purpose MIP solver (line 19 of Algorithm S3, Supplementary Material), and its solution constitutes the final output of NILS-SP.

The number of variables equals $|\mathcal{P}|$ and the number of constraints equals $n+1$; both quantities remain manageable for the instance sizes considered in our experiments, since the LP relaxation of the SP formulation is typically very tight and the MIP solver converges quickly in practice.

Restricting $\mathcal{P}$ to feasible routes only guarantees that the SP solution inherits the full feasibility guarantees of the labeling framework: every route in $\mathcal{P}$ was verified feasible during the ILS-RVND local search phase, so no penalty terms are required in the objective and any solution to (2)–(5) automatically satisfies all resource constraints encoded in the REFs.

4. Counting infeasibility

This subsection presents the main methodological contribution of this paper, namely a procedure to compute an upper bound on the number of infeasible arcs, σ, in a given solution. Since each neighborhood operator introduced in the previous section exploits this bound in a specific manner, the corresponding methods are described separately in dedicated subsections, in order to facilitate comprehension and reproducibility.

An issue that arises due to the generality of the considered problem is efficiently computing the change in the number of infeasible arcs, σ, resulting from a candidate move, as computing all labels of each path affected by the move is highly costly. Instead, we compute an upper bound for σ, denoted as $\bar{\sigma}$, while searching for an improving movement. If the cost of the modified solution that considers $\bar{\sigma}$ as the resulting number of infeasible arcs does not improve the current cost, the move is discarded. Thus, we allow for discarding improving moves. However, non-discarded moves are guaranteed to improve the current solution. After the search for an improving movement

is finished, we compute the exact value of σ and re-calculate the objective function.

We say that a label L is *feasible*, written $\texttt{isFeas}(L)$, if and only if $c(L) < \infty$; this criterion is used throughout the remainder of this section every time a label resulting from an extension or a concatenation is tested.

To apply these methods, we maintain auxiliary data structures that store, for each route of a solution, its forward and backward labels. These structures are denoted by $\overrightarrow{\mathcal{L}}$ and $\overleftarrow{\mathcal{L}}$, respectively. Given a route r with $|r|$ arcs (and therefore $|r|+1$ vertices, indexed $1, \ldots, |r|+1$, with vertex 1 and vertex $|r|+1$ both equal to the depot), we write $\overrightarrow{\mathcal{L}}_p^r$, $p \in \{1, \ldots, |r|+1\}$, for the forward label associated with the partial path that starts at the depot and ends at the p-th vertex of r, and $\overleftarrow{\mathcal{L}}_p^r$ for the backward label associated with the partial path that starts at the p-th vertex of r and ends at the depot. In particular, $\overrightarrow{\mathcal{L}}_1^r$ and $\overleftarrow{\mathcal{L}}_{|r|+1}^r$ are both trivial labels, and $\overrightarrow{\mathcal{L}}_{|r|+1}^r$ and $\overleftarrow{\mathcal{L}}_1^r$ both correspond to the complete route. Note that $\overrightarrow{\mathcal{L}}_p^r$ and $\overleftarrow{\mathcal{L}}_p^r$ are indexed by the same vertex position p, even though they traverse the route in opposite directions; this position-indexed convention is adopted throughout this section because it yields simpler and more symmetric formulas. Whenever the route is clear from the context, we shall omit the superscript and write simply $\overrightarrow{\mathcal{L}}_p$ and $\overleftarrow{\mathcal{L}}_p$.

To illustrate this concept, consider a VRP defined on a directed graph $G = (V, A)$, where $V = \{0, v_1, v_2, v_3, v_4\}$ and $A = V \times V$. Suppose a solution composed of the routes

$$r_1 = (0, v_1, v_2, 0) \qquad \text{and} \qquad r_2 = (0, v_3, v_4, 0),$$

so that $|r_1| = |r_2| = 3$ and both routes have 4 vertices, indexed $1, 2, 3, 4$. For route r_1, the forward label structure is given by

$$\overrightarrow{\mathcal{L}}^{r_1} \;=\; \big(\, L(0),\; L(0 \to v_1),\; L(0 \to v_1 \to v_2),\; L(0 \to v_1 \to v_2 \to 0) \,\big),$$

so that $\overrightarrow{\mathcal{L}}_1^{r_1} = L(0)$, $\overrightarrow{\mathcal{L}}_2^{r_1} = L(0 \to v_1)$, $\overrightarrow{\mathcal{L}}_3^{r_1} = L(0 \to v_1 \to v_2)$, and $\overrightarrow{\mathcal{L}}_4^{r_1} = L(0 \to v_1 \to v_2 \to 0)$. The corresponding backward label structure

is

$$\overleftarrow{\mathcal{L}}^{r_1} = \big(L(0 \to v_1 \to v_2 \to 0),\ L(v_1 \to v_2 \to 0),\ L(v_2 \to 0),\ L(0) \big),$$

so that $\overleftarrow{\mathcal{L}}_1^{r_1} = L(0 \to v_1 \to v_2 \to 0)$, $\overleftarrow{\mathcal{L}}_2^{r_1} = L(v_1 \to v_2 \to 0)$, $\overleftarrow{\mathcal{L}}_3^{r_1} = L(v_2 \to 0)$, and $\overleftarrow{\mathcal{L}}_4^{r_1} = L(0)$. Analogous structures are defined for route r_2.

By exploiting the recursive nature of the resource extension functions, both $\overrightarrow{\mathcal{L}}^{r}$ and $\overleftarrow{\mathcal{L}}^{r}$ can be built in $O(|r|)$ time and stored in $O(|r|)$ space, so that the total cost of maintaining the auxiliary structures for an entire solution is $O(n)$, where n is the number of customers. This is asymptotically better than the $O(n^2)$ effort required by neighborhood evaluations that recompute partial labels from scratch at each move, and the gain becomes particularly pronounced in instances with long routes. Algorithm 4 details the procedure used to compute $\overrightarrow{\mathcal{L}}$ and $\overleftarrow{\mathcal{L}}$ from a given solution S, based on the label functions introduced in the previous section.

For each position $p \in \{2, \ldots, |r|+1\}$, we denote by $a_p(r) = (v_{p-1}, v_p) \in A$ the arc of route r that ends at the p-th vertex. Whenever the route is clear from the context, we shall write simply a_p.

Algorithm 4 is executed only once at the beginning of the NILS procedure. Since each neighborhood move affects at most two routes, it is sufficient to recompute the label structures exclusively for the modified routes, ensuring that the overall computational overhead remains limited during the search.

This structure also allows us to easily compute the violation magnitudes of each standard resource, as we can just extend/concatenate labels to retrieve these values; because of this, we omit these calculations and focus only on the customized resources' procedures.

For a route r, let $\overrightarrow{\texttt{lastFeas}}(r) \in \{1, \ldots, |r|+1\}$ denote the largest position p such that $\overrightarrow{\mathcal{L}}_p^{r}$ is feasible; since $\overrightarrow{\mathcal{L}}_1^{r}$ is always feasible (trivial label), $\overrightarrow{\texttt{lastFeas}}(r)$ is well defined. By construction, once $\overrightarrow{\mathcal{L}}_p^{r}$ is infeasible for some p, so is $\overrightarrow{\mathcal{L}}_{p'}^{r}$ for every $p' > p$; hence $\overrightarrow{\texttt{lastFeas}}(r) - 1$ is exactly the number of feasible arcs encountered by the forward sweep, and a fully feasible route satisfies $\overrightarrow{\texttt{lastFeas}}(r) = |r| + 1$.

Algorithm 4 Computation of the forward and backward label structures $\overrightarrow{\mathcal{L}}$ and $\overleftarrow{\mathcal{L}}$.

procedure EVALUATEPATHS(S)
  $\overrightarrow{\mathcal{L}} \leftarrow \emptyset$
  $\overleftarrow{\mathcal{L}} \leftarrow \emptyset$
  for $r \in$ `GetRoutes`(S) **do**
    $\overrightarrow{\mathcal{L}}_1^r \leftarrow$ `initStateForward`() ▷ Creates the trivial forward label $L(0)$
    $\overleftarrow{\mathcal{L}}_{|r|+1}^r \leftarrow$ `initStateBackward`() ▷ Creates the trivial backward label $L(0)$
    for $p = 2$ to $|r| + 1$ **do**
      $a \leftarrow a_p(r)$
      $\overrightarrow{\mathcal{L}}_p^r \leftarrow$ `extendAlongArcForward`($\overrightarrow{\mathcal{L}}_{p-1}^r$, a)
      if $\neg$ `isFeas`($\overrightarrow{\mathcal{L}}_p^r$) **then** ▷ Check if the label is feasible
        break
      end if
    end for
    for $p = |r|$ down to 1 **do**
      $a \leftarrow a_{p+1}(r)$
      $\overleftarrow{\mathcal{L}}_p^r \leftarrow$ `extendAlongArcBackward`($\overleftarrow{\mathcal{L}}_{p+1}^r$, a)
      if $\neg$ `isFeas`($\overleftarrow{\mathcal{L}}_p^r$) **then**
        break
      end if
    end for
  end for
  return $\overrightarrow{\mathcal{L}}$, $\overleftarrow{\mathcal{L}}$
end procedure

Symmetrically, let $\overleftarrow{\texttt{lastFeas}}(r) \in \{1, \ldots, |r| + 1\}$ denote one plus the number of arcs successfully traversed by the backward sweep before the first infeasible extension is found (equivalently, one plus the number of iterations of the backward loop of Algorithm 4 that completed before the **break**, or $|r| + 1$ if none did). In terms of positions, $\overleftarrow{\texttt{lastFeas}}(r) = |r| + 2 - p$, where p is the smallest position such that $\overleftarrow{\mathcal{L}}_p^r$ is feasible; since $\overleftarrow{\mathcal{L}}_{|r|+1}^r$ is always feasible (trivial label), p, and hence $\overleftarrow{\texttt{lastFeas}}(r)$, is well defined. By construction, once $\overleftarrow{\mathcal{L}}_p^r$ is infeasible for some p, so is $\overleftarrow{\mathcal{L}}_{p'}^r$ for every $p' < p$; hence $\overleftarrow{\texttt{lastFeas}}(r) - 1$ is exactly the number of feasible arcs encountered by the backward sweep, and a fully feasible route satisfies $\overleftarrow{\texttt{lastFeas}}(r) = |r|+1$.

Both $\overrightarrow{\texttt{lastFeas}}(r)$ and $\overleftarrow{\texttt{lastFeas}}(r)$ are thus expressed as positions along the route, one counted from the origin depot and the other from the destination depot. For $p > \overrightarrow{\texttt{lastFeas}}(r)$ (resp. $p < \overleftarrow{\texttt{lastFeas}}(r)$), we treat $\overrightarrow{\mathcal{L}}_p^r$

(resp. $\overleftarrow{\mathcal{L}}_p^r$) as an implicit infeasible sentinel label, with $c = \infty$, so that any extension, concatenation, or `isFeas` test applied to it correctly propagates infeasibility without requiring the label to have been explicitly constructed. When the route is clear from the context, we write simply $\overrightarrow{\texttt{lastFeas}}$ and $\overleftarrow{\texttt{lastFeas}}$.

4.1. InterShift(1,0)

The `InterShift(1,0)` move removes a customer v_{pos} from a source route r_1 and inserts it at a position pos' in a destination route r_2. Since the resources are route-decomposable, the impact on solution quality is evaluated by independently computing an upper bound on the number of infeasible arcs of both affected routes. This evaluation uses the precomputed forward and backward label structures, allowing for a rapid estimation of the feasibility frontier of the modified routes without full reconstruction.

Removal from r_1.. Let r denote the source route, with vertices indexed $1, \ldots, |r| + 1$, and let r' denote the route obtained after removing the customer at position $pos \in \{2, \ldots, |r|\}$ (so that pos is neither the source nor the destination depot). The removal deletes the arcs $a_{pos}(r)$ and $a_{pos+1}(r)$ and introduces the bridge arc $a' = (v_{pos-1}, v_{pos+1})$. Algorithm 5 details the procedure. The algorithm first tests whether r' is fully feasible by extending the forward label at position $pos - 1$ along the bridge arc and concatenating the result with the backward label at position $pos + 1$. If this concatenation is feasible, the modified route is fully feasible and zero infeasible arcs are returned. Otherwise, the algorithm counts how many arcs of r' survive in each sweep direction, accounting for both the surviving arcs of the original route and the contribution of the bridge. Note that the procedure only takes into account the arcs associated with the unmodified labels $\overrightarrow{\mathcal{L}}_{pos-1}^r$ and $\overleftarrow{\mathcal{L}}_{pos+1}^r$, as well as the bridge arc a'. For better performance, the remaining arcs are considered infeasible without checking, even if the extension through the bridge arc is feasible; this is what makes the procedure return an *upper bound* $\bar{\sigma}$ rather than the exact value.

In Algorithm 5, r' has $|r| - 1$ arcs (two arcs are deleted and the bridge adds one back). The term $\min\{pos - 2, \overrightarrow{\texttt{lastFeas}} - 1\}$ counts the arcs of the

Algorithm 5 Upper bound on infeasible arcs (Removal)

procedure INFEASARCSREMOVAL(r, pos, $\overrightarrow{\mathcal{L}}$, $\overleftarrow{\mathcal{L}}$)
  $a' \leftarrow (v_{pos-1}, v_{pos+1})$ ▷ Bridge arc
  $\overrightarrow{L} \leftarrow$ `extendAlongArcForward`($\overrightarrow{\mathcal{L}}^{r}_{pos-1}$, a')
  $\overleftarrow{L} \leftarrow$ `extendAlongArcBackward`($\overleftarrow{\mathcal{L}}^{r}_{pos+1}$, a')
  $L \leftarrow$ `concatenateOnVertex`($\overrightarrow{L}$, $\overleftarrow{\mathcal{L}}^{r}_{pos+1}$)
  if `isFeas`(L) **then**
    return 0 ▷ Modified route is fully feasible
  end if

  $nFwd \leftarrow \min\{pos - 2, \overrightarrow{\texttt{lastFeas}} - 1\}$ ▷ Surviving feasible arcs before the bridge
  if `isFeas`($\overrightarrow{L}$) **then**
    $nFwd \leftarrow nFwd + 1$ ▷ Bridge is feasible
  end if

  $nBwd \leftarrow \min\{|r| - pos, \overleftarrow{\texttt{lastFeas}} - 1\}$ ▷ Surviving feasible arcs after the bridge
  if `isFeas`($\overleftarrow{L}$) **then**
    $nBwd \leftarrow nBwd + 1$
  end if

  return $(|r| - 1) - \max\{nFwd, nBwd\}$ ▷ Upper bound on infeasible arcs of r'
end procedure

original route that survive on the left side of the bridge: these are the arcs among vertices $1, \ldots, pos - 1$, of which there are $pos - 2$ in total, capped by the number of arcs already known to be feasible in the forward sweep ($\overrightarrow{\texttt{lastFeas}} - 1$); the bridge itself adds $+1$ if its extended label $\overrightarrow{L}$ is feasible. The term $\min\{|r| - pos, \overleftarrow{\texttt{lastFeas}} - 1\}$ is symmetric: it counts the arcs among vertices $pos + 1, \ldots, |r| + 1$, of which there are $|r| - pos$ in total (this range includes the arc leaving vertex $pos + 1$, which is why the count is $|r| - pos$ rather than $|r| - pos - 1$), capped by the number of arcs already known feasible in the backward sweep.

Insertion into r_2. Let r denote the destination route and r' the route obtained by inserting customer c between the vertices at positions $pos - 1$ and pos, for $pos \in \{2, \ldots, |r| + 1\}$. The insertion deletes the arc (v_{pos-1}, v_{pos}) and introduces the two new arcs $a^- = (v_{pos-1}, c)$ and $a^+ = (c, v_{pos})$. Algorithm 6 follows the same structure as the removal procedure, but accounts for the fact that two new arcs are introduced instead of one. Accordingly,

the forward bound checks the feasibility of two successive extensions, $\overrightarrow{L}$ (along a^-) and $\overrightarrow{L}'$ (further extending $\overrightarrow{L}$ along a^+); the backward bound proceeds symmetrically.

Algorithm 6 Upper bound on infeasible arcs (Insertion)

procedure INFEASARCSINSERTION(r, c, pos, $\overrightarrow{\mathcal{L}}$, $\overleftarrow{\mathcal{L}}$)
  $a^- \leftarrow (v_{pos-1}, c)$
  $a^+ \leftarrow (c, v_{pos})$
  $\overrightarrow{L} \leftarrow$ `extendAlongArcForward`$(\overrightarrow{\mathcal{L}}^r_{pos-1}, a^-)$
  $\overrightarrow{L}' \leftarrow$ `extendAlongArcForward`$(\overrightarrow{L}, a^+)$
  $\overleftarrow{L} \leftarrow$ `extendAlongArcBackward`$(\overleftarrow{\mathcal{L}}^r_{pos}, a^+)$
  $\overleftarrow{L}' \leftarrow$ `extendAlongArcBackward`$(\overleftarrow{L}, a^-)$
  $L \leftarrow$ `concatenateOnVertex`$(\overrightarrow{L}', \overleftarrow{\mathcal{L}}^r_{pos})$
  if `isFeas`(L) **then**
    return 0 ▷ Modified route is fully feasible
  end if

  $nFwd \leftarrow \min\{pos - 2, \overrightarrow{\texttt{lastFeas}} - 1\}$
  if `isFeas`$(\overrightarrow{L})$ **then**
    $nFwd \leftarrow nFwd + 1$
  end if
  if `isFeas`$(\overrightarrow{L}')$ **then**
    $nFwd \leftarrow nFwd + 1$
  end if

  $nBwd \leftarrow \min\{|r| + 1 - pos, \overleftarrow{\texttt{lastFeas}} - 1\}$
  if `isFeas`$(\overleftarrow{L})$ **then**
    $nBwd \leftarrow nBwd + 1$
  end if
  if `isFeas`$(\overleftarrow{L}')$ **then**
    $nBwd \leftarrow nBwd + 1$
  end if

  return $(|r| + 1) - \max\{nFwd, nBwd\}$
end procedure

In Algorithm 6, r' has $|r|+1$ arcs (one arc is deleted and two are introduced). The two indicators in the forward count reflect the two arcs introduced by the insertion: by the monotonicity of the resource extension functions, if $\overrightarrow{L}'$ is feasible then so is $\overrightarrow{L}$, but not conversely. Hence the contribution of the new arcs to $nFwd$ ranges in $\{0, 1, 2\}$. The backward case is symmetric. The two min terms, $\min\{pos - 2, \ldots\}$ and $\min\{|r| + 1 - pos, \ldots\}$, count the

arcs of the original route that are entirely to the left (vertices $1, \ldots, pos-1$) and entirely to the right (vertices $pos, \ldots, |r|+1$) of the insertion point, respectively; unlike the removal case, no original arc is deleted on either side, which is why these two counts sum to $|r|-1$, i.e., all arcs of r except the single arc (v_{pos-1}, v_{pos}) replaced by the insertion.

By the route-decomposability of the resources, the bound on the total number of infeasible arcs introduced by the InterShift(1,0) move is the sum of the bounds for the source and destination routes.

Both INFEASARCSREMOVAL and INFEASARCSINSERTION extend readily to the removal and insertion of $\lambda > 1$ consecutive customers, which is what `InterShift(λ,0)` requires: removal replaces the bridge arc by $(v_{pos-1}, v_{pos+\lambda})$, and insertion links the λ new customers with $\lambda - 1$ additional forward and backward extensions within the inserted segment; in both cases, $nFwd$ and $nBwd$ are updated after every such extension exactly as in Algorithms 5 and 6.

The same technique can be applied to the other moves, as detailed in Section S3 of the Supplementary Material.

5. Computational Experiments

This section reports computational experiments conducted to assess the effectiveness of the proposed framework on a diverse set of vehicle routing problem variants. All experiments were carried out on a machine equipped with Ryzen 5 5600G, 16 GB of RAM, running Windows 11. The algorithm was implemented in Julia 1.12 and all MIP models were solved using CPLEX version 20.1.

The four problem variants selected for evaluation are the *Risk-Constrained Cash-in-Transit Vehicle Routing Problem* (RCTVRP), the *Vehicle Routing Problem with Simultaneous Pickup and Delivery* (VRPSPD), the *Multi-Depot Vehicle Routing Problem* (MDVRP), and the *Heterogeneous Fleet Vehicle Routing Problem* (HFVRP). These problems were deliberately chosen because, taken together, they exercise complementary aspects of the proposed framework. The RCTVRP stresses the resource extension mecha-

nism under non-standard risk-based feasibility criteria. The VRPSPD tests the framework's ability to handle coupled forward and backward load constraints within the same route. The MDVRP exercises the multi-depot generalization of the graph structure and the initialization procedure. Finally, the HFVRP evaluates the framework under heterogeneous resource capacities, requiring vehicle-dependent REFs. Collectively, these variants probe different combinations of the labeling scheme, the infeasibility estimation technique, and the Set Partitioning post-processing step, providing a comprehensive assessment of the generality and robustness of the proposed approach.

For each problem, the proposed framework is compared against the best-performing method reported in the literature. Solution quality is assessed via the gap to the Best Known Solution (BKS), defined as

$$\text{gap} = 100 \cdot \frac{z - z^*}{z^*},$$

where z is the solution value obtained by the proposed framework and z^* is the BKS. We report the best gap (GAP_{BEST}), the average gap (GAP_{AVG}) over 5 independent runs, and the average computing time (TIME) in seconds.

The REF model adopted for each problem variant, along with a brief problem description, is given in the Supplementary Material (Section S4). Results for all four variants are consolidated below.

5.1. Results

This section reports the computational results of the proposed framework on the four VRP variants introduced above. For each problem, we compare against the best-performing method reported in the literature for its standard benchmark instances; the instance sets and reference algorithm adopted for each problem are as follows.

For **RCTVRP**, we use sets O, S, and V (Talarico et al. (2016)), against the reference algorithm ILS-SP, proposed by Roboredo et al. (2026). For **VRPSPD**, we use sets D (Dethloff (2001)), MG (Montané and Galvao (2006)), and SN (Salhi and Nagy (1999)), against the reference algorithm

FSPD, run at its standard 100,000-iteration configuration (FSPD100k), proposed by Cavaliere et al. (2024). For **MDVRP**, we use the standard sets P and PR (Cordeau et al. (1997)), against the reference algorithm PyVRP, proposed by Wouda et al. (2024). For **HFVRP**, we use sets Golden (Golden et al. (1984)) and Brandão (Brandão (2011)), each considered under an unlimited fleet (Fleet Size and Mix, FSM), further split by variant into FSMFD (fixed cost per vehicle plus a variable, type-dependent per-distance cost), FSMF (fixed cost per vehicle only, uniform variable cost across types), and FSMD (variable, type-dependent per-distance cost only, no fixed cost) – denoted, e.g., $\text{Golden}_{\text{FSMFD}}$ – with the Brandão instances available only in the FSMD variant, against the reference algorithm AILS, proposed by Máximo et al. (2022).

Because each reference algorithm was run on different hardware, the running times reported in the literature are not directly comparable to those obtained with our own machine (an AMD Ryzen 5 5600G). Table 1 reports, for each reference algorithm, the CPU it was run on, and Table 2 reports its single-thread PassMark score and the resulting scaling factor applied to convert the literature running time to an equivalent running time on our machine.

Table 1: Hardware used by each reference algorithm

Problem	Reference algorithm	CPU
RCTVRP	ILS-SP Roboredo et al. (2026)	Intel Core i7-10700 @ 2.90 GHz
VRPSPD	FSPD Cavaliere et al. (2024)	AMD Ryzen 5 PRO 4650GE @ 3.30 GHz
MDVRP	PyVRP Wouda et al. (2024)	same machine as ours
HFVRP	AILS Máximo et al. (2022)	Intel Xeon E5-2680v2 @ 2.80 GHz

Table 2: The single-thread PassMark score (P_{ref}) used to scale its reported running time to our own machine.

Problem	P_{ref}	Scaling factor
RCTVRP	2879	0.906
VRPSPD	2632	0.828
MDVRP	3178	1.000
HFVRP	1783	0.561

Section 5.1.1 reports the overall comparison against these reference meth-

ods, under a time budget matched to each reference method's own running time, split by instance size; Section 5.1.2 instead evaluates the framework on the BCP-relevant instances (those with $|V| \leq 150$) under a substantially larger time budget, reflecting its intended use as a source of upper bounds for Branch-Cut-and-Price algorithms, where solution quality, not running time, is the binding requirement; and Section 5.1.3 isolates the contribution of the Set Partitioning post-processing step to the results reported in the first two.

5.1.1. Overall comparison

Table 3 summarizes, for each problem and instance set, the average percentage gap of the best solution found over all runs (Gap_{best}), the average percentage gap of the average solution quality (Gap_{avg}), both relative to the BKS, and the average running time in seconds (T), for our framework (*This paper*) and for the best-performing method reported in the literature for that problem (Ref). For each instance, the proposed framework was run with a time budget T equal to the running time reported by the corresponding reference method, so T for *This paper* throughout this section is a deliberately chosen stopping budget rather than the time the search needed to converge. For *This paper* we additionally report T*, the average time, within that budget, at which the search last improved the incumbent solution of the run, i.e., the point after which no further improvement was found before the budget was exhausted. T* is therefore always at most T, and the gap between the two indicates how much of the allotted budget was spent without further gains. The time budget itself was set from the reference method's raw, as-published running time; for the comparison in Table 3, however, the Ref column T is instead rescaled to our own machine using the single-thread PassMark factors of Table 2, so that the T columns of Ref and *This paper* become directly comparable. Each instance set is further split by instance size: for a set X, X^- denotes the subset of its instances with at most 150 customers and X^+ the subset with more than 150 customers; a set with no instances above 150 customers is reported as a single, unsplit row.

Table 3: Comparison between *This paper* and the best-performing literature method (Ref) for each problem, by instance set, split by instance size (X^-: at most 150 customers; X^+: more than 150 customers). T for both *Ref* and *This paper* is hardware-scaled to our own machine (see Tables 1 and 2), so the two T columns are directly comparable.

		Ref			This paper			
Problem	Set	Gap_{best}	Gap_{avg}	T (s)	Gap_{best}	Gap_{avg}	T (s)	T*(s)
RCTVRP	O^-	0.004	0.022	10.625	0.010	0.095	14.454	12.194
	O^+	0.000	0.000	73.621	0.434	1.628	90.334	87.327
	S^-	0.004	0.013	5.734	0.037	0.110	7.649	4.410
	S^+	0.223	0.813	144.523	-0.011	0.245	180.981	165.174
	V^-	0.067	0.284	25.289	0.230	0.544	34.759	33.403
	V^+	0.310	0.822	178.309	1.310	3.462	239.829	233.833
VRPSPD	D	0.000	0.000	25.651	0.000	0.013	30.973	5.899
	MG^-	0.011	0.039	26.562	0.002	0.051	35.265	26.500
	MG^+	0.091	0.240	28.465	1.777	2.358	40.602	40.600
	SN^-	0.000	0.013	23.021	0.318	0.780	28.738	23.857
	SN^+	0.000	0.012	27.533	0.930	2.747	44.531	44.531
MDVRP	P^-	0.000	0.017	49.144	0.034	0.172	54.598	40.321
	P^+	0.000	0.108	139.675	0.846	1.524	154.096	145.539
	PR^-	0.000	0.000	60.505	0.105	0.355	66.890	52.571
	PR^+	0.000	0.076	146.962	1.079	1.644	162.160	162.157
HFVRP	$\text{Golden}_{\text{FSMFD}}$	0.002	0.017	14.092	0.863	1.000	16.613	16.236
	$\text{Golden}_{\text{FSMF}}$	0.000	0.043	15.423	0.343	0.775	18.670	18.072
	$\text{Golden}_{\text{FSMD}}$	0.001	0.038	17.567	0.639	1.051	20.274	15.739
	$\text{Brandão}_{\text{FSMD}}{}^-$	0.071	0.117	35.755	0.495	1.025	39.723	38.680
	$\text{Brandão}_{\text{FSMD}}{}^+$	0.032	0.364	50.213	1.033	1.550	55.364	55.362

On the RCTVRP, the framework stays close to the reference on the − subsets of all three sets, while the + subsets show a markedly wider gap, most notably V^+ (1.310% best, 3.462% avg); S^+ is the one exception, with a slightly negative best gap.

On the VRPSPD, D and MG^- match the reference closely, but MG^+ shows the largest degradation in the whole comparison (1.777% best), and SN^+ widens comparably, consistent with the overhead of the generic REF-based evaluation growing with route length.

On the MDVRP, both P^- and PR^- stay within about a tenth of a percentage point of PyVRP, widening to roughly 0.8–1.1 points on their + counterparts.

On the HFVRP, the three Golden sets, which have no instances above 150 customers, show a gap of up to 0.86% best against AILS, and $\text{Brandão}_{\text{FSMD}}$

widens further still on its single + instance.

Across all sets, Gap_{best} stays below 0.5% on the − subset of every set except $\text{Golden}_{\text{FSMFD}}$ and $\text{Golden}_{\text{FSMD}}$ (0.863% and 0.639%), and rises further, up to 1.777%, on the + subsets, confirming that route length, not problem-specific overhead, is the main driver of the residual gap. Running time for *This paper* is somewhat above Ref's hardware-scaled T, since the budget itself was set from the literature's raw, unscaled time; even so, the two remain of the same order of magnitude on every set, without any problem-specific engineering beyond the definition of the corresponding REFs. Section 5.1.2 shows that, given a larger time budget, the residual gap on the − subsets, the size range most relevant to Branch-Cut-and-Price integration, shrinks even further.

5.1.2. Solution quality under a larger time budget

The proposed framework is intended primarily as a source of upper bounds for Branch-Cut-and-Price algorithms, a setting in which the binding requirement is solution quality within a computing budget of minutes, rather than the sub-second running times targeted by the specialized reference heuristics of Section 5.1.1. To assess how close to optimality the framework can get under such a budget, Table 4 revisits the BCP-relevant instances of Table 3, i.e., those already denoted X^- (at most 150 customers), now run with a time budget T much larger.

Under this tenfold-larger budget, the gap to the reference methods narrows sharply relative to Table 3. Gap_{best} drops below 0.12% on every set, and $\text{MDVRP}_{\text{P}}{}^-$ turns slightly negative, meaning the best solution found across runs is, on average, marginally below the reported BKS. $\text{VRPSPD}_{\text{MG}}{}^-$ and $\text{VRPSPD}_{\text{D}}{}^-$ both reach essentially the same quality as the reference, and the three Golden HFVRP sets stay within a few hundredths of a percentage point of AILS in Gap_{best}. The largest remaining gaps are on $\text{VRPSPD}_{\text{SN}}{}^-$ (0.115% best, 0.236% avg) and, in Gap_{avg} terms, $\text{Golden}_{\text{FSMFD}}{}^-$ (0.323%), still a small fraction of a percentage point. Comparing running times, T for *This paper* is consistently well above the (hardware-scaled) reference time, by roughly an order of magnitude on most sets, since the budget was de-

Table 4: Comparison between *This paper*, run with a time budget ten times the reference method's running time, and the best-performing literature method (Ref), for each problem, on the BCP-relevant (X^-) instances of Table 3. As in Table 3, Ref's T is hardware-scaled to our own machine (see Tables 1 and 2).

		Ref			This paper			
Problem	Set	Gap_{best}	Gap_{avg}	T (s)	Gap_{best}	Gap_{avg}	T (s)	T*(s)
RCTVRP	O^-	0.004	0.022	10.625	0.008	0.037	121.212	81.219
	S^-	0.004	0.013	5.733	0.008	0.012	66.473	35.053
	V^-	0.067	0.284	25.288	0.022	0.206	291.348	278.911
VRPSPD	D^-	0.000	0.000	25.650	0.000	0.001	318.985	22.836
	MG^-	0.011	0.039	26.562	0.000	0.000	336.348	193.245
	SN^-	0.000	0.013	23.021	0.115	0.236	302.254	252.357
MDVRP	P^-	0.000	0.017	49.144	-0.027	-0.013	491.936	357.994
	PR^-	0.000	0.000	60.505	0.000	0.010	604.849	413.978
HFVRP	$\text{Golden}_{\text{FSMFD}}{}^-$	0.002	0.017	14.092	0.011	0.323	271.331	239.493
	$\text{Golden}_{\text{FSMF}}{}^-$	0.000	0.043	15.423	0.046	0.126	303.383	285.181
	$\text{Golden}_{\text{FSMD}}{}^-$	0.001	0.038	17.567	0.018	0.139	320.007	203.545
	$\text{Brandão}_{\text{FSMD}}{}^-$	0.071	0.117	35.755	0.054	0.084	643.168	643.082

liberately set ten times larger; T*, however, shows that the search typically converges to its best solution well before the budget is exhausted, particularly on the smaller RCTVRP and HFVRP sets. Taken together, these results confirm that, given a computing budget an order of magnitude larger than that of specialized heuristics, the proposed framework closes almost the entire gap to the literature on the instance sizes of primary interest for Branch-Cut-and-Price integration, supporting its intended role as a generic source of near-optimal upper bounds.

5.1.3. Contribution of Set Partitioning

To isolate the contribution of the Set Partitioning post-processing step, Table 5 reports, for each problem and instance set, the average percentage gap to the BKS before Set Partitioning is applied ($\text{Gap}_{\text{pre-SP}}$, computed from the best NILS-SP-incumbent cost at the moment the Set Partitioning model is invoked) and after ($\text{Gap}_{\text{post-SP}}$, the Gap_{best} values already reported in Table 3), together with the average number of routes collected in the pool that the Set Partitioning model selects from. Twenty runs on the largest RCTVRP_{V} instances, where the underlying NILS search never reached a feasible solution within the time budget, are excluded from this comparison,

since no pre-SP gap can be computed for them; Set Partitioning nonetheless recovered a feasible solution from the route pool in all twenty cases.

Table 5: Impact of the Set Partitioning post-processing step, by problem and instance set.

Problem	Set	$\text{Gap}_{\text{pre-SP}}$	$\text{Gap}_{\text{post-SP}}$	Pool
RCTVRP	O	0.618	0.085	642
	S	0.350	0.028	714
	V	1.379	0.462	2268
VRPSPD	D	0.039	0.000	1476
	MG	2.073	1.104	2239
	SN	1.307	0.331	2925
MDVRP	P	1.138	0.382	973
	PR	1.438	0.592	779
HFVRP	$\text{Golden}_{\text{FSMFD}}$	1.280	0.066	620
	$\text{Golden}_{\text{FSMF}}$	1.245	0.304	846
	$\text{Golden}_{\text{FSMD}}$	1.070	0.278	672
	$\text{Brandão}_{\text{FSMD}}$	1.373	0.393	1047

Set Partitioning reduces the average gap on every set of every problem, typically by 50% to 95%, reaching a full recovery to 0.000% on VRPSPD_{D}, and in no case leaves the gap unchanged or worse. The effect is largest in absolute terms on the sets with the largest pre-SP gaps, $\text{VRPSPD}_{\text{MG}}$ (2.073% to 1.104%) and RCTVRP_{V} (1.379% to 0.462%), and, as noted above, Set Partitioning is occasionally the only source of a feasible solution at all on the largest RCTVRP instances. The pool size, which ranges from roughly 600 to nearly 3,000 routes depending on the problem and set, reflects the number of distinct feasible routes collected during the local search phase and grows with instance size and with the number of neighborhoods explored during the search. In every case, the route pool is large enough for Set Partitioning to recombine high-quality fragments discovered separately across different runs of the local search into markedly better solutions than any single NILS run achieves on its own. This confirms that the Set Partitioning step is not a marginal refinement but a substantial and consistent contributor to the solution quality reported throughout this section.

Instance-level results, including raw solution costs and running times, together with the full set of parameter values used for each problem variant, are reported in the Supplementary Material (Sections S5 and S6).

6. Conclusion

This paper presented a generic heuristic framework for a broad class of Vehicle Routing Problems, in which problem structure is specified through Resource Extension Functions. The framework integrates a Nested Iterated Local Search with a Set Partitioning post-processing step, and handles infeasible solutions throughout the search without requiring the REFs to provide explicit infeasibility measures.

A central contribution is the technique for estimating move infeasibility in constant time. By maintaining forward and backward label structures in $O(n)$ time and space per solution, the method derives an upper bound on the number of infeasible arcs of the modified route without reconstructing the full label sequence. For each neighborhood structure analyzed, the bound requires at most a constant number of label extensions and one concatenation, regardless of route length, improving upon the $O(n^2)$ effort of approaches that recompute partial labels from scratch at each move evaluation.

The NILS-SP algorithm combines this technique with a Randomized Variable Neighborhood Descent over seven neighborhood structures, a route pool collected throughout the search, and a final Set Partitioning step that recombines high-quality route fragments into the best overall solution. The framework was assessed on four structurally distinct VRP variants, the Risk Constrained VRP, the VRP with Simultaneous Pickup and Delivery, the Multi-Depot VRP, and the Heterogeneous Fleet VRP, producing competitive results across all variants and confirming that a single REF-driven framework can match or surpass specialized algorithms on a wide range of routing problems.

Several directions remain open for future research. On the theoretical side, deriving tighter infeasibility bounds that exploit the union of forward and backward feasible arc sets could reduce the number of moves forwarded to exact evaluation. On the modeling side, extending the framework to problems with optional visits, such as the Orienteering Problem and its routing variants, and to cluster-based settings such as the Generalized VRP represents a natural but non-trivial generalization, requiring revisiting the

construction heuristic, the perturbation operators, and the Set Partitioning formulation. More broadly, problems combining routing with assignment or scheduling subproblems present further opportunities to test the limits of the REF-based approach proposed here.

Acknowledgments

MR received funding from project CNPq Universal 408562/2025-0. AP received funding from project Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) (Grant number: 311654/2023-1).

References

Brandão, J., 2011. A tabu search algorithm for the heterogeneous fixed fleet vehicle routing problem. Computers & Operations Research 38 (1), 140–151.

Cavaliere, F., Accorsi, L., Laganà, D., Musmanno, R., Vigo, D., 2024. An efficient heuristic for very large-scale vehicle routing problems with simultaneous pickup and delivery. Transportation Research Part E: Logistics and Transportation Review 186, 103550.

Cordeau, J.-F., Gendreau, M., Laporte, G., 1997. A tabu search heuristic for periodic and multi-depot vehicle routing problems. Networks 30 (2), 105–119.

Dethloff, J., 2001. Vehicle routing and reverse logistics: The vehicle routing problem with simultaneous delivery and pick-up: Fahrzeugeinsatzplanung und redistribution: Tourenplanung mit simultaner auslieferung und rückholung. OR-spektrum 23 (1), 79–96.

Golden, B., Assad, A., Levy, L., Gheysens, F., 1984. The fleet size and mix vehicle routing problem. Computers & Operations Research 11 (1), 49–66.

Hussin, M. S., Stützle, T., 2009. Hierarchical iterated local search for the quadratic assignment problem. In: Blesa, M. J., Blum, C., Di Gaspero, L., Roli, A., Sampels, M., Schaerf, A. (Eds.), Hybrid Metaheuristics. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 115–129.

Irnich, S., 2008a. Resource extension functions: Properties, inversion, and generalization to segments. OR Spectrum 30 (1), 113–148.

Irnich, S., 2008b. A unified modeling and solution framework for vehicle routing and local search-based metaheuristics. INFORMS Journal on Computing 20 (2), 270–287.

Kirkpatrick, S., Gelatt Jr, C. D., Vecchi, M. P., 1983. Optimization by simulated annealing. science 220 (4598), 671–680.

Lourenço, H. R., Martin, O. C., Stützle, T., 2003. Iterated Local Search. pp. 320–353.

Máximo, V. R., Cordeau, J.-F., Nascimento, M. C., 2022. An adaptive iterated local search heuristic for the heterogeneous fleet vehicle routing problem. Computers & Operations Research 148, 105954.

Montané, F. A. T., Galvao, R. D., 2006. A tabu search algorithm for the vehicle routing problem with simultaneous pick-up and delivery service. Computers & Operations Research 33 (3), 595–619.

Osman, I. H., 1993. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. Annals of Operations Research 41 (4), 421–451.

Pessoa, A., Sadykov, R., Uchoa, E., Vanderbeck, F., 2020. A generic exact solver for vehicle routing and related problems. Mathematical Programming 183 (1), 483–523.

Potvin, J.-Y., Rousseau, J.-M., 1993. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. European Journal of Operational Research 66 (3), 331–340.

Roboredo, M., Oliveira, B., Lichau, S., Pessoa, A., Uchoa, E., 2026. Exact and heuristic methods for the risk-constrained cash-in-transit vehicle routing problem. Kuwait Journal of Science 53 (4), 100648.
URL https://www.sciencedirect.com/science/article/pii/S2307410826001185

Sadykov, R., Froger, A., Uchoa, E., Pessoa, A., Bulhoes, T., de Araujo, D., 2026. Bucket graph meta-solver for the resource constrained shortest path problem.

Salhi, S., Nagy, G., 1999. A cluster insertion heuristic for single and multiple depot vehicle routing problems with backhauling. Journal of the operational Research Society 50 (10), 1034–1042.

Subramanian, A., Uchoa, E., Ochi, L. S., 2013. A hybrid algorithm for a class of vehicle routing problems. Computers & Operations Research 40 (10), 2519–2531.

Talarico, L., Springael, J., Sörensen, K., Talarico, F., 2016. A large neighbourhood metaheuristic for the risk-constrained cash-in-transit vehicle routing problem. Computers & Operations Research 78, 547–556.

Vidal, T., Crainic, T. G., Gendreau, M., Lahrichi, N., Rei, W., 2012. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. Operations Research 60 (3), 611–624.

Vidal, T., Crainic, T. G., Gendreau, M., Prins, C., 2013. Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. European Journal of Operational Research 231 (1), 1–21.

Vidal, T., Crainic, T. G., Gendreau, M., Prins, C., 2014. A unified solution framework for multi-attribute vehicle routing problems. European Journal of Operational Research 234 (3), 658–673.

Wouda, N. A., Lan, L., Kool, W., 2024. Pyvrp: A high-performance vrp solver package. INFORMS Journal on Computing 36 (4), 943–955.