https://pubsonline.informs.org/journal/ijoc

INFORMS JOURNAL ON COMPUTING

Vol. 00, No. 0, Xxxxx 0000, pp. 000–000

Submitted to *INFORMS Journal on Computing*

ISSN 0899-1499, EISSN 1526-5528

RouteOpt: An Open-Source Modular Exact Solver for Vehicle Routing Problems

Zhengzhong You,[1] Yu Yang[1,*]

[a]Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL, USA

[*]Corresponding author.

Contact: you.z@ufl.edu, yu.yang@ise.ufl.edu

Abstract. Despite significant advancements in exact methods for vehicle routing problems (VRPs) over the past three decades, there remains a lack of high-performing and accessible open-source solvers for researchers and practitioners. To bridge this gap, we introduce RouteOpt, the first open-source modular exact solver for VRPs, delivering state-of-the-art performance while maintaining a flexible and extensible structure. RouteOpt achieves the best performance reported in the literature on both the capacitated vehicle routing problem (CVRP) and vehicle routing problem with time windows (VRPTW). Crucially, its modular design allows users to develop and integrate customized branching, cutting plane, and variable reduction modules to tackle a broad range of VRP variants. Furthermore, RouteOpt introduces a novel node restoration mechanism, enabling efficient parallel processing of a branch-and-bound tree. Leveraging this feature, we have, for the first time, proven the optimality of three open CVRP instances. By combining modularity, efficiency, and open accessibility, RouteOpt establishes itself as an invaluable platform for both academic research and real-world applications. The source code of RouteOpt and relevant data are publicly available at https://github.com/Zhengzhong-You/IJOC_JUN20_2025.

Key words: vehicle routing; open-source exact solver; modular design

1. Introduction

The branch-price-and-cut (BPC) solution framework is widely recognized as the most effective method for solving vehicle routing problems (VRPs). It typically provides tighter bounds and proves optimality faster than alternative branch-and-cut-based approaches. Despite the proven effectiveness of BPC algorithms, their implementation remains notoriously complex. As noted by Pessoa et al. (2021), designing and coding a sophisticated BPC system can require several months of effort from a skilled team. This complexity often demands deep expertise in both algorithm design and operations research, making BPC-based solvers inaccessible to many researchers and practitioners who could benefit from their use.

To address this critical gap, RouteOpt has been developed as an open-source, modular exact solver that implements a BPC framework alongside state-of-the-art optimization techniques. Specifically, it integrates advanced branching strategies, including three-phased branching (3PB; Pessoa et al. 2020) and two-stage learning-based branching (2LBB; You et al. 2023), along with their

corresponding dynamic versions, effectively balancing branching variable selection efficiency and quality. For pricing (i.e., column generation), RouteOpt implements bi-directional labeling (Righini and Salani 2006), *ng*-route relaxation (Baldacci et al. 2011), decremental state space relaxation (Martinelli et al. 2014), dynamic *ng* augmentation (Roberti and Mingozzi 2014, Bulhoes et al. 2018b), bucket graphs (Sadykov et al. 2021), and arc elimination (Irnich et al. 2010, Sadykov et al. 2021). Regarding cutting planes, it employs the well-established rounded capacity cuts (RCCs) (Naddef and Rinaldi 2002, Lysgaard et al. 2004) and limited memory rank-1 cuts (lm-R1Cs) (Pecin et al. 2017b, Bulhoes et al. 2018a), further boosting efficiency in proving optimality. Additionally, RouteOpt utilizes an enumeration procedure (Baldacci et al. 2008, Sadykov et al. 2021, Yang 2023) and the novel variable reduction method, DeLuxing, from Yang (2025) to close a branch-and-bound node when the optimality gap is small enough, further accelerating the solution process.

Equipped with these advanced techniques, RouteOpt achieves a remarkable breakthrough by proving the optimality of capacitated vehicle routing problem (CVRP) instances that would require years to solve with other exact VRP solvers. This demonstrates RouteOpt's exceptional capability to efficiently tackle large-scale VRPs.

1.1. Contributions

RouteOpt is designed with a modular architecture and an open-source philosophy, offering accessibility and adaptability to both researchers and practitioners. It allows users to easily customize or extend individual components to meet their specific research or practical needs. Notable high-performance modules, each implemented as separate, dedicated classes for maximum flexibility and clarity, include: (i) a **branching module** for efficient branching selection and management of the branch-and-bound tree (BBT); (ii) two specialized **cutting plane modules** (R1Cs and RCCs) for effective cut separation and streamlined reduced cost (RC) computations in the labeling algorithm (for column generation) and constraint matrix updates; and (iii) a robust **variable reduction module** leveraging the DeLuxing approach proposed in Yang (2025). Furthermore, RouteOpt introduces a novel **node restoration** functionality allowing critical node information, including solver environments and settings, to be externally saved and restored. This capability enables parallel computation across machines to achieve massive scalability.

This work makes the following key contributions.

(1) RouteOpt empowers researchers to test novel approaches by seamlessly replacing individual modules, avoiding the need to develop an entire solver from scratch. For example, branching strategy researchers, such as those studying the cluster branching method from (Silva et al.

2024), only need to specify the branching variable type and candidate set based on the current LP fractional solution. RouteOpt can then employ its default 3PB or best-k formula (BKF; You et al. 2023) or any user-defined selection strategies without code modification, thereby streamlining both experimental setup and evaluation. Similarly, researchers exploring new cutting-plane separation methods, such as neural network-based RCC separators (Kim et al. 2024), can assess integrative performance within a state-of-the-art BPC framework, facilitating direct comparison with established tools like CVRPSEP (Lysgaard 2003). As a result, RouteOpt dramatically lowers the barrier for related research, including the largely unexplored learning-based R1C separation, creating new opportunities for advancing algorithmic and learning-to-optimize research in exact VRP solution methods.

(2) RouteOpt enables the rapid and straightforward development of exact solvers for a wide range of VRP variants. Users typically only need to customize the pricing function while leveraging existing modules as-is. Its components are designed for broad applicability, facilitating research on problems such as the capacitated multi-trip VRPTW (CMTVRPTW; Yang 2023), the heterogeneous fleet VRP (HFVRP; Sadykov et al. 2021), and the split delivery VRP (SDVRP; Balster et al. 2023), all of which share significant structural similarities with the CVRP. In particular, RouteOpt's modular design facilitates methodological innovations in pricing, such as machine learning-based column selection (Morabit et al. 2021), by supporting comprehensive evaluation beyond the root node.

(3) RouteOpt introduces a unique node restoration feature that allows researchers to save and reload solver states and related data. This capability enables distributed computing for tackling challenging, large-scale instances. Furthermore, it supports rigorous benchmarking of heuristic methods and enables in-depth analysis of optimal solution structures, unlocking new potential for research on large-scale VRP instances.

The remainder of the paper is organized as follows. Section 2 reviews related solvers, highlighting their distinctive features and contributions to VRP research. Section 3 describes the workflow of RouteOpt, providing insights into its operational logic and process flow. Sections 4 through 7 detail the core modules, summarizing the underlying algorithms and providing usage examples for each. Section 8 presents computational results that demonstrate the efficiency and effectiveness of RouteOpt in addressing two fundamental VRPs: the CVRP and VRPTW. Finally, Section 9 concludes with a discussion on the impact of open-source, modular solvers like RouteOpt in advancing vehicle routing and other combinatorial optimization challenges and outlines directions for future research.

2. Comparative Analysis of Existing VRP Solvers

In this section, we review the most relevant solvers to RouteOpt. While a comprehensive survey is beyond our scope, we focus on three (two heuristic and one exact) notable solvers selected for their state-of-the-art performance, user-friendly design, or both. For a broader overview of related solvers, we refer readers to Wouda et al. (2024). We summarize the key strengths, limitations, and ideal use cases of each solver and clarify how RouteOpt distinguishes itself within this landscape and guiding researchers in choosing the tool that best aligns with their specific needs.

Vidal (2022) introduces HGS-CVRP, a high-performance open-source hybrid genetic search (HGS) solver designed specifically for the CVRP. Building on the foundation of Vidal et al. (2012), it incorporates more than a decade of methodological refinements, including the SWAP* operator, which significantly enhances local search performance. HGS-CVRP remains among the most competitive metaheuristics in terms of solution quality. It is implemented in C++ with wrappers available in C, Python, and Julia. Owing to its exceptional performance and focused design, it is ideal for quickly obtaining high-quality solutions for CVRP instances or serving as a primal heuristic within exact solvers. It is worth mentioning that RouteOpt leverages HGS-CVRP to obtain a primal solution when a good one is not provided by users. However, despite being open-source, adapting HGS-CVRP to other VRP variants requires considerable effort and domain expertise. Therefore, for use cases that require customization, a more versatile solver may be preferable.

Wouda et al. (2024) presents PyVRP, a Python package implementing a variant of the HGS algorithm originally introduced in Vidal et al. (2013) for the VRPTW. PyVRP has steadily expanded its capabilities and now supports over six core VRP variants while maintaining strong performance. It strikes a balance between flexibility and efficiency by delegating performance-critical components to C++ while retaining all user customizations at the Python level. The package is well-documented and adheres to strong software engineering practices, making it highly extensible. This emphasis on generality comes at the cost of a slight decrease in solution quality, though the impact is often negligible for small to medium-sized instances. Since exact solvers are often limited by time constraints to moderate-sized instances, PyVRP can serve as an effective primal heuristic. However, if the highest possible solution quality is required, especially for large-scale instances, a specialized solver may be more appropriate.

Pessoa et al. (2020) introduces VRPSolver, one of the leading BPC-based exact solvers for VRPs. Built upon BaPCod (Sadykov and Vanderbeck 2021), it supports multiple VRP variants through a flexible modeling interface. Its Julia frontend allows users to define decomposition schemes with

reformulation automatically generated, eliminating the need to explicitly model the master program or pricing subproblems. VRPSolver also offers customization options, including cut generation and branching callbacks, while providing high-performing default column generation, cut separation, and branching variable selection for VRPs. A simplified Python interface (Errami et al. 2024) further increases accessibility, allowing use by researchers with limited optimization expertise. Unlike traditional solvers, this interface abstracts away explicit integer programming formulations. However, its support is limited to standard VRP variants (e.g., those with capacity constraints, time windows, and heterogeneous fleets), and the underlying implementation details of its sophisticated BPC method remain proprietary.

As discussed, BPC-based solvers are notoriously complex to implement and customize, and currently, no state-of-the-art exact solver offers both full open-source access and ease of customization. This lack of openness hinders innovation, as any methodological improvements require substantial effort. RouteOpt fills this gap by providing an open-source, modular, and extensible BPC-based solver. Users can integrate only the components they need, without requiring familiarity with the entire system. Researchers interested in specific mechanisms, e.g., pricing, can directly inspect and experiment with the relevant source code. This transparency and modularity lower the barrier to entry, promote reproducibility, and accelerate research on exact methods for VRPs and related combinatorial optimization problems.

3. Overview

To clearly present RouteOpt's operational principles and provide context for the modules described in Sections 4–7, we first illustrate its general workflow in Figure 1. This diagram highlights the interplay among key components–cutting planes, pricing, branching, arc elimination, enumeration, and DeLuxing–that collectively drive the solution process. We then present the node restoration workflow in Figure 2, which outlines how node-related information is reloaded to initialize the solver, enabling massive parallelization across machines for tackling challenging large-scale problem instances.

3.1. Core Operational Workflow

As shown in Figure 1, the RouteOpt workflow begins with the construction of the root node from scratch, whose state is initialized to 0. This indicates that the node is not yet *enumerated*, i.e., its column (variable) set is incomplete. This root node is then inserted into the BBT to initiate the solution process. At each iteration, a node is selected from the BBT and processed by solving

the restricted master problem (RMP) via column generation. If the node is in an enumerated state (state 1), cutting planes are immediately applied to improve the lower bound. Otherwise, the solver attempts arc elimination and column enumeration; if successful, the node transitions to state 1; if not, it remains at 0. Cutting planes are then applied unless either the tailing-off condition is met (i.e., no significant improvement in the lower bound is observed) or the pricing process becomes prohibitively slow. If the LP solution remains fractional after the cutting phase, branching is typically performed. However, if the node is already in an enumerated state, RouteOpt can alternatively apply DeLuxing, a procedure that reduces the size of the enumerated column pool, thereby decreasing the reliance on branching. Child nodes generated by branching inherit their parent's state and are added to the BBT. This iterative process continues until the BBT is empty, marking the termination of the algorithm, or when optimality has been certified.

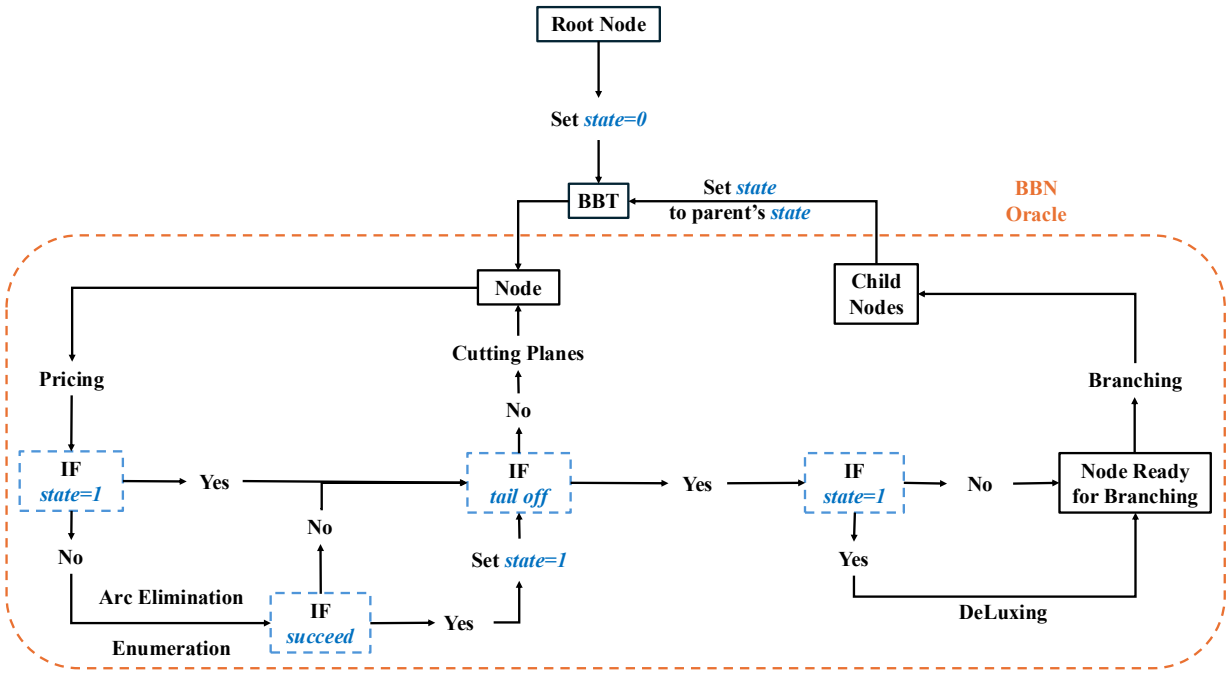

Figure 1 The general workflow of RouteOpt.

3.2. Scalable Parallelization via Node Restoration

The latest release of RouteOpt (version 2.0.0) introduces a powerful node restoration capability, enabling key node-specific information, such as solution environments and configuration settings, to be saved externally and subsequently reloaded to fully reconstruct the solver's state. This feature makes it possible to solve a single problem instance across machines simultaneously, potentially yielding orders-of-magnitude speedups for extremely challenging instances.

The node restoration workflow, illustrated in Figure 2, demonstrates how this feature achieves the desired parallelism. Starting from the construction of the root node, the workflow proceeds into the branch-and-bound node (BBN) Oracle. At each branching step, only one node is selected and passed to the BBT, while all other child nodes (possibly more than one for multiway branching) are saved externally. These external nodes are monitored by a separate program, typically implemented in Python or Bash scripts. When this external program detects a new external node file, it initiates a new job by loading the node data and reconstructing the corresponding BBN. Similar to the root node process, every branching operation within these new jobs maintains only one single active node, saving the rest externally. The scalability of this workflow depends mainly on available computational resources, i.e., CPUs and RAM. Ideally, running k jobs in parallel can yield up to a k-fold acceleration, provided there are no resource conflicts. Although this example illustrates retaining only one active node, RouteOpt does not impose strict rules for node saving; users have full flexibility in deciding when or whether to save nodes externally. In practice, RouteOpt avoids saving nodes with fewer than 10,000 enumerated columns, as these nodes can be closed fast.

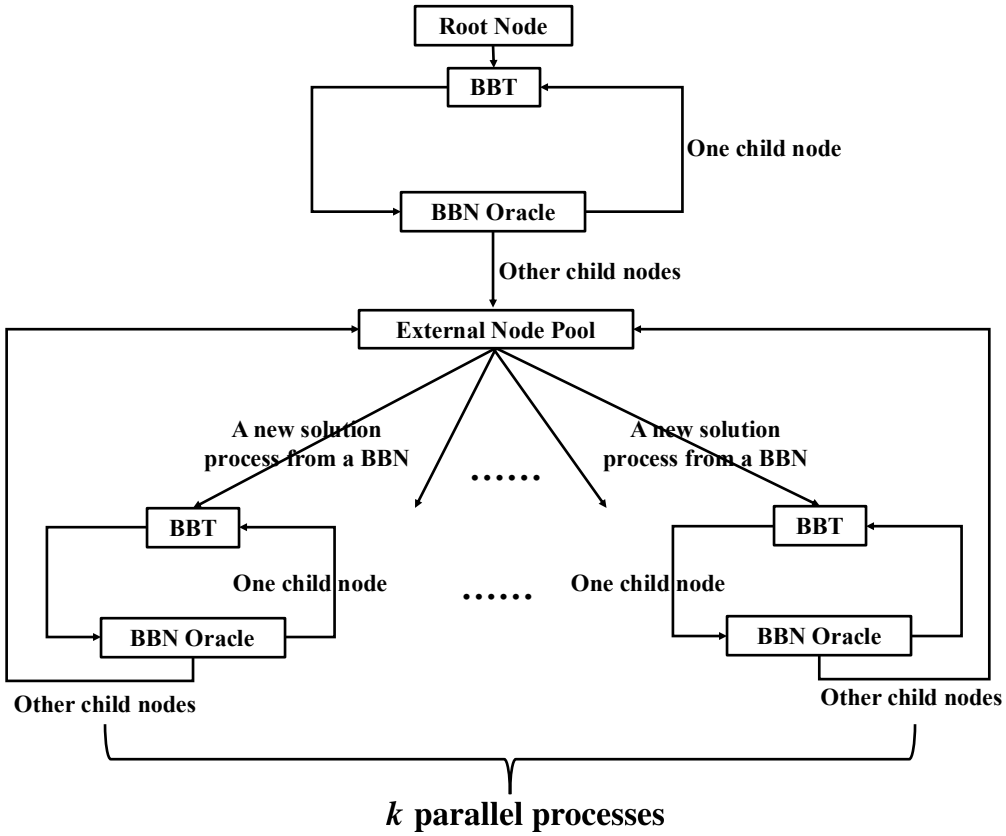

Figure 2 The node restoration workflow in RouteOpt.

4. The Branching Module

The branching module in RouteOpt offers a robust and customizable framework for building exact BPC-based VRP solvers. It enables users to construct solvers by simply providing key callback functions (e.g., pricing and cutting functors) without having to implement the entire branching component from scratch. By default, the module offers an efficient implementation of the 3PB rule (Pessoa et al. 2020), which automatically collects and utilizes historical data to make effective branching decisions. Additionally, the module features a dynamic adjustment mechanism (the BKF method from You et al. 2023), which adaptively determines the number of tests to perform before finalizing each branching decision. RouteOpt employs a best-bound-first search strategy by default but also supports user-defined node selection criteria. Beyond branching, the module efficiently manages the BBT by allowing node sorting with custom rules, updating the global lower bound, tracking the number of explored nodes, and maintaining detailed historical data. This comprehensive tracking supports intelligent solver control, such as detecting the tailing-off of cutting planes, further enhancing solver performance.

4.1. Algorithmic Structure and Branching Workflow

RouteOpt is built upon the well-established BPC solution framework (outlined in Algorithm 1). This framework augments the classic branch-and-bound method by integrating column generation and cutting planes. The solution process begins by either restoring a previously saved node or constructing a new root node, which is then added to the BBT. At each iteration, a node is selected from the tree for processing. The node first undergoes pricing to generate a valid local lower bound (LB), followed by the collection of historical data and the application of cutting planes to strengthen the LB. If the node is not `terminated`, i.e., its local LB is still less than the global upper bound (UB), the algorithm proceeds with branching to further explore the solution space. Branching is performed using a user-defined branching rule or the default 3PB rule. When the `BKF` option is enabled, it dynamically selects the optimal number of tests to perform for making the branching decision. Based on the branching decision, a branching constraint is imposed, and child nodes are generated accordingly. The treatment of child nodes is governed by the chosen configuration: if `writeOut` is enabled, some child nodes are saved externally with unique filenames and then removed from the job's memory. Otherwise, all child nodes are added to the BBT for subsequent processing. At the end of each iteration, the BBT is updated to reflect the current state of the search. This process iteratively continues until the BBT becomes empty or optimality has been certified.

Algorithm 1: The BPC Framework

Input: Root node: R

Output: Optimality certificate

```
if restoreNode is enabled then
    restoreNode(R);
else
    construct(R);
BBT ← R;
while BBT ≠ ∅ do
    n ← extractNode(BBT);
    pricing(n);
    collectBranchingData(n);
    cutting(n);
    if isTerminated(n) is false then
        if BKF is enabled then
            k* ← BKF(n);
        if userDefinedBranching is enabled then
            brc ← userDefinedBranching(n, k*);
        else
            brc ← 3PB(n, k*);
        {n1, n2, ···} ← imposeBranching(n, brc);
        delete(n) ;
        n1 → BBT;
        foreach ni ∈ {n2, ···} do
            if writOut is enabled then
                writeOut(ni) and delete(ni) ;
            else
                ni → BBT;
    else
        delete(n)
    updateBBT(BBT)
```

4.2. Branching Module Configuration and Execution

Figure 3 illustrates the steps for configuring and running the branching module to solve a CVRP instance. The process begins by initializing the problem, creating the root node, and setting up the BBT controller with appropriate parameters. The controller then manages the branching process,

applies the 3PB selection rule, and performs pricing and cutting operations to effectively guide the search toward an optimal solution. With minor modifications, this setup can be readily adapted to solve other VRPs, such as the VRPTW. For detailed explanations of each parameter, readers are referred to the online RouteOpt user manual.

4.3. Customization and Extension of Branching Strategies

Research on branching strategies primarily focuses on two key aspects: the type of variables to branch and the variable selection rule. RouteOpt is designed to facilitate easy comparison and experimentation with methods along both dimensions.

To customize the branching variable type, users must define a data structure that enables one-to-one mapping. For example, in branching on edge, a structure such as `std::pair<int, int>` is commonly used, while in branching on cutset, a container like `std::vector<int>` needs to be provided. In addition, a corresponding hash function (e.g., `PairHasher`) must be implemented to ensure each branching variable is uniquely identifiable, as RouteOpt relies on `std::unordered_map` for internal mapping. Proper hash function design is vital to minimize collisions. Once these components are specified, the BBTController can be initialized as `BBTController</* node_type*/, /* cut_type*/, /* cut_map*/>`.

For customizing the variable selection rule, users can override the default 3PB rule by providing their own selection function. For instance, to implement the 2LBB rule developed in You et al. (2023), users can pass a custom function such as `ml_candidate_selection` to the BBTController. This user-defined function will replace the default, and is supplied with relevant historical data at each call, enabling more context-aware branching decisions.

It is worth noting that although the current version of BBT supports multiway branching (allowing nodes to have more than two children), the default historical data collector currently only supports binary branching and stores information for left and right branches. To extend historical data tracking to multiway branching, the collector's data structure must be modified accordingly. As an alternative, users may choose to disable historical data collection altogether, thereby avoiding any potential inconsistencies.

5. The Limited-Memory Chvátal-Gomory Rank-1 Cut Module

Chvátal-Gomory (CG) cuts are a foundational class of cutting planes in integer programming (IP), first introduced by Chvátal (1973). They are derived by taking nonnegative linear combinations of valid inequalities and rounding down the resulting coefficients. CG cuts, particularly their rank-1

```cpp
#include "cvrp.hpp"
#include "cvrp_macro.hpp"
#include "bbt_controller.hpp"
using namespace RouteOpt;
using namespace Application::CVRP;

int main(int argc, char *argv[]) {
    auto *cvrp = new CVRPSolver(argc, argv); // Initialize solver.
    auto node = new BbNode(); // Create root node.
    /* Other functions definitions */
    Branching::BBT::BBTController<BbNode, pair<int, int>, PairHasher>bbt_controller(
        cvrp->getDim(),
        cvrp->refUB(),
        static_cast<int>(NUM_TESTING::PHASE0), static_cast<int>(NUM_TESTING::PHASE1),
        static_cast<int>(NUM_TESTING::PHASE2), static_cast<int>(NUM_TESTING::PHASE3),
        IF_BKF ? vector<pair<int, int>>({
        {static_cast<int>(BKF_TYPE::M_LP), static_cast<int>(BKF_TYPE::N_LP)},
        {static_cast<int>(BKF_TYPE::M_HEUR), static_cast<int>(BKF_TYPE::N_HEUR)}
        }) : vector<pair<int, int>>(),
        BbNode::defineBetterNode,
        BbNode::getNodeValue,
        BbNode::getNodeIdx,
        BbNode::getNodeIfInEnumState,
        BbNode::getLastBrc,
        BbNode::refNodeBrIncreaseVal,
        BbNode::getNodeIfTerminate,
        BbNode::obtainSolEdgeMap,
        [cvrp](auto a1, auto &a2, auto &a3, auto &a4) {
            return cvrp->processLPTesting(a1, a2, a3, a4);
        },
        [cvrp](auto a1, auto &a2, auto &a3, auto &a4) {
            return cvrp->processCGTesting<false>(a1, a2, a3, a4);
        },
        [cvrp](auto a1, auto &a2, auto &a3, auto &a4) {
            return cvrp->processCGTesting<true>(a1, a2, a3, a4);
        },
        [cvrp](auto a1) {
            return cvrp->callPricingAtBeg(a1);
        },
        [cvrp](auto a1) {
            return cvrp->callCutting(a1);
        },
        [cvrp](auto a1, auto a2, auto &a3) {
            return cvrp->imposeBranching(a1, a2, a3);
        },
        ml_candidate_selection,
        node_out_func,
        node_in_func
    );
    bbt_controller.solve(node, TIME_LIMIT); // Solve BBT.
    cvrp->printOptSol(
        cout,
        bbt_controller.getNumNodesExplored(),
        bbt_controller.getLowerBound()
    ); // Output solution.
    delete cvrp;
    return 0;
}
```

Figure 3 Usage example of branching module in RouteOpt.

variants, generated by a single round of aggregation and rounding, are particularly effective in strengthening the LP relaxation and are widely used in modern IP solvers.

The general form of a rank-1 cut applied to the set partitioning formulation of VRPs is $\sum_{r\in\Omega}\left\lfloor\sum_{i\in C} p_i a_i^r\right\rfloor \le \left\lfloor\sum_{i\in C} p_i\right\rfloor$, where Ω denotes the set of all columns, $C \subseteq N$ is the cut set (i.e., a subset of customers excluding the depot), $p_i \ge 0$ is the multiplier associated with customer $i \in C$, and a_i^r represents the number of times customer i is visited by route r. Each R1C is defined by the choice of customer subset C and a multiplier plan p, typically chosen from a set of theoretically optimal candidates. RouteOpt includes a list of such optimal multiplier plans as identified in Bulhoes et al. (2018a), which are summarized in Table 1.

Plan	Multipliers	Condition
0	$\left(\frac{1}{2}, \frac{1}{2}, \ldots, \frac{1}{2}\right)$	if n is odd
1	$\left(\frac{1}{3}, \frac{1}{3}, \ldots, \frac{1}{3}\right)$	if $n = 3k+2$ for $k \ge 1$
2	$\left(\frac{n-3}{n-2}, \frac{n-3}{n-2}, \frac{2}{n-2}, \frac{1}{n-2}, \ldots, \frac{1}{n-2}\right)$	$n \ge 5$
3	$\left(\frac{n-2}{n-1}, \frac{n-2}{n-1}, \frac{2}{n-1}, \frac{2}{n-1}, \frac{1}{n-1}, \ldots, \frac{1}{n-1}\right)$	$n \ge 5$
4	$\left(\frac{n-3}{n-1}, \frac{2}{n-1}, \frac{1}{n-1}, \ldots, \frac{1}{n-1}\right)$	$n \ge 5$
5	$\left(\frac{n-2}{n-1}, \frac{1}{n-1}, \ldots, \frac{1}{n-1}\right)$	$n \ge 4$
6	$\left(\frac{n-2}{n}, \frac{2}{n}, \frac{2}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right)$	$n \ge 5$

Table 1 Optimal multipliers for R1Cs (Bulhoes et al. 2018a)

To further enhance computational efficiency, RouteOpt incorporates the limited memory technique from Pecin et al. (2017b), which is based on the *ng*-route relaxation (Baldacci et al. 2011). This method embeds memory sets (either node-based or arc-based) into the labeling algorithm, substantially strengthening its dominance rule to accelerate the computation while preserving the cut's effectiveness. RouteOpt further employs a local search strategy adapted from Pecin et al. (2017a) to select a minimized memory set (node- or arc-based) for effective cuts without compromising pricing speed.

5.1. Configuration and Execution of R1C Generation

Figure 4 illustrates the usage of the R1C separation module in RouteOpt. In this example, a `Rank1SeparationController` is first initialized, after which the process iteratively updates the controller and generates new rank-1 cuts. The workflow primarily relies on two key functions:

- `updateInfo`: Updates the controller with the current LP solution and the set of previously generated cuts.

```
void updateInfo(MemoryType limited_memory_type,
                PRICING_HARD_LEVEL pricing_hard_level,
                bool if_collect_sol,
                const std::vector<RouteInfo>& sol,
                const std::vector<R1c>& old_cuts);
```

- `separateRank1Cuts`: Executes the rank-1 cut separation, optionally leveraging memory-based search and advanced cut selection strategies.

```
void separateRank1Cuts(std::vector<R1c>& cuts,
                       bool if_mem = true,
                       bool if_select_cuts = true);
```

These functions offer considerable flexibility through their input parameters. Users can specify the type of memory used (node-based, arc-based, or none), the difficulty level of the pricing problem, and whether to collect intermediate LP solutions. Collecting LP solutions is particularly advantageous when node-based memory becomes impractical due to pricing complexity, as it enables a seamless transition to arc-based memory. Additionally, the `separateRank1Cuts` function provides two primary operational modes: memory-based search after cut separation and cut selection by balancing violation magnitude with memory set size. The latter is especially useful when pricing is computationally intensive and careful management of memory sets is critical.

5.2. Customization and Integration of R1Cs

RouteOpt's built-in R1C separation module is designed for ease of use, allowing users to generate cuts without knowledge of the internal implementation. However, for advanced research or customization, a deeper understanding of the internal structure and management of R1Cs is essential. Such insight enables seamless integration of custom separation strategies, efficient memory set identification, and effective dominance checks within labeling algorithms.

The standard R1C representation in RouteOpt is defined as follows.

```
std::pair<std::vector<int>, int> info_r1c{};
```

In this representation, the first component stores an ordered list of customer indices, while the second component indicates the specific multiplier plan employed. Although alternative formats may be employed in the initial stages of custom separation, integration with RouteOpt's internal functions necessitates conversion to this standard representation.

```cpp
#include <iostream>
#include "rank1_data_shared.hpp"
#include "rank1_separation_controller.hpp"
#include "solver.hpp"
using namespace RouteOpt::Rank1Cuts;
using namespace RouteOpt::Rank1Cuts::Separation;

int main() {
    // Assume necessary data (shared data, solver instance, cost matrix) are already
    prepared
    Rank1CutsDataShared sharedData(/* instance dimension */);
    Solver solver; // IP solver instance
    // Instantiate Rank-1 separation controller (assume parameters are already
    defined)
    Rank1SeparationController separationController(
        sharedData, /* max_row_rank1 */, /* max_num_r1c3_per_round */,
        /* max_num_r1c_per_round */, solver, /* cost_matrix */
    );
    // Assume solutionRoutes and oldCuts are prepared from previous iterations
    std::vector<RouteInfo> solutionRoutes;
    std::vector<R1c> oldCuts;
    // Update controller with current solution and existing cuts
    separationController.updateInfo(
    MemoryType::NODE_MEMORY, PRICING_HARD_LEVEL::EASY,
    true, solutionRoutes, oldCuts
    );
    std::vector<R1c> newCuts; // Container for new R1Cs
    // Generate R1Cs
    separationController.separateRank1Cuts(newCuts, true, true);
    // Optionally convert to arc-memory representation
    separationController.convert2ArcMemory(newCuts);
    std::cout << "Number of new R1Cs generated: " << newCuts.size() << std::endl;
    return 0;
}
```

Figure 4 Example usage of the Rank-1 separation module in RouteOpt.

Figure 5 illustrates RouteOpt's standard R1C separation procedure, encompassing both the cut generation and memory assignment. These operations are handled by the classes `CutGenerator` and `MemGenerator`, respectively.

```cpp
void Rank1SeparationController::separateRank1Cuts(
        std::vector<R1c> &cuts, bool if_mem, bool if_select_cuts) {
    if (if_mem) memGen.fillMemory();
    cutGen.generateRank1Cuts();
    if (if_mem) memGen.findMemory4Cuts();
    if (if_select_cuts) cutSelector.selectR1CsByVioNMemory();
    setRhs(sharedData.refCuts());
    cuts = sharedData.refCuts();
    cleanData();
}
```

Figure 5 Implementation of Rank-1 separation in RouteOpt

For customized separation methods, users only need to redefine the `CutGenerator::generateRank1Cuts()` method with their own algorithms. Typically, such approaches require access to key data, including the current LP fractional solutions and route sequences. These details are readily available through member variables of the `CutGenerator` class.

```
std::reference_wrapper<const R1CsDataShared> r1CsDataShared_ref;
std::reference_wrapper<DataShared> sharedData_ref;
```

To ensure smooth integration and streamlined data management, it is strongly recommended that any customized separation functions be implemented directly within the `CutGenerator` class. Similarly, users can define their own memory assignment procedures by customizing the `MemGenerator::findMemory4Cuts()` method. Again, necessary data structures are conveniently accessible via the provided references `sharedData_ref` and `r1CsDataShared_ref`.

6. The Rounded Capacity Cut Module

RCCs are a class of cutting planes commonly used in capacity-constrained VRPs (e.g., the CVRP) to strengthen dual bounds derived from LP relaxations. In its most straightforward form, an RCC is expressed as $x(\bar{S} : S) \geq 2k(S)$, where S is a subset of the customer set N, and $\bar{S} = N \setminus S$. Here, $x(S,T)$ represents the undirected flow between customers in sets S and T, and $k(S)$ indicates the minimum number of vehicles necessary to serve all customers in S. The factor 2 arises naturally due to the undirected nature of this flow representation.

However, RCCs formulated in this way are typically dense and computationally expensive. To address this issue, the following sparse alternative forms (Forms 1 and 2) are typically employed:

$$\text{Form 1: } x(S : S) \leq |S| - k(S),$$

$$\text{Form 2: } x(\bar{S} : \bar{S}) + \tfrac{1}{2}x(\{0\} : \bar{S}) - \tfrac{1}{2}x(\{0\} : S) \leq |\bar{S}| - k(S).$$

According to the guidelines provided in the CVRPSEP documentation (Lysgaard 2003), Form 1 is usually preferable when $|S| \leq n/2$, whereas Form 2 is better suited for larger subsets. RouteOpt follows this recommendation by automatically selecting the sparsest formulation available. Furthermore, RouteOpt implements a strengthened variant (Baldacci et al. 2008), defined as:

$$\text{Form 3: } \sum_{r \in \Omega} I(|C(r) \cap S| \geq 1)\, x_r \geq k(S),$$

where the indicator function $I(|C(r) \cap S| \geq 1)$ is equal to 1 if route r serves at least one customer in S and 0 otherwise. Form 3, although non-robust for labeling algorithms, is particularly useful in enumerated states where all necessary columns are explicitly known.

RouteOpt's RCC module is composed of three core components:

- CVRPSEP library (Lysgaard 2003), which automates RCC generation and selection.
- Efficient mapping of RCC dual values to reduced-cost updates in labeling algorithms.
- Computation of the coefficients for RCCs in the RMP.

Detailed descriptions of these components can again be found in the comprehensive RouteOpt documentation (You 2025).

6.1. Configuration and Execution of RCC Generation

RouteOpt leverages the CVRPSEP package for RCC separation, opting not to re-implement this functionality internally. RCC separation is managed via the `RCCSeparationController` class, primarily through its `generateRCCs` method. This method generates RCCs tailored to the current problem instance, utilizing parameters such as problem size, customer demands, and the current LP solution. Its function signature is given below:

```
void generateRCCs(int dim,
                  double cap,
                  const std::vector<double>& demand,
                  bool if_keep_rcc,
                  bool if_strengthen_rcc,
                  const std::vector<double>& sol_x,
                  const std::vector<SequenceInfo>& sols,
                  const std::vector<Rcc>& old_cuts,
                  std::vector<Rcc>& new_cuts);
```

The following two key boolean parameters provide crucial flexibility for RCC generation.

- `if_keep_rcc`: Specifies whether generated RCCs should remain active (permanent) throughout the entire solution process. This option is particularly valuable for VRPTW type-2 instances, where relaxing capacity constraints during pricing may yield integer solutions that violate these constraints. By retaining these RCCs permanently, RouteOpt effectively prevents repeated generation of the same infeasible solutions.
- `if_strengthen_rcc`: Forces the use of strengthened RCCs, which should be enabled in enumerated states.

Figure 6 demonstrates practical usage of the RCC separation module in RouteOpt. In this example, essential input parameters such as the problem dimension, vehicle capacity, and customer demands are initialized. The `generateRCCs` function is then called with key settings, controlling, for example, whether generated cuts should remain active throughout the LP solution process, and whether strengthened RCCs are to be used.

```cpp
#include <vector>
#include <iostream>
#include "rcc_separation_controller.hpp"
using namespace RouteOpt;

int main() {
    int dimension = /* problem dimension */;
    double capacity = /* vehicle capacity */;
    std::vector<double> demand(dimension);
    // Fill in the demand vector appropriately

    std::vector<double> solutionX; // Fractional LP solution
    std::vector<SequenceInfo> solutionRoutes; // Routes from current solution
    std::vector<Rcc> previousRccs; // Existing RCCs
    std::vector<Rcc> newRccs; // Newly generated RCCs

    // Generate RCCs
    RCCSeparationController::generateRCCs(
        dimension,
        capacity,
        demand,
        false,  // Generated RCCs not permanently retained
        true,   // Use strengthened RCC (Form 3)
        solutionX,
        solutionRoutes,
        previousRccs,
        newRccs
    );

    // Output number of generated RCCs
    std::cout << "Generated RCCs: " << newRccs.size() << std::endl;

    return 0;
}
```

Figure 6 Example usage of the RCC separation module in RouteOpt.

6.2. Customization and Integration of RCCs

As with R1Cs, users can customize RCC separation strategies within RouteOpt. For compatibility with RouteOpt's RCC modules, custom implementations must adhere to the standard RCC data structure, as summarized in Figure 7.

```cpp
// RCC data structure
int form_rcc{}; // Specifies RCC formulation type
std::vector<int> info_rcc_customer{}; // Customer set S
std::vector<int> info_rcc_outside_customer{}; // Complement of set S
double rhs{}; // RHS value of the RCC
bool if_keep{}; // Flag indicating persistence across solution process
```

Figure 7 Standard RCC data structure in RouteOpt.

In this structure, the vectors `info_rcc_customer` and `info_rcc_outside_customer` represent the customer set S and its complement $\bar{S}$, respectively. For RCCs in Form 1 and

Form 3, where the complement set is not explicitly required, `info_rcc_outside_customer` may remain empty. To implement a custom RCC separator, simply modify the body of `RCCSeparationController::generateRCCs()` as needed. Each new cut should specify the relevant fields in the standard structure, ensuring seamless integration with RouteOpt.

7. The DeLuxing Module

Deep Lagrangian Underestimate Fixing (DeLuxing), proposed by Yang (2025), is an advanced variable reduction method designed specifically for exact column-generation-based optimization methods. By leveraging LP dual solutions, DeLuxing efficiently identifies and eliminates up to 99% of unnecessary variables, achieving substantial acceleration. In routing problems, DeLuxing is typically used in the following two ways.

- Column Pool Reduction: Rapidly pruning variables from fully enumerated pools, expediting the certification of optimality for challenging instances.
- Primal Heuristics: Enumerating a manageable, high-quality set of columns by applying stringent cut-off criteria, followed by DeLuxing to eliminate unnecessary variables. The resulting pool, typically containing 10,000 to 20,000 columns, is then sorted according to the reduced costs from the final DeLuxing iteration. The top-ranked columns can be efficiently provided to an IP solver, significantly improving the chances of rapidly finding high-quality feasible solutions.

7.1. Configuration and Execution of Variable Reduction with DeLuxing

Figure 8 provides a practical demonstration of employing DeLuxing within RouteOpt. The example showcases how users can specify essential parameters to effectively control the variable reduction process. The key parameters influencing DeLuxing performance are:

```
int NClust = 20;    // Number of clusters for grouping columns
int beta1 = 1000;   // Initial heuristic threshold
int beta2 = 1000;   // Deep-search heuristic threshold
```

In particular, the parameter `NClust` determines the number of clusters into which variables are partitioned; increasing this value typically results in the removal of more variables but may require additional computational time. The parameter `beta1` dictates when to use the slower yet more accurate k-means++ clustering algorithm: if the column pool size is less than or equal to `beta1`, k-means++ is employed; otherwise, a faster, less precise heuristic method is used. Finally, `beta2` controls the depth of the iterative variable elimination process: a smaller value enables more exhaustive exploration and potentially greater variable reduction, albeit at the expense of increased runtime.

```cpp
#include "deluxing.hpp"
#include "solver.hpp"
using namespace RouteOpt::DeLuxing;

int main() {
    Solver solver;
    double UB = 100.0;          // Upper bound of the solution
    int NClust = 20;            // Number of clusters for column grouping
    int beta1 = 1000;           // Threshold for initial removal heuristic
    int beta2 = 1000;           // Threshold for deep-search removal
    std::vector<int> Idxdel;    // Indices of removed variables
    double Timelimit = 3600.0;  // 1-hour computation limit
    double Tolerance = 1e-6;    // Numerical tolerance threshold
    bool Verbose = true;        // Enable detailed logging

    DeLuxingController::deLuxing(
        solver, UB, NClust, beta1, beta2,
        Idxdel, Timelimit, Tolerance, Verbose
    );

    return 0;
}
```

Figure 8 Example usage of DeLuxing in RouteOpt.

8. Computational Results

This section presents a comprehensive evaluation of RouteOpt's performance on two classic VRPs, the CVRP and VRPTW. The goal is to demonstrate the solver's efficiency through computational experiments conducted using well-established benchmark datasets. All the code and datasets can be accessed at https://github.com/Zhengzhong-You/IJOC_JUN20_2025.

8.1. Experimental Setup

The experimental evaluation leverages two distinct computing setups: the HiPerGator supercomputing cluster at the University of Florida and four identical high-performance Dell workstations. HiPerGator is equipped with nodes running Red Hat Enterprise Linux 8.8, each featuring two AMD EPYC 7702 64-core processors at 2.0 GHz and 1003 GB of RAM. Our group has access to 1080 CPU cores and 8640 GB RAM, enabling the large-scale parallel computations necessary to simultaneously run hundreds of jobs and achieve several CVRP optimality results for the first time (see Section 8.2). For precise time comparisons, identical Dell workstations are used, each featuring a 12th Gen Intel(R) Core(TM) i9-12900K CPU at 3.2 GHz with 128 GB of RAM, running Ubuntu 20.04 LTS. All code was compiled using g++ version 9.4.0.

8.2. Performance on CVRP Benchmarks

The CVRP involves routing a fleet of vehicles from a single depot to deliver goods to a set of customers, each exactly once, while minimizing total travel distance and respecting vehicle capacity

constraints. The foundational research on CVRP can be traced to the seminal work of Dantzig and Ramser (1959), which laid the groundwork for systematic studies in vehicle routing. Due to its pivotal role in transportation logistics and its academic significance, the CVRP has inspired numerous variants and remains a benchmark problem in operations research. Standard benchmark datasets for the CVRP, available via CVRPLIB (Lima et al. 2014), include sets A, B, E, F, M, P, and X. In this study, we consider instances with 60 to 199 customers, categorized into two groups: *Small* (60–99 customers) and *Medium* (100–199 customers).

We benchmark the computational performance of RouteOpt against VRPSolver v0.6.13 with BaPCod v0.82 and CPLEX 22.1.0 (IBM Corporation 2023) as the internal LP/IP solver. VRPSolver is a state-of-the-art exact CVRP solver developed by Pessoa et al. (2020). Both RouteOpt and VRPSolver are initialized with optimal solution values as upper bounds (UB) and a time limit of 3600 seconds (1 hour). Performance is assessed using the following metrics: optimality status (Opt), root node processing time in seconds (Root/s), root node gap percentage (Root Gap/%), total computation time in seconds (CPU/s), number of explored nodes (#Nodes), and final global gap percentage (Gap/%). Table 2 summarize these metrics for both the Small and Medium groups, reporting the gap as an average and other metrics as geometric means. Detailed numerical results are provided in Table 5 in the Appendix.

The results demonstrate that RouteOpt consistently outperforms VRPSolver. For the Small group, RouteOpt optimally solves all 22 instances, processing root nodes and proving optimality faster than VRPSolver. For the Medium group, RouteOpt solves 27 out of 30 instances optimally, while VRPSolver solves 25. Additionally, RouteOpt achieves substantially lower overall CPU times (78.5 s versus 125.1 s) and a smaller average final optimality gap (0.02% versus 0.04%) at termination.

	RouteOpt						VRPSolver					
Group	Opt	Root/s	Root Gap (%)	CPU/s	#Nodes	Gap (%)	Opt	Root/s	Root Gap (%)	CPU/s	#Nodes	Gap (%)
Small	22/22	2.5	0.00	2.5	1.0	0.00	22/22	3.7	0.00	4.3	1.0	0.00
Medium	27/30	36.4	0.11	78.5	3.4	0.02	25/30	62.0	0.11	125.1	3.0	0.04

Table 2 Summary of performance on CVRP: RouteOpt vs. VRPSolver

8.3. Performance on VRPTW Benchmarks

The VRPTW extends the standard CVRP by adding an additional resource constraint related to time. Specifically, each customer is associated with a service time $s_i \geq 0$ and an earliest and latest

service starting time $e_i \geq 0$ and a$l_i \geq 0$, where $e_i \leq l_i$. Service at each customer must begin within this time window. Vehicles may wait if they arrive before e_i, but must not arrive after l_i. The travel time from customer i to customer j, denoted as $t_{ij} \geq 0$, is assumed equal to the corresponding distance d_{ij} in widely used benchmark instances.

Our evaluation employs Solomon's classic benchmark dataset(Solomon 1987), which consists of 56 instances with 100 customers each, and the HG-200 dataset, comprising 60 instances with 200 customers. Both datasets are publicly available from CVRPLIB (Lima et al. 2014). To ensure meaningful comparison, we excluded instances that both VRPSolver and RouteOpt solved in less than 60 seconds. The time limit is again set to 3600 seconds. After filtering, the test set consisted of 8 Solomon instances and 51 HG-200 instances. For consistency, we report the same performance metrics as in the CVRP benchmarking.

As summarized in Table 3, RouteOpt consistently outperforms VRPSolver on both datasets. For the Solomon set, RouteOpt solves all 8 instances to optimality within the one-hour limit, while VRPSolver solves 7 out of 8. RouteOpt's average solution time is 68.6 seconds per instance, less than half of VRPSolver's 167.7 seconds, demonstrating a significant speed advantage. Moreover, RouteOpt achieves a better root node gap (0.02% compared to 0.05%) in less CPU time (57.6 s versus 143.5 s). On the more challenging HG-200 dataset, RouteOpt again demonstrates superior performance, solving 38 out of 50 instances optimally compared to 32 for VRPSolver. RouteOpt achieves a remarkably low average optimality gap of just 0.18%, significantly outperforming VRP-Solver's 0.94%. Additionally, RouteOpt processes root nodes more quickly (229.9 s versus 389.1 s), maintains a smaller root gap (0.34% versus 1.06%), and keeps total CPU time lower (592.9 s versus 720.1 s), even though it explores more nodes per instance (5.4 compared to 2.7 for VRPSolver). Detailed numerical results are provided in Table 6 in the Appendix.

	RouteOpt						VRPSolver					
Group	Opt	Root/s	Root Gap (%)	CPU/s	#Nodes	Gap (%)	Opt	Root/s	Root Gap (%)	CPU/s	#Nodes	Gap (%)
Solomon	8/8	57.6	0.02	68.6	1.3	0.00	7/8	143.5	0.05	167.7	1.3	0.03
HG	38/50	229.9	0.34	592.9	5.4	0.18	32/50	389.1	1.06	720.1	2.7	0.94

Table 3 Summary of performance on VRPTW: RouteOpt vs. VRPSolver

8.4. Solution of Open CVRP Instances

In this section, we aim to address the challenge of proving optimality for several previously unsolved instances using RouteOpt's novel node restoration feature. Specifically, we target the well-known and challenging CVRP instances X-n313-k71, X-n359-k29, and X-n384-k52. Proving optimal solutions for these problem instances on a single machine would typically require an impractical amount of computational effort, even when using the best-known value as an upper bound. To ensure a clear analysis of any identified optimal solution, we initialize the upper bound u to the best-known solution value v^* plus one, i.e., $u = v^* + 1$. Table 4 presents our results, reporting key metrics such as root node processing time (Root/s), root node gap percentage (Root Gap/%), total computing time in CPU years (CPU/y), and the total number of explored nodes (#Nodes/10^6), aggregated from all solution logs. For the node yielding the optimal solution, we further report its branching structure, specifying the total number of branches taken as well as the respective counts of left (edges forbidden) and right (edges required) branches from the root to this optimal solution.

Notably, the root node gaps obtained are impressively small (ranging from 0.22% to 0.29%), with root node solution times of just a few hundred seconds. This clearly demonstrates the effectiveness of RouteOpt's advanced cutting and reduced-cost fixing techniques in tightening dual bounds at the root. However, despite achieving such narrow root gaps rapidly, proving optimality with conventional methods remains computationally prohibitive, often requiring one or more CPU year. This challenge arises from the need to explore millions of branch-and-bound nodes, even when employing advanced branching strategies such as 2LBB combined with BKF. Fortunately, RouteOpt's node restoration capability enables a dramatic reduction in wall time when utilizing supercomputing resources like the HiPerGator cluster, cutting the runtime for each instance down to just a few days.

Additionally, we observe a clear relationship between the route length, defined as the average number of customers visited per vehicle, and the number of nodes explored. Instances with shorter average route lengths tend to require the exploration of more BBNs. For example, instance X-n313-k71, with an average route length of approximately $313/71 \approx 4.4$, required exploring about 0.71 million nodes. In contrast, instance X-n384-k52, with average route length≈ 7.4, required fewer nodes (0.62 million) and instance X-n359-k29, with the longest average route length of approximately 12.4, required just 0.04 million nodes. An additional noteworthy pattern is observed in the branching structure: across all three instances, more left branches are consistently needed to reach optimality. Recognizing this branching tendency could offer valuable strategic insights for expediting the search for optimal or near-optimal solutions.

Instance	Root/s	Root Gap/%	CPU/y	#Nodes/10^6	#Total Branches	#Left Branches	#Right Branches
X-n313-k71	149.66	0.22%	0.49	0.71	33	17	16
X-n359-k29	926.34	0.24%	0.36	0.04	17	9	8
X-n384-k52	575.01	0.29%	1.50	0.62	26	16	10

Table 4 Performance on open CVRP instances

9. Conclusions

This paper introduces RouteOpt, an advanced, modular, and open-source solver specifically designed to tackle the challenges of solving VRPs exactly. RouteOpt's modular architecture not only streamlines development by removing the need for building intricate components from the ground up, but also empowers researchers to focus on refining targeted functionalities. Its open framework encourages innovation and user-driven customization, making it an ideal platform for experimenting with novel techniques. RouteOpt also serves as a comprehensive benchmarking platform, capable of analyzing and decoding branching patterns in large-scale VRP instances. These insights can directly inform the design of new branching methods and optimization strategies.

We believe RouteOpt will lower the barrier to entry for researchers and practitioners interested in BPC-based methods for VRPs, encouraging broader adoption of exact approaches in the field. By relieving users of repetitive implementation burdens, RouteOpt enables greater focus on creative algorithmic development and scientific exploration. Ultimately, we hope that RouteOpt will serve as both a research accelerator and a catalyst for future advances in exact combinatorial optimization.

Acknowledgments

This work is partially supported by the National Science Foundation [Grant CMMI-2309667].

References

Baldacci R, Christofides N, Mingozzi A (2008) An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming* 115:351–385.

Baldacci R, Mingozzi A, Roberti R (2011) New route relaxation and pricing strategies for the vehicle routing problem. *Operations Research* 59(5):1269–1283.

Balster I, Bulhões T, Munari P, Pessoa AA, Sadykov R (2023) A new family of route formulations for split delivery vehicle routing problems. *Transportation Science* 57(5):1359–1378.

Bulhoes T, Pessoa A, Protti F, Uchoa E (2018a) On the complete set packing and set partitioning polytopes: Properties and rank 1 facets. *Operations Research Letters* 46(4):389–392.

Bulhoes T, Sadykov R, Uchoa E (2018b) A branch-and-price algorithm for the minimum latency problem. *Computers & Operations Research* 93:66–78.

Chvátal V (1973) Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics* 4(4):305–337.

Dantzig GB, Ramser JH (1959) The truck dispatching problem. *Management Science* 6(1):80–91.

Errami N, Queiroga E, Sadykov R, Uchoa E (2024) Vrpsolvereasy: a python library for the exact solution of a rich vehicle routing problem. *INFORMS Journal on Computing* 36(4):956–965.

IBM Corporation (2023) IBM ILOG CPLEX Optimization Studio CPLEX User's Manual. URL https://www.ibm.com/docs/en/icos/22.1.0?topic=manuals-cplex-users-manual.

Irnich S, Desaulniers G, Desrosiers J, Hadjar A (2010) Path-reduced costs for eliminating arcs in routing and scheduling. *INFORMS Journal on Computing* 22(2):297–313.

Kim H, Park J, Kwon C (2024) A neural separation algorithm for the rounded capacity inequalities. *INFORMS Journal on Computing* 36(4):987–1005.

Lima I, Uchoa E, Pecin D, Pessoa A, Poggi M, Vidal T, Subramanian A, Oliveira D, Queiroga E (2014) Cvrplib: Capacitated vehicle routing problem library. URL http://vrp.galgos.inf.puc-rio.br/index.php/en/updates.

Lysgaard J (2003) CVRPSEP: A package of separation routines for the capacitated vehicle routing problem. Technical report, Aarhus School of Business, Department of Management Science and Logistics.

Lysgaard J, Letchford AN, Eglese RW (2004) A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming* 100:423–445.

Martinelli R, Pecin D, Poggi M (2014) Efficient elementary and restricted non-elementary route pricing. *European Journal of Operational Research* 239(1):102–111.

Morabit M, Desaulniers G, Lodi A (2021) Machine-learning–based column selection for column generation. *Transportation Science* 55:815–831.

Naddef D, Rinaldi G (2002) Branch-and-cut algorithms for the capacitated vrp. *The vehicle routing problem*, 53–84 (SIAM).

Pecin D, Contardo C, Desaulniers G, Uchoa E (2017a) New enhancements for the exact solution of the vehicle routing problem with time windows. *INFORMS Journal on Computing* 29(3):489–502.

Pecin D, Pessoa A, Poggi M, Uchoa E, Santos H (2017b) Limited memory rank-1 cuts for vehicle routing problems. *Operations Research Letters* 45(3):206–209.

Pessoa A, Sadykov R, Uchoa E (2021) Solving bin packing problems using vrpsolver models. *Operations Research Forum*, 20 (Springer).

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

Righini G, Salani M (2006) Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization* 3(3):255–273.

Roberti R, Mingozzi A (2014) Dynamic ng-path relaxation for the delivery man problem. *Transportation Science* 48(3):413–424.

Sadykov R, Uchoa E, Pessoa A (2021) A bucket graph–based labeling algorithm with application to vehicle routing. *Transportation Science* 55(1):4–28.

Sadykov R, Vanderbeck F (2021) *BaPCod-a generic branch-and-price code*. Ph.D. thesis, Inria Bordeaux Sud-Ouest.

Silva JMP, Uchoa E, Subramanian A (2024) Cluster branching for vehicle routing problems. *Optimization Online* URL https://optimization-online.org/?p=27150.

Solomon MM (1987) Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research* 35(2):254–265.

Vidal T (2022) Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *Computers & Operations Research* 140:105643.

Vidal T, Crainic TG, 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 TG, Gendreau M, Prins C (2013) A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & Operations Research* 40(1):475–489.

Wouda NA, Lan L, Kool W (2024) Pyvrp: A high-performance vrp solver package. *INFORMS Journal on Computing* .

Yang Y (2023) An exact price-cut-and-enumerate method for the capacitated multitrip vehicle routing problem with time windows. *Transportation Science* 57(1):230–251.

Yang Y (2025) Deluxing: Deep lagrangian underestimate fixing for column-generation-based exact methods. *Operations Research* 73(3):1184–1207.

You Z (2025) Routeopt documentation. Available at https://zhengzhong-you.github.io/RouteOpt-Docs/rank1_cuts/r1c_index.html#id72.

You Z, Yang Y, Wang X, Yin W (2023) Two-stage learning to branch in branch-price-and-cut algorithms for solving vehicle routing problems exactly. *Available at SSRN 4630549* .

Appendix. Detailed Numerical Results

This appendix presents detailed numerical results on CVRP (Table 5) and VRPTW (Table 6). Solver performance is evaluated by several metrics: optimality status (Opt; 1 for optimal and 0 otherwise), root node processing time in seconds (Root/s), root node gap (Root Gap/%), total CPU computation time in seconds (CPU/s), the number of explored nodes (#Nodes), and final global gap (Gap/%). The last two rows summarize performance metrics, reporting the gap as an arithmetic average and all other metrics as geometric means.

Instance	RouteOpt						VRPSolver					
	Opt	Root/s	Root Gap/%	CPU/s	#Nodes	Gap/%	Opt	Root/s	Root Gap/%	CPU/s	#Nodes	Gap/%
A-n62-k8	1	1.9	0.00	1.9	1	0.00	1	6.0	0.00	6.1	1	0.00
A-n63-k10	1	1.6	0.00	1.6	1	0.00	1	2.0	0.00	3.0	1	0.00
A-n63-k9	1	1.3	0.00	1.3	1	0.00	1	3.0	0.00	3.5	1	0.00
A-n64-k9	1	2.4	0.00	2.4	1	0.00	1	5.0	0.00	5.2	1	0.00
A-n65-k9	1	1.0	0.00	1.0	1	0.00	1	1.0	0.00	1.9	1	0.00
A-n69-k9	1	1.7	0.00	1.7	1	0.00	1	3.0	0.00	3.2	1	0.00
A-n80-k10	1	2.7	0.00	2.7	1	0.00	1	5.0	0.00	5.5	1	0.00
B-n63-k10	1	1.3	0.00	1.3	1	0.00	1	1.0	0.00	1.9	1	0.00
B-n64-k9	1	1.7	0.00	1.7	1	0.00	1	1.0	0.00	1.3	1	0.00
B-n66-k9	1	18.1	0.00	18.1	1	0.00	1	4.0	0.00	4.8	1	0.00
B-n67-k10	1	1.4	0.00	1.4	1	0.00	1	3.0	0.00	3.4	1	0.00
B-n68-k9	1	8.8	0.00	8.8	1	0.00	1	12.0	0.00	12.8	1	0.00
B-n78-k10	1	2.2	0.00	2.2	1	0.00	1	4.0	0.00	4.1	1	0.00
E-n76-k10	1	2.7	0.00	2.7	1	0.00	1	6.0	0.00	6.1	1	0.00
E-n76-k14	1	1.7	0.00	1.7	1	0.00	1	2.0	0.00	2.4	1	0.00
E-n76-k7	1	3.6	0.00	3.6	1	0.00	1	5.0	0.00	5.9	1	0.00
E-n76-k8	1	2.4	0.00	2.4	1	0.00	1	9.0	0.00	9.1	1	0.00
E-n101-k14	1	5.6	0.00	5.6	1	0.00	1	8.0	0.00	8.6	1	0.00
E-n101-k8	1	15.9	0.00	15.9	1	0.00	1	22.0	0.00	22.7	1	0.00
F-n72-k4	1	8.4	0.00	8.4	1	0.00	1	12.0	0.00	12.1	1	0.00
F-n135-k7	1	1646.5	0.06	1909.6	3	0.00	0	2737.0	0.43	3600.0	3	0.43
M-n101-k10	1	1.8	0.00	1.8	1	0.00	1	1.0	0.00	1.2	1	0.00
M-n121-k7	1	11.2	0.00	11.2	1	0.00	1	20.0	0.00	20.8	1	0.00
M-n151-k12	1	24.5	0.00	24.5	1	0.00	1	61.0	0.00	61.4	1	0.00
M-n200-k16	1	214.6	0.57	1266.6	43	0.00	1	227.0	0.47	2355.7	29	0.00
P-n65-k10	1	0.7	0.00	0.7	1	0.00	1	1.0	0.00	1.3	1	0.00
P-n70-k10	1	1.9	0.00	1.9	1	0.00	1	3.0	0.00	3.3	1	0.00
P-n76-k4	1	4.7	0.00	4.7	1	0.00	1	6.0	0.00	6.2	1	0.00
P-n76-k5	1	7.7	0.00	7.7	1	0.00	1	15.0	0.00	15.7	1	0.00
P-n101-k4	1	23.2	0.00	23.2	1	0.00	1	39.0	0.00	40.0	1	0.00
X-n101-k25	1	1.4	0.00	1.4	1	0.00	1	2.0	0.00	2.8	1	0.00
X-n106-k14	1	11.4	0.00	11.4	1	0.00	1	21.0	0.00	21.6	1	0.00
X-n110-k13	1	3.2	0.00	3.2	1	0.00	1	5.0	0.00	5.7	1	0.00
X-n115-k10	1	18.6	0.00	18.6	1	0.00	1	29.0	0.00	29.7	1	0.00
X-n120-k6	1	98.0	0.36	765.8	9	0.00	1	482.0	0.29	2705.9	9	0.00
X-n125-k30	1	18.3	0.06	51.1	3	0.00	1	35.0	0.00	35.2	1	0.00
X-n129-k18	1	16.8	0.12	51.8	3	0.00	1	36.0	0.10	78.2	5	0.00
X-n134-k13	1	303.3	0.29	1594.3	11	0.00	0	96.0	0.61	3600.0	41	0.33
X-n139-k10	1	73.0	0.00	73.0	1	0.00	1	174.0	0.00	174.9	1	0.00
X-n143-k7	0	385.0	0.35	3600.0	24	0.19	0	1024.0	0.30	3600.0	9	0.19
X-n148-k46	1	3.5	0.00	3.5	1	0.00	1	6.0	0.00	6.1	1	0.00
X-n153-k22	1	160.6	0.10	441.4	13	0.00	1	310.0	0.08	442.4	11	0.00
X-n157-k13	1	16.2	0.00	16.2	1	0.00	1	20.0	0.00	20.7	1	0.00
X-n162-k11	1	254.4	0.21	611.2	5	0.00	1	975.0	0.05	1264.7	3	0.00
X-n167-k10	1	195.9	0.00	195.9	1	0.00	1	403.0	0.00	403.5	1	0.00
X-n172-k51	1	37.1	0.11	53.1	3	0.00	1	34.0	0.11	49.7	3	0.00
X-n176-k26	1	30.1	0.18	72.4	7	0.00	1	70.0	0.12	268.8	13	0.00
X-n181-k23	1	33.9	0.13	111.9	13	0.00	1	35.0	0.15	260.7	15	0.00
X-n186-k15	0	176.6	0.40	3600.0	23	0.18	0	423.0	0.32	3600.0	11	0.25
X-n190-k8	0	309.9	0.17	3600.0	15	0.11	0	1464.0	0.14	3600.0	5	0.13
X-n195-k51	1	21.1	0.17	46.3	5	0.00	1	28.0	0.14	60.5	3	0.00
X-n200-k36	1	40.1	0.08	447.4	79	0.00	1	87.0	0.08	603.8	37	0.00
Small	22/22	2.5	0.00	2.5	1.0	0.00	22/22	3.7	0.00	4.3	1.0	0.00
Medium	27/30	36.4	0.11	78.5	3.4	0.02	25/30	62.0	0.11	125.1	3.0	0.04

Table 5 Performance of CVRP for RouteOpt and VRPSolver

Instance	RouteOpt						VRPSolver					
	Opt	Root/s	Root Gap/%	CPU/s	#Nodes	Gap/%	Opt	Root/s	Root Gap/%	CPU/s	#Nodes	Gap/%
C204	1	36.8	0.00	36.8	1	0.00	1	98.2	0.00	98.2	1	0.00
R204	1	64.7	0.00	64.7	1	0.00	1	83.0	0.00	83.0	1	0.00
R208	1	558.2	0.20	2248.4	9	0.00	0	1034.6	0.42	3600.0	7	0.21
R209	1	30.0	0.00	30.0	1	0.00	1	82.5	0.00	82.5	1	0.00
R210	1	42.7	0.00	42.7	1	0.00	1	70.5	0.00	70.5	1	0.00
R211	1	32.5	0.00	32.5	1	0.00	1	171.5	0.00	171.5	1	0.00
RC204	1	46.7	0.00	46.7	1	0.00	1	127.1	0.00	127.1	1	0.00
RC208	1	47.2	0.00	47.2	1	0.00	1	167.9	0.00	167.9	1	0.00
C1_2_3	1	95.3	0.12	102.4	3	0.00	1	75.3	0.00	75.3	1	0.00
C1_2_4	1	113.7	0.00	113.7	1	0.00	1	98.4	0.00	98.4	1	0.00
C2_2_1	1	26.2	0.00	26.2	1	0.00	1	94.1	0.00	94.1	1	0.00
C2_2_2	1	87.6	0.17	171.2	3	0.00	1	115.7	0.00	115.7	1	0.00
C2_2_3	0	1261.5	0.47	3600.0	8	0.31	0	262.4	1.13	3600.0	17	0.91
C2_2_4	0	620.0	1.53	3600.0	9	1.37	0	299.2	2.48	3600.0	17	2.26
C2_2_5	1	83.1	0.09	87.1	3	0.00	1	88.5	0.00	88.5	1	0.00
C2_2_6	1	124.3	0.00	124.3	1	0.00	1	153.5	0.00	153.5	1	0.00
C2_2_7	1	65.0	0.00	65.0	1	0.00	1	64.9	0.00	64.9	1	0.00
C2_2_8	1	78.7	0.00	78.7	1	0.00	1	81.2	0.00	81.2	1	0.00
C2_2_9	1	207.5	0.00	207.5	1	0.00	1	137.9	0.00	137.9	1	0.00
C2_2_10	1	269.2	0.00	269.2	1	0.00	1	249.6	0.00	249.7	1	0.00
R1_2_1	1	5.2	0.00	5.2	1	0.00	1	10.4	0.15	71.5	27	0.00
R1_2_3	0	117.2	0.85	3600.0	79	0.34	0	408.0	0.78	3600.0	11	0.61
R1_2_4	0	159.1	0.55	3600.0	146	0.28	0	787.4	0.49	3600.0	9	0.36
R1_2_5	1	33.8	0.27	80.9	11	0.00	1	51.1	0.18	131.0	5	0.00
R1_2_6	1	127.6	0.47	816.2	53	0.00	1	161.1	0.39	1278.3	37	0.00
R1_2_7	1	99.8	0.11	150.4	3	0.00	1	215.6	0.00	215.6	1	0.00
R1_2_8	1	212.5	0.26	380.1	7	0.00	1	1245.3	0.17	1708.5	7	0.00
R1_2_9	1	89.3	0.29	202.4	17	0.00	1	75.5	0.20	198.6	9	0.00
R1_2_10	1	89.3	0.56	851.2	61	0.00	1	188.4	0.44	1240.4	35	0.00
R2_2_2	1	130.9	0.00	130.9	1	0.00	1	150.1	0.00	150.1	1	0.00
R2_2_3	1	731.6	0.04	956.6	3	0.00	1	2456.8	0.00	2456.8	1	0.00
R2_2_4	1	886.6	0.20	3559.7	29	0.00	0	3600.0	0.54	3600.0	1	0.54
R2_2_5	1	254.3	0.03	259.3	3	0.00	1	385.0	0.00	385.0	1	0.00
R2_2_6	1	527.2	0.06	680.1	3	0.00	1	1307.8	0.00	1307.8	1	0.00
R2_2_7	1	1532.5	0.00	1532.5	1	0.00	0	3600.0	0.81	3600.0	1	0.81
R2_2_8	0	3466.4	0.26	3600.0	2	0.43	0	3600.0	2.98	3600.0	1	2.98
R2_2_9	1	186.8	0.09	342.5	5	0.00	1	320.5	0.00	320.5	1	0.00
R2_2_10	1	260.6	0.21	394.2	3	0.00	1	686.9	0.16	936.4	3	0.00
RC1_2_1	1	67.2	0.52	215.1	31	0.00	1	63.2	0.45	405.0	19	0.00
RC1_2_2	1	80.8	0.59	644.3	65	0.00	1	223.8	0.46	641.6	9	0.00
RC1_2_3	0	148.2	0.80	3600.0	17	0.46	0	1055.5	0.52	3600.0	5	0.34
RC1_2_4	1	344.7	0.43	2935.4	13	0.00	0	1017.1	0.42	3600.0	9	0.25
RC1_2_5	1	191.4	0.54	1502.3	29	0.00	1	536.2	0.36	1118.9	15	0.00
RC1_2_6	1	107.8	0.25	273.1	11	0.00	1	173.1	0.16	320.2	3	0.00
RC1_2_7	0	264.4	1.04	3600.0	12	0.65	0	869.5	0.67	3600.0	5	0.39
RC1_2_8	1	332.9	0.45	3527.7	21	0.00	1	614.2	0.32	1361.5	3	0.00
RC1_2_9	0	150.6	1.39	3600.0	16	0.91	0	965.9	1.00	3600.0	5	0.84
RC1_2_10	0	264.8	0.91	3600.0	13	0.67	0	954.3	0.70	3600.0	5	0.53
RC2_2_1	1	64.8	0.00	64.8	1	0.00	1	51.3	0.00	51.3	1	0.00
RC2_2_2	1	448.8	0.23	2887.4	21	0.00	1	803.1	0.22	2152.4	5	0.00
RC2_2_3	1	1939.7	0.05	2593.4	5	0.00	0	3600.0	2.05	3600.0	1	2.05
RC2_2_4	0	2441.7	0.65	3600.0	3	0.73	0	3600.0	27.72	3600.0	1	27.72
RC2_2_5	1	341.1	0.00	341.1	1	0.00	1	187.7	0.00	187.8	1	0.00
RC2_2_6	1	301.9	0.00	301.9	1	0.00	1	226.7	0.00	226.7	1	0.00
RC2_2_7	1	312.2	0.42	2071.3	21	0.00	0	1393.2	0.59	3600.0	5	0.21
RC2_2_8	1	895.6	0.00	1731.6	5	0.00	0	3600.0	0.51	3600.0	1	0.51
RC2_2_9	0	3600.0	1.06	3600.0	1	1.06	0	3600.0	2.57	3600.0	1	2.57
RC2_2_10	0	3478.8	1.18	3600.0	2	1.69	0	3600.0	3.22	3600.0	1	3.22
Solomon	8/8	57.6	0.02	68.6	1.3	0.00	7/8	143.5	0.05	167.7	1.3	0.03
HG	38/50	229.9	0.34	592.9	5.4	0.18	32/50	389.1	1.06	720.1	2.7	0.94

Table 6 Performance of VRPTW for RouteOpt and VRPSolver