VRPSolverEasy: a Python library for the exact solution of a rich vehicle routing problem

Najib Errami, Eduardo Queiroga, Ruslan Sadykov, Eduardo Uchoa

VRPSolverEasy: a Python library for the exact solution of a rich vehicle routing problem

Najib Errami*[1], Eduardo Queiroga†[1], Ruslan Sadykov‡[1], and Eduardo Uchoa§[1,2]

[1]Inria Centre of the University of Bordeaux – Sud-Ouest, 200 Avenue de la Vieille Tour, Talence 33405, France
[2]Engenharia de Produção, University Federal Fluminense, Niteroi-RJ, 24210-240, Brazil

4 April 2023

Abstract

The optimization community has made significant progress in solving Vehicle Routing Problems (VRPs) to optimality using sophisticated Branch-Cut-and-Price (BCP) algorithms. VRPSolver is a BCP algorithm with excellent performance in many VRP variants. However, its complex underlying mathematical model makes it hardly accessible to routing practitioners. To address this, VRPSolverEasy provides a Python interface to VRPSolver that does not require any knowledge of Mixed Integer Programming modeling. Instead, routing problems are defined in terms of familiar elements such as depots, customers, links, and vehicle types. VRPSolverEasy can handle several popular VRP variants and arbitrary combinations of them.

1 Introduction

In recent years the optimization community has accomplished significant advances in its ability to solve Vehicle Routing Problems (VRPs) to optimality (Costa et al., 2019). Complex Branch-Cut-and-Price (BCP) algorithms for some of the most classical VRP variants now solve many instances with up to a few hundred customers. Heuristic solvers are also very advanced and can obtain good quality solutions even for larger instances in a short time. They are likely to remain the best option in most VRP real-world applications. However, the newly improved exact algorithms may have important uses: (1) while the solution of instances with more than 200 customers may take a very long time (hours or days), instances with less than 100 customers are often solvable in a few minutes, which is quite reasonable in many cases; (2) even if one wants to build a heuristic, knowing true optimal solutions for a set of medium-sized instances is important for benchmarking purposes.

VRPSolver (Pessoa et al., 2020) is a BCP algorithm that solves a generic model encompassing a wide range of VRPs. It incorporates most of the key elements found in the best existing BCPs for particular VRP variants, including pricing by a sophisticated labeling algorithm (Sadykov et al., 2021), ng-path relaxation (Baldacci et al., 2011), rank-1 cuts with limited memory (Jepsen et al., 2008; Pecin et al., 2017a,b,c; Bulhoes et al., 2018), rounded capacity cuts (Laporte and Nobert, 1983; Lysgaard et al., 2004), path enumeration (Baldacci et al., 2008; Contardo and

*najib.errami@inria.fr
†eduardovqueiroga@gmail.com
‡rrsadykov@gmail.com
§uchoa@producao.uff.br

Martinelli, 2014), hierarchical strong branching (Røpke, 2012; Pecin et al., 2017b), and non-robustness control (Pecin et al., 2017b). Extensive experiments have shown excellent overall performance. In fact, in many of the most classic VRP variants, VRPSolver is as good or even better than the best specific exact algorithms. VRPSolver is freely available for academic purposes, new applications can be developed by defining a suitable model in a C++ (Sadykov and Vanderbeck, 2021) or Julia/JuMP language interface[1]. At the time of writing, VRPSolver website lists 23 papers that used that tool on a variety of problems, including for example, multi-shuttle crane scheduling (Polten and Emde, 2022) or the differential harvesting problem (Volte et al., 2023).

However, VRPSolver is less used than it could be due to its intricate generic model. A VRPSolver model is a general MIP model plus the specification that some of its variables should be linear combinations of the incidence vectors of certain resource-constrained paths over user-defined graphs. To obtain a good performance, it requires the proper definition of "packing sets" over the arcs or nodes of the graphs. The use of VRPSolver is restricted to those who are acquainted with the MIP modeling technique. This is unfortunately not the case for most routing practitioners. Mastering the connection between the MIP part and graphs is even more demanding. Finally, full comprehension of non-standard concepts requires knowledge of how column generation and advanced BCP algorithms work.

This article introduces VRPSolverEasy, a tool that provides a Python language interface to VRPSolver which does not require any knowledge of MIP modeling or column generation. Instead, the problem is defined only in terms of elements that are familiar to routing practitioners: points (which can be customers or depots), links (possible paths between points), and vehicle types. VRPSolverEasy internally translates the problem into a VRPSolver model, which is then solved by the BCP algorithm (Pessoa et al., 2020).

VRPSolverEasy is a compromise between simplicity and generality, and it is restricted to a subset of VRP variants. It is designed to handle problems where the customers have demands (the sum of the demands in a route is limited by the vehicle capacity), time windows (limiting the time when service can happen), and possible penalties for not being served. The fleet of vehicles can be heterogeneous and there may be multiple depots. Certain vehicles may be incompatible with some customers. Routes can be closed (starting and ending at predefined depots) or open (starting and/or ending at any point). It is possible to define parallel links between the same pair of points, modeling alternative non-dominated paths, i.e., less costly paths that take more time to traverse. Finally, customers may have alternative locations and/or time windows. These features already cover the most popular VRP variants, including Capacitated VRP (CVRP), VRP with Time Windows (VRPTW), Heterogeneous Fleet VRP (HFVRP), Multi-Depot VRP (MDVRP), Open CVRP (OVRP), and Team Orienteering Problem (TOP). Moreover, it can handle arbitrary *combinations* of those variants. Overall, VRPSolverEasy is still far more generic than attempts preceding VRPSolver to provide generic exact methods, like Baldacci and Mingozzi (2009).

The outline of the paper is the following. In Section 2, we introduce the VRPSolverEasy model, and the package interface for its definition. Section 3 is optional to read, its understanding is not required to start using VRPSolverEasy. In it, we show how the VRPSolverEasy model can be reduced to the more generic VRPSolver model, introduced before in the literature. Computational results for the package are given in Section 4. Finally, some conclusions and perspectives are outlined in Section 5.

2 The VRPSolverEasy Model

VRPSolverEasy package has two versions. The free version uses open-source LP solver CLP. It can be obtained through the Python package installer with the command:

```
python3 -m pip install VRPSolverEasy
```

[1] https://vrpsolver.math.u-bordeaux.fr

The more performant academic version uses commercial LP/MIP solver CPLEX. It requires an e-mail address from an academic institution in order to download and then compile the generic BCP solver BaPCod (Sadykov and Vanderbeck, 2021) with the CPLEX support. Installation of this version is described in the package documentation.

The model is created using the commands

```
import VRPSolverEasy
model = VRPSolverEasy.Model()
```

VRPSolverEasy model is defined by four sets of entities: *depot points*, *customer points*, *links*, and *vehicle types*. A depot can be added by the command

```
model.add_depot(id=<id>, name='', service_time=0.0, tw_begin=0.0, tw_end=0.0)
```

The characterization of a depot point is presented in Table 1.

Field	Explanation	Type	Default	Notation
id	ID of the point	int ≥ 0	–	v
name	name	str	empty	–
service_time	service time for loading/unloading of the vehicle	float ≥ 0.0	0.0	s_v
tw_begin	start of the time window	float	0.0	b_v
tw_end	end of the time window	float	0.0	e_v

Table 1: Characterization of a depot point v.

Each customer is associated to one or several points. Each point, either depot or customer one, should have a unique point ID. A customer can be added by the command

```
model.add_customer(id=<id>, id_customer=<id>, name='', demand=0, penalty=0.0,
    service_time=0.0, tw_begin=0.0, tw_end=0.0, incompatible_vehicles=[])
```

The point ID (`id`) given may be different from the customer ID (`customer_id`). Additional points for a customer can be added using command

```
model.add_point(id=<id>, id_customer=<id>, name='', service_time=0.0, tw_begin
    =0.0, tw_end=0.0, incompatible_vehicles=[])
```

Additional points can be used to represent alternative time windows and/or alternative customer locations with possibly different set of compatible vehicles. Demand and penalty of a customer is the same across all its points. The characterization of a customer point is presented in Table 2.

Field	Explanation	Type	Default	Notation
id	ID of the point	int ≥ 1	–	v
id_customer	ID of the customer	int ≥ 1	id	$j(v)$
name	name	str	empty	–
demand	demand of the customer	int ≥ 0	0	$q_{j(v)}$
penalty	a positive penalty if the customer is optional, zero penalty if the customer is compulsory	float ≥ 0.0	0.0	$g_{j(v)}$
service_time	service time for the customer point	float ≥ 0.0	0.0	s_v
tw_begin	start of the time window	float	0.0	b_v
tw_end	end of the time window	float	0.0	e_v
incompatible_vehicles	IDs of incompatible vehicle types	list	[ ]	$I(v)$

Table 2: Characterization of a customer point v.

Characterization of a link is presented in Table 3. A link can be added by the command

```
model.add_link(start_point_id=<id>, end_point_id=<id>, name='', is_directed=False
    , distance=0.0, time=0.0, fixed_cost=0.0)
```

Field	Explanation	Type	Default	Notation
start_point_id	ID of the tail point	int ≥ 0	–	v_l^-
end_point_id	ID of the head point	int ≥ 0	–	v_l^+
name	name	str	empty	–
is_directed	whether can be followed in both directions	bool	False	–
distance	distance (or length) of the link	float ≥ 0.0	0.0	d_l
time	traversal time of the link	float ≥ 0.0	0.0	t_l
fixed_cost	fixed cost of the link	float ≥ 0.0	0.0	f_l

Table 3: Characterization of a link l.

A link represents either an arc (is_directed = True) or an edge (is_directed = False).

Characterization of a vehicle type is presented in Table 4. A vehicle type can be added by the command

```
model.add_vehicle_type(id=<id>, start_point_id=-1, end_point_id=-1, name='',
    capacity=0, fixed_cost= 0.0, var_cost_dist=0.0, var_cost_time=0.0, max_number
    =1, tw_begin=0.0, tw_end=0.0)
```

Field	Explanation	Type	Default	Notation
id	ID of the vehicle type	int ≥ 1	–	k
start_point_id	ID of the start depot point; -1 if a vehicle may start anywhere	int ≥ -1	-1	v_k^{start}
end_point_id	ID of the end depot point; -1 if a vehicle may end anywhere	int ≥ -1	-1	v_k^{end}
name	name	str	empty	–
capacity	capacity of a vehicle of this type	int ≥ 0	0	Q_k
max_number	number of available vehicles of this type	int ≥ 1	1000	U_k
fixed_cost	fixed cost of using one vehicle of this type	float ≥ 0.0	0.0	F_k
var_cost_dist	unitary cost for the distance traveled by a vehicle	float ≥ 0.0	0.0	C_k^{dist}
var_cost_time	unitary cost for the time spent by a vehicle (excludes waiting time)	float ≥ 0.0	0.0	C_k^{time}
tw_begin	start of the vehicle availability time window	float	0.0	B_k
tw_end	end of the vehicle availability time window	float	0.0	E_k

Table 4: Characterization of a vehicle type k.

Field	Explanation	Type	Notation
vehicle_type_id	ID of the vehicle type	int ≥ 1	$k(r)$
route_cost	cost of the route	float	$c(r)$
point_ids	ID of each visited points	list(int)	$v(r)$
point_names	name of each visited point	list(str)	–
incoming_arc_names	name of each followed arc	list(str)	–
cap_consumption	load of the vehicle after service at each visited point	list(int)	$q^{\text{cap}}(r)$
time_consumption	time of the end of the service for each visited point	list(float)	$q^{\text{time}}(r)$

Table 5: Characterization of a vehicle route r.

Finally, the model has the total maximum number of vehicles $\bar{U}$ (across all types). Its default value is 10000. It can be changed with command

```
model.set_max_total_vehicles_number(<value>)
```

Characterization of a vehicle route is presented in Table 5. The cost of the route is the sum of the costs of links it follows. The cost of a link is the sum of 1) the link fixed cost; 2) link time multiplied by the unitary time cost of the vehicle type; and 3) link distance multiplied by the unitary distance cost of the vehicle type. A vehicle route is valid if

- it starts and finishes at the start and end depot point of the vehicle type (if these points are defined);
- all visited points are compatible with the vehicle type;
- the sequence of points and links the route follows is valid;
- the total demand of visited points does not exceed the vehicle capacity;
- the service at a visited point does not start sooner than the end of the service of the previous visited point plus the traversal time of the link connecting them;
- at each visited point, the service interval is included in the time window.

A solution of the model is a set of routes. It is valid if

- all its routes are feasible;
- number of routes for each vehicle type does not exceed the number of available vehicles of this type;
- total number of routes does not exceed the maximum total vehicles number;
- all compulsory customers are visited exactly once.

The cost of a solution is the sum of 1) the total cost of its routes; 2) the total penalty of non-visited optional customers; and 3) the total fixed cost of used vehicles.

VRPSolverEasy searches for a minimum-cost feasible solution by translating the presented model to a more generic VRPSolver model (Pessoa et al., 2020) and then calling the generic state-of-the-art branch-cut-and-price solution approach (Sadykov et al., 2021). The solver can be parameterized using the command

```
model.set_parameters(time_limit=300, upper_bound=1000000, heuristic_used=False,
    time_limit_heuristic=20, solver_name='CLP', print_level=-1)
```

The list of available parameters is given in Table 6. The efficiency of the solver depends heavily on the cut-off value given by the user: the closer is this value to the optimal solution, the faster the solver will run. It is often advantageous to first run an external heuristic for the problem at hand, and then to give to VRPSolverEasy the solution value found by the heuristic as the cut-off value. To solve the given problem, VRPSolverEasy uses a BCP algorithm, which requires an external LP solver. The free version of VRPSolverEasy comes with an embedded open-source CLP solver. The academic version of VRPSolverEasy allows one to use commercial LP/MIP solver CPLEX instead. Using CPLEX improves performance of VRPSolverEasy, and allows one to use the MIP-based built-in heuristic. The built-in heuristic cannot be used with CLP solver.

Parameter	**Explanation**	**Type**	**Default**
time_limit	time limit for the solver (in seconds)	float > 0	300
upper_bound	cut-off value, i.e. only solutions with smaller value will be searched for	float	10^6
heuristic_used	switch on/off built-in heuristic	bool	False
time_limit_heuristic	time limit for a run of the built-in heuristic (in seconds)	float	20
solver_name	which underlying LP/MIP solver is used	str	"CLP"
print_level	verbosity of the solver (-2: no output; -1: reduced output; 0: normal output):	int $\in \{-2, -1, 0\}$	-1

Table 6: Parameterization of the solver.

The solver can be run using the command `model.solve()`. After execution, the solution status code is available in `model.status`. A negative code states en error. List of different errors is given

Code	Status	Explanation
0	OPTIMAL_SOL_FOUND	solver found a solution and proved its optimality
1	BETTER_SOL_FOUND	solver interrupted by the time limit, but found a solution with a cost smaller than the cut-off value
2	BETTER_SOL_DOES_NOT_EXISTS	solver proved that no solution exists with a cost smaller than the cut-off value
3	BETTER_SOL_NOT_FOUND	solver interrupted by the time limit and did not find a solution with a cost smaller than the cut-off values

Table 7: Solution status codes of the solver.

in the documentation of the solver. The list of solution status codes is presented in Table 7. If a solution is found (`model.solution.is_defined()==True`), its value is available in `model.solution.value`, and the routes forming the solution are available in the list `model.solution.routes`.

After execution, solver statistics listed in Table 8 are available in `model.statistics`.

Statistic	Explanation	Type
solution_time	solution time (in seconds)	float
best_lb	value of the global lower bound obtained	float
root_lb	value of the lower bound obtained at the root node	float
root_time	solution time of the root node	float
number_branch_and_bound_nodes	number of branch-and-bound nodes explored	int

Table 8: Statistics of the solver.

3 Translation to VRPSolver Model

In this section, we formalize how a VRPSolverEasy model is translated to a VRPSolver model (Pessoa et al., 2020). Let V, J, L, and K the sets of points, customers, links, and vehicle types.

3.1 Path Generator Graphs

For each vehicle type $k \in K$, we define a directed graph $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{A}^k)$. Set $\mathcal{V}^k$ includes the set of all customer points which are compatible with vehicle type k. Set $\mathcal{V}^k$ also includes starting and ending depot points v_k^{start} and v_k^{end} if they are defined. If at least one of such depot points is not defined, we include artificial point v_k^{art} to $\mathcal{V}^k$ and define $v_k^{\text{start}} = v_k^{\text{art}}$ and/or $v_k^{\text{end}} = v_k^{\text{art}}$.

For every link $l \in L$ such that both v_l^- and v_l^+ belong to $\mathcal{V}^k$, we include to $\mathcal{A}^k$ 1) arc (v_l^-, v_l^+), and 2) arc (v_l^+, v_l^-) if link l is undirected. Let $l(a)$ be the link corresponding to arc a. If $v_k^{\text{start}} = v_k^{\text{art}}$, in $\mathcal{A}^k$ we include artificial arcs (v_k^{art}, v) for all $\mathcal{V}^k \setminus \{v_k^{\text{end}}\}$. If $v_k^{\text{end}} = v_k^{\text{art}}$, in $\mathcal{A}^k$ we include artificial arcs (v, v_k^{art}) for all $\mathcal{V}^k \setminus \{v_k^{\text{start}}\}$. For any artificial arc a, we have $l(a) = \emptyset$, $d_\emptyset = 0$, $t_\emptyset = 0$, $f_\emptyset = 0$.

Finally, if $v_k^{\text{start}} = v_k^{\text{end}}$, we split the start and end vertices by introducing copy node v_k^{copy}, replace all arcs $a = (v, v_k^{\text{end}}) \in \mathcal{A}^k$ by arcs $a = (v, v_k^{\text{copy}})$, and set v_k^{end} to v_k^{copy}. We also set $j(v_k^{\text{start}}) = j(v_k^{\text{end}}) = \emptyset$, and $q_\emptyset = 0$.

The cost c_a^k of an arc $a \in \mathcal{A}^{\|}$ is equal to $f_{l(a)} + C_k^{\text{dist}} d_{l(a)} + C_k^{\text{time}} t_{l(a)}$. To take into account the fixed vehicle cost, c_a^k is increased by $\frac{1}{2}F_k$ for every arc a incident to v^{start} or v^{end}.

If at least one customer $j \in J$ has a positive demand $q_j > 0$, we introduce the capacity resource. Capacity resource consumption q_a^{cap} of every arc $a = (v^-, v^+) \in \mathcal{A}^k$ is equal to $\frac{1}{2}q_{j(v^-)} + \frac{1}{2}q_{j(v^+)}$. If at least one point or vehicle type has a positive service time, the traversal time of a link is positive, or the time window of a point or a vehicle type is different from $[0, 0]$, we introduce the time resource. Time resource consumption q_a^{time} of every arc $a = (v^-, v^+) \in \mathcal{A}^k$ is equal to $s_{v^-} + t_{l(a)}$.

If all links $l \in L$ are undirected and if the time windows of all points and all vehicle types are the same, then the forward-backward route symmetry can be exploited by the solver. In this case, we set $q_a^{\text{time}} = \frac{1}{2}s_{v^-} + t_{l(a)} + \frac{1}{2}s_{v^+}$.

A path P of length $n(P)$ in graph $\mathcal{G}^{\|}$ is defined by a sequence of vertices and arcs $\left(v_0^P, a_1^P, v_1^P, \ldots, a_{n(P)-1}^P, v_{n(P)-1}^p, a\right.$ such that for every arc a_i^P, v_{i-1}^P is its tail and v_i^P is its head, for all $1 \le i \le n(P)$. A path P in $\mathcal{G}^{\|}$ is feasible if and only if

- $v_0^P = v_k^{\text{start}}$, $v_{n(P)}^P = v_k^{\text{end}}$;
- the capacity resource is not defined or $\sum_{i=1}^{n(P)} q_{a_i^P}^{\text{cap}} \le Q_k$;
- the time resource is not defined or $t_i^P \le e_{v_i^P}$ for $1 \le i \le n(P)$, where $t_0^P = \max\left\{b_{v_0^P}, B_k\right\}$, $t_i^P = \max\left\{t_{i-1}^P + q_a^{\text{time}}, b_{v_i^P}\right\}$, and $t_{n(P)}^P \le E_k$.

By definition, every feasible path in graph $\mathcal{G}^{\|}$ corresponds to a feasible route for a vehicle of type k. Let $\mathcal{P}^{\|}$ be the set of all feasible paths in graph $\mathcal{G}^{\|}$. For each arc $a \in \mathcal{A}^{\|}$ and path $P \in \mathcal{P}^{\|}$, let h_a^P indicate how many times the arc a appears in path P.

3.2 Integer Programming Formulation

For every vehicle type $k \in K$ and every path $P \in \mathcal{P}^{\|}$, we define a binary variable λ_P which is equal to one if and only if path P participates in the solution. For every vehicle type and every arc $a \in \mathcal{A}^{\|}$, we also define an integer variable x_a^k stating how many times arc a is followed in the solution. For every customer $j \in J$, we define a binary variable y_j which is equal to one if and only if customer j is served. Our IP formulation is the following.

$$\text{Min} \quad \sum_{k \in K} \sum_{a \in \mathcal{A}^{\|}} c_a^k x_a^k + \sum_{j \in J} g_j(1 - y_j) \tag{1a}$$

$$\text{S.t.} \quad \sum_{k \in K} \sum_{\substack{a \in \delta(v): \\ v \in \mathcal{V}^{\|},\, \updownarrow(\sqsubseteq) = |}} \frac{1}{2} x_a^k = y_j, \qquad j \in J \tag{1b}$$

$$y_j = 1, \qquad j \in J:\ g_j = 0, \tag{1c}$$

$$x_a^k = \sum_{p \in \mathcal{P}^{\|}} \left(\sum_{a \in \mathcal{A}^{\|}} h_a^P \right) \lambda_P, \qquad k \in K, a \in \mathcal{A}^{\|}, \tag{1d}$$

$$\sum_{P \in \mathcal{P}^{\|}} \lambda_P \le U_k, \qquad k \in K, \tag{1e}$$

$$\sum_{k \in K} \sum_{P \in \mathcal{P}^{\|}} \lambda_P \le \bar{U}, \tag{1f}$$

$$\lambda_P \in \{0, 1\}, \qquad k \in K, P \in \mathcal{P}^{\|} \tag{1g}$$

$$x_a^k \in \mathbb{Z}_+, \qquad k \in K, a \in \mathcal{A}^{\|}, \tag{1h}$$

$$y_j \in \{0, 1\}, \qquad j \in J. \tag{1i}$$

The objective function (1a) minimizes the solution cost, which is the sum of the cost of the used arcs (which includes the fixed vehicle cost) and the total penalty for non-served optional customers. In the case customer $j \in J$ is served, i.e., $y_j = 1$, constraints (1d) state that two arcs in the solution should be incident to a vertex corresponding to this customer. Constraints (1c) fix variables y to one for compulsory clients. Constraints (1d) express variables x as a linear combination of path variables λ. These linear expressions are completely defined by mapping h between variables x and arcs in graphs. For this reason, x are called *mapped* variables. They enable us to formulate the objective function (1a), constraints (1b), and branching expressions

(see below) in a convenient and intuitive way. Constraints (1e) and (1f) guarantee that the number of used vehicles is smaller than the number of available ones.

Formulation (1), which includes graphs, resources, and the mapping, defines a special case of the generic model introduced in (Pessoa et al., 2020). Thus, VRPSolverEasy model can be solved by the BCP algorithm presented by Pessoa et al. (2020). In this algorithm, the linear relaxation of (1) is solved by the column generation approach after the elimination of mapped variables x. To dynamically generate path variables, the pricing subproblem is solved for each vehicle type. In the subproblem for vehicle type $k \in K$, one searches for a feasible resource-constrained path $P \in \mathcal{P}^{\Vert}$ in graph $\mathcal{G}^{\Vert}$ with the minimum reduced cost. For this purpose, the bucket graph based labeling algorithm proposed by Sadykov et al. (2021) is employed.

The branching is performed on variables y and on integer linear expressions of arc variables:

- number of used vehicles of type $k \in K$: $z_k^{\text{veh}} = \frac{1}{2} \sum_{a \in \delta(v_k^{\text{start}})} x_a^k + \frac{1}{2} \sum_{a \in \delta(v_k^{\text{end}})} x_a^k$;
- assignment of customer $j \in J$ to vehicle type $k \in K$: $z_{kj}^{\text{assign}} = \sum_{v \in \mathcal{V}^{\Vert}:\, |(\sqsubseteq)=|} \sum_{a \in \delta(v)} \frac{1}{2} x_a^k$;
- usage of link $l \in L$: $z_l^{\text{link}} = \sum_{k \in K} \sum_{a \in \mathcal{A}^{\Vert}:\, \updownarrow(\dashv)=\updownarrow} x_a^k$.

3.3 Packing sets and advanced BCP components

The generic model proposed in (Pessoa et al., 2020) also includes the definition of so-called *packing sets*, which serve to enhance the solution of the model by transitioning from a basic Branch-and-Price algorithm to an advanced Branch-Cut-and-Price algorithm. Each packing set is formed by a subset W of vertices in $\mathcal{V} = \bigcup_{\Vert \in \mathcal{K}} \mathcal{V}^{\Vert}$, such that at most one vertex in W is visited at most once in any optimal solution, and other vertices are not visited in this solution. Naturally, for the VRPSolverEasy model, we can define one packing set W_j for every customer $j \in J$: $W_j = \bigcup_{k \in K} \left\{ v \in \mathcal{V}^{\Vert} :\, |(\sqsubseteq) = | \right\}$, as every customer cannot be visited more than once in any solution. The solver also requires the definition of a distance matrix between packing sets for each graph $\mathcal{G}^{\Vert}$. We define the distance between packing sets W_j and $W_{j'}$ in graph $\mathcal{G}^{\Vert}$ as the average cost of arcs from a vertex $v \in W_j$ to a vertex $v' \in W_{j'}$. The distance is set to infinity if there are no such arcs.

Pessoa et al. (2020) describe how algorithmic components used in modern BCPs for vehicle routing, such as *ng*-paths, route enumeration, and rank-1 cuts with limited memory, have been adapted to work with packing sets. As a result, the solution process of the model is significantly more efficient.

To further improve solution efficiency, we use the rounded capacity cuts separator defined in (Pessoa et al., 2020). The following valid inequalities are separated, which impose a lower bound on the number of vehicles that visit a subset of customers with positive demand:

$$\sum_{k \in K} \sum_{\substack{a=(v^-,v^+) \in \mathcal{A}^{\Vert}: \\ j(v^-) \notin S,\, j(v^+) \in S}} x_a^k \geq \left\lceil \frac{\sum_{j \in S} d_j}{\max_{k \in K} Q_k} \right\rceil, \quad S \subseteq \{ j \in J :\, q_j > 0 \}.$$

3.4 Parameterization

We mostly use the default parameterization of the BCP algorithm described in (Pessoa et al., 2020). Changes to the default parameterization are automated. They depend on the *tentative average route size* ℓ. If the time resource is not defined, we set $\ell = (\max_{k \in K} Q_k) \,/\, \left(\sum_{j \in J} d_j / |J| \right)$, the maximum capacity of the available vehicles divided by the average customer demand. If the time resource is defined, ℓ is calculated in the following way. For each vehicle type $k \in K$, we randomly generate 21 paths in $\mathcal{G}^{\Vert}$ starting in v_k^{start}. Every path is generated iteratively. In every iteration, we define the set of resource-feasible extensions, which correspond to customer vertices

v such that the current partial path can be extended to v and then to v_k^{end}. If at least the half of potential extensions are infeasible due to the vehicle capacity, the generation of the current path is completed, and it is declared capacity-critical. If all potential extensions are infeasible, the generation is also completed, and the current path is declared time-critical. Otherwise, the next extension is chosen according to the geometric distribution with parameter 0.5, where extensions are sorted by their cost. We set ℓ to the average length of randomly generated paths. If $\ell > 12$, the solver is parameterized to use the limited arc memory for rank-one cuts (Pecin et al., 2017a). If $\ell \leq 12$, the limited node memory is used (Pecin et al., 2017b).

If both resources are defined, two following decisions are also made.

- First, we determine for every graph $\mathcal{G}^{\|}$, $k \in K$, which resource is more critical based on whether there are more time-critical or capacity-critical randomly generated paths. The critical resource is used to perform the bi-directional search in the labeling algorithm for the pricing problem.
- The second check is performed if the time resource is critical for all graphs, capacities are the same for all vehicle types, and there are no optional customers. In this case, for all randomly generated paths, we calculate the average ratio between the accumulated consumption of the capacity resource and the vehicle capacity. If this average ratio is smaller than 0.35, then capacity resource is removed from all graphs, and the vehicle capacity satisfaction is ensured by rounded capacity cuts, generated as lazy constraints.

A fixed randomization seed is always used to ensure that the solver parameterization does not change when the same instance is solved multiple times. Advanced users may override any default or automated parameterization by setting `model.parameters.config_file` to the path to a VRPSolver configuration file.

4 Computational Experiments

In this section, we evaluate the performance of VRPSolverEasy for the classic CVRP, VRPTW, and HFVRP variants. We aim to give a idea to the reader about the solver efficiency. For this, experiments were carried out using the following well-known datasets:

- **CVRP instances.** The first dataset, *Small*, is formed by the instances of the classic sets A, B, E, F, M, and P with at most 100 customers. The second dataset, *Medium*, has the four instances from the classic sets (F-n135-k7, M-n121-k7, M-n151-k12, and M-n200-k16) plus the 22 instances from recent set X (Uchoa et al., 2017) having between 100 and 200 customers. All these instances can be found at the CVRPLIB[2] website.
- **VRPTW instances.** The first dataset, *Solomon*, is the classic benchmark proposed by Solomon (1987), which is composed only of instances with 100 customers. The second dataset, *HG-200*, are the Homberger and Gehring (1999) instances with 200 customers. These instances are also available at the CVRPLIB.
- **HFVRP instances.** The first dataset, *Classic*, also known as Golden–Taillard, has instances from 50 to 100 customers. The second dataset, *XH-200*, is a subset of the benchmark XH (based on the CVRP X instances) proposed by Pessoa et al. (2018) having between 100 and 200 customers. A detailed description of these instances can be found in Pessoa et al. (2018).

All the tests were done on i7-10700 CPUs running at 2.90GHz and with 16 GB of RAM. Each VRPSolverEasy run had a time limit of 1800 seconds on a single thread. We used CLP 1.17.7 as the default (and free) solver and CPLEX 20.1 as a commercial solver (free for academic

[2]http://vrp.atd-lab.inf.puc-rio.br

use). Both solvers were configured to run in single-thread mode. When using CPLEX, the built-in heuristic was enabled or disabled via the VRPSolverEasy parameter `heuristic_used`. We run OR-Tools[3] open-source solver for $|J|/2$ (the number of customers divided by two) seconds to hot start VRPSolverEasy with an initial primal bound. That time is included in the total running time. We remark that OR-Tools is far from being the best existing heuristic for each of those three standard variants. For example, HGS-CVRP (Vidal, 2022) can obtain much better solutions, but only for CVRP. OR-Tools was chosen due to its ease of use and generality. It can be used for all variants handled by VRPSolverEasy.

Table 9 compare the performance of VRPSolverEasy when using commercial (CPLEX) and free (CLP) solvers. The table reports the number of instances solved to proven optimality (**#opt**), the geometric mean of the total running times in seconds (**GMT**), the average gap between the initial upper bound and the final lower bound ($\mathbf{gap_I}$), and the average final gap ($\mathbf{gap_F}$). The difference between initial and final gaps show the average improvement by VRP-Solver of the feasible solution supplied by OR-Tools. We can give the following analysis from the obtained results.

- VRPSolverEasy is a very good choice for small instances with up to 100 customers. The vast majority of them can indeed be optimally solved, and small gaps are obtained for the remaining ones.
- VRPSolverEasy may still be tried for solving larger instances with up to 200 customers. Slightly less than the half of instances can be solved to optimality within the half-hour time limit. However, OR-Tools solutions are likely to be improved and reasonable final gaps may be obtained.

The performance is better when using the CPLEX solver and the built-in heuristic, especially for the HFVRP. The exception was on the VRPTW Solomon instances.

Table 9: VRPSolverEasy (with initial upper bound by OR-Tools) using CLP and CPLEX solvers.

Problem	**Dataset**	**CLP**				**CPLEX** (heuristic off)				**CPLEX** (heuristic on)			
		#opt	**GMT**	$\mathbf{gap_I}$	$\mathbf{gap_F}$	**#opt**	**GMT**	$\mathbf{gap_I}$	$\mathbf{gap_F}$	**#opt**	**GMT**	$\mathbf{gap_I}$	$\mathbf{gap_F}$
CVRP	Small	87/87	35.9	0.99	0.00	87/87	33.8	0.99	0.00	87/87	31.0	0.99	0.00
	Medium	11/26	914.2	4.45	2.64	14/26	844.9	4.36	2.35	16/26	617.3	4.36	1.96
VRPTW	Solomon	55/56	120.6	14.20	0.05	53/56	109.0	14.20	0.14	54/56	112.2	14.20	0.10
	HG-200	26/60	915.5	9.88	5.64	27/60	881.0	9.02	4.63	28/60	879.7	9.16	4.78
HFVRP	Classic	30/40	241.1	10.37	1.00	31/40	234.5	10.35	0.95	36/40	154.2	10.32	0.34
	XH-200	1/22	1733.8	18.06	13.88	2/22	1730.4	18.07	13.44	8/22	1088.6	18.65	8.25
Total		210/291				215/291				229/291			

To better assess the impact of the initial upper bound on VRPSolverEasy performance, we performed additional experiments on CVRP instances. In Table 10 on the left, we give results for runs without any external upper bound. For instances with up to 100 customers, the results were excellent, all 87 instances are solved, with a geometric mean time of 4.5 seconds. However, for instances with up to 200 customers, a very large final average gap of 49.2% is obtained. Although VRPSolverEasy could find a proven optimal solution for 11 instances, it could not find any feasible solution (100% gap) for most of the other instances. This shows that a robust approach using VRPSolverEasy, at least for instances with more than 100 customers, should include an external heuristic, even with an average performance like OR-Tools. In Table 10 on the right, we show a positive impact of using upper bounds equal or very close to the optimum provided by the state-of-the-art heuristic HGS-CVRP[4] (Vidal, 2022). We remark that such excellent heuristics are not available for the vast majority of the VRP variants covered by VRPSolverEasy.

[3]https://github.com/google/or-tools
[4]https://github.com/chkwon/PyHygese

Table 10: Impact of the initial upper bound on the VRPSolverEasy (using CLP solver) for CVRP.

Dataset	No upper bound			OR-Tools				HGS-CVRP			
	#opt	GMT	gap_F	#opt	GMT	gap_I	gap_F	#opt	GMT	gap_I	gap_F
Small	87/87	4.5	0.00	87/87	35.9	0.99	0.00	87/87	28.6	0.01	0.00
Medium	11/26	775.3	49.20	11/26	914.2	4.45	2.64	16/26	541.8	0.13	0.13

5 Conclusions

The paper introduces VRPSolverEasy, a Python package that can find provably optimal solutions for classic vehicle routing problems with state-of-the-art performance. The package is designed to be user-friendly and accessible to individuals who lack expertise in mathematical optimization. We believe that VRPSolverEasy will encourage greater usage of exact approaches in vehicle routing practice, where they have not been widely employed until now. Other potential uses of the package include the benchmarking of new heuristic solvers, by comparing the heuristic solutions with the optimum values or lower bounds obtained by VRPSolverEasy, possibly in long runs. It can also be used as a subroutine in decomposition-based heuristics like POPMUSIC (Queiroga et al., 2021), where a subset of routes having less than 100 customers in total in the current solution is optimally re-optimized.

VRPSolverEasy model can already handle several popular VRP variants. It can also be extended in the future to solve other variants that have already been effectively formulated and solved as VRPSolver models. Some examples of such variants include clustered and generalized VRPs (Freitas et al., 2023), arc routing problems (Pessoa et al., 2020), problems with backhauls (Queiroga et al., 2020), cumulative VRPs (Damião et al., 2023), multi-trip VRPs, VRPs with intermediate stops (Roboredo et al., 2022), and VRPs with capacitated depots having opening costs (Liguori et al., 2023). A particular extension will depend on the number of requests for it. Such requests can be done in the Issues section of the VRPSolverEasy repository[5].

Finally, we hope that our work will inspire the appearance of other easy-to-use software that provides efficient and generic exact or heuristic algorithms for vehicle routing and other related combinatorial optimization problems.

References

R. Baldacci, N. Christofides, and A. Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115(2):351–385, 2008.

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

Roberto Baldacci and Aristide Mingozzi. A unified exact method for solving different classes of vehicle routing problems. *Mathematical Programming*, 120:347–380, 2009.

Teobaldo Bulhoes, Artur Pessoa, Fábio Protti, and Eduardo Uchoa. On the complete set packing and set partitioning polytopes: Properties and rank 1 facets. *Operations Research Letters*, 46 (4):389–392, 2018.

C. Contardo and R. Martinelli. A new exact algorithm for the multi-depot vehicle routing problem under capacity and route length constraints. *Discrete Optimization*, 12:129–146, 2014.

[5]https://github.com/inria-UFF/VRPSolverEasy

Luciano Costa, Claudio Contardo, and Guy Desaulniers. Exact branch-price-and-cut algorithms for vehicle routing. *Transportation Science*, 53(4):946–985, 2019.

Caio Marinho Damião, João Marcos Pereira Silva, and Eduardo Uchoa. A branch-cut-and-price algorithm for the cumulative capacitated vehicle routing problem. *4OR*, 21:47–71, 2023.

Matheus Freitas, Jo ao Marcos Pereira Silva, and Eduardo Uchoa. A unified exact approach for clustered and generalized vehicle routing problems. *Computers & Operations Research*, 149: 106040, 2023.

Jörg Homberger and Hermann Gehring. Two evolutionary metaheuristics for the vehicle routing problem with time windows. *INFOR: Information Systems and Operational Research*, 37(3): 297–318, 1999. doi: 10.1080/03155986.1999.11732386.

M. Jepsen, B. Petersen, S. Spoorendonk, and D. Pisinger. Subset-row inequalities applied to the vehicle-routing problem with time windows. *Operations Research*, 56(2):497–511, 2008.

G. Laporte and Y. Nobert. A branch and bound algorithm for the capacitated vehicle routing problem. *OR Spectrum*, 5(2):77–85, 1983.

Pedro Henrique P. V. Liguori, A. Ridha Mahjoub, Guillaume Marques, Ruslan Sadykov, and Eduardo Uchoa. Non-robust strong knapsack cuts for capacitated location-routing and related problems. *Operations Research*, 2023. To appear.

Jens Lysgaard, Adam N Letchford, and Richard W Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100:423–445, 2004.

Diego Pecin, Claudio Contardo, Guy Desaulniers, and Eduardo Uchoa. New enhancements for the exact solution of the vehicle routing problem with time windows. *INFORMS Journal on Computing*, 29(3):489–502, 2017a.

Diego Pecin, Artur Pessoa, Marcus Poggi, and Eduardo Uchoa. Improved branch-cut-and-price for capacitated vehicle routing. *Mathematical Programming Computation*, 9(1):61–100, 2017b.

Diego Pecin, Artur Pessoa, Marcus Poggi, Eduardo Uchoa, and Haroldo Santos. Limited memory rank-1 cuts for vehicle routing problems. *Operations Research Letters*, 45(3):206–209, 2017c.

Artur Pessoa, Ruslan Sadykov, and Eduardo Uchoa. Enhanced branch-cut-and-price algorithm for heterogeneous fleet vehicle routing problems. *European Journal of Operational Research*, 270(2):530–543, 2018. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2018.04.009. URL `https://www.sciencedirect.com/science/article/pii/S0377221718303126`.

Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. A generic exact solver for vehicle routing and related problems. *Mathematical Programming*, 183:483–523, 2020.

Lukas Polten and Simon Emde. Multi-shuttle crane scheduling in automated storage and retrieval systems. *European Journal of Operational Research*, 302(3):892–908, 2022.

Eduardo Queiroga, Yuri Frota, Ruslan Sadykov, Anand Subramanian, Eduardo Uchoa, and Thibaut Vidal. On the exact solution of vehicle routing problems with backhauls. *European Journal of Operational Research*, 287(1):76–89, 2020.

Eduardo Queiroga, Ruslan Sadykov, and Eduardo Uchoa. A POPMUSIC matheuristic for the capacitated vehicle routing problem. *Computers & Operations Research*, 136:105475, 2021.

Marcos Roboredo, Ruslan Sadykov, and Eduardo Uchoa. Solving vehicle routing problems with intermediate stops using vrpsolver models. *Networks*, n/a(n/a), 2022.

S. Røpke. Branching decisions in branch-and-cut-and-price algorithms for vehicle routing problems. *Presentation in Column Generation 2012*, 2012.

Ruslan Sadykov and François Vanderbeck. BaPCod — a generic Branch-And-Price Code. Technical report HAL-03340548, Inria Bordeaux — Sud-Ouest, September 2021.

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

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

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2016.08.012. URL `https://www.sciencedirect.com/science/article/pii/S0377221716306270`.

Thibaut Vidal. Hybrid genetic search for the CVRP: Open-source implementation and SWAP* neighborhood. *Computers & Operations Research*, 140:105643, 2022.

Gabriel Volte, Eric Bourreau, Rodolphe Giroudeau, and Olivier Naud. Using VRPSolver to efficiently solve the differential harvest problem. *Computers & Operations Research*, 149: 106029, 2023.