# Package: vrpr (via r-universe)

July 5, 2026

**Title** Vehicle Routing Problem Solver Built on 'PyVRP'

**Version** 0.1.0

**Description** A 'tidyverse'-style interface to high-performance vehicle routing problem (VRP) solving. Vendors the C++ core of the 'PyVRP' solver (<https://github.com/PyVRP/PyVRP>) and rewires it through 'cpp11', with no 'Python' runtime dependency. Supports the capacitated VRP, time windows, multiple depots, heterogeneous fleets, prize-collecting and multi-trip variants, driven by an iterated local search metaheuristic.

**License** MIT + file LICENSE

**URL** https://github.com/StrategicProjects/vrpr, https://strategicprojects.github.io/vrpr/

**BugReports** https://github.com/StrategicProjects/vrpr/issues

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**Depends** R (>= 4.3)

**Imports** cli, rlang, stats, tibble, vctrs

**LinkingTo** cpp11

**Suggests** ggplot2, knitr, reticulate, rmarkdown, testthat (>= 3.0.0), withr

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**SystemRequirements** C++20, GNU make

**Config/roxygen2/version** 8.0.0

**Config/pak/sysreqs** make

**Repository** https://strategicprojects.r-universe.dev

**Date/Publication** 2026-07-04 12:18:16 UTC

**RemoteUrl** https://github.com/strategicprojects/vrpr

**RemoteRef** HEAD

**RemoteSha** d4c3f099a16e6a4e261fd65f7e6cec9cff65cc31

# Contents



---

`add_client_group` *Add a mutually exclusive group of clients*

---

## Description

Defines a group from which at most one client is visited (or exactly one if `required = TRUE`). Useful for prize-collecting with exclusive alternatives (e.g. serving one of several equivalent points). Clients in the group automatically become optional (individual `required = FALSE`); use `prize` to encourage a visit.

## Usage

```
add_client_group(model, clients, required = FALSE)
```

## Arguments

| | |
|---|---|
| `model` | A `vrpr_model`. |
| `clients` | Vector of client numbers (1-based, in the order of `add_clients()`) that form the group. |
| `required` | If `TRUE`, exactly one client in the group must be visited; if `FALSE` (default), at most one. |

### Value

The updated `vrpr_model`.

---

`add_clients` *Add clients to the model*

---

### Description

Add clients to the model

### Usage

```
add_clients(model, data)
```

### Arguments

| | |
|---|---|
| `model` | A `vrpr_model`. |
| `data` | A tibble/data.frame with at least the columns `x` and `y`. Optional columns: `demand` (delivery), `pickup`, `tw_early`, `tw_late`, `service`, `release_time`, `prize`, `required`. The time-window columns (`tw_early`/`tw_late`/`service`) enable the VRPTW; `pickup` enables simultaneous pickup and delivery / backhaul. |

### Value

The updated `vrpr_model`.

---

`add_depot` *Add a depot to the model*

---

### Description

Add a depot to the model

### Usage

```
add_depot(model, x, y, tw_early = 0, tw_late = Inf, service = 0)
```

### Arguments

| | |
|---|---|
| `model` | A `vrpr_model`. |
| `x, y` | Depot coordinates. |
| `tw_early, tw_late` | Depot time window (opening/closing). `tw_late = Inf` leaves the closing time unconstrained. |
| `service` | Service time at the depot (e.g. loading), per trip. |

## Value

The updated `vrpr_model`.

---

`add_vehicle_type` *Add a vehicle type to the model*

---

## Description

Add a vehicle type to the model

## Usage

```
add_vehicle_type(
  model,
  num_available,
  capacity,
  fixed_cost = 0,
  tw_early = 0,
  tw_late = Inf,
  max_duration = Inf,
  unit_distance_cost = 1,
  unit_duration_cost = 0,
  depot = 1L,
  start_depot = depot,
  end_depot = depot,
  reload_depots = integer(0),
  max_reloads = Inf
)
```

## Arguments

| | |
|---|---|
| `model` | A `vrpr_model`. |
| `num_available` | Number of vehicles available of this type. |
| `capacity` | Vehicle capacity. |
| `fixed_cost` | Fixed cost per vehicle used. |
| `tw_early`, `tw_late` | Vehicle shift time window (start/end). `tw_late` = `Inf` leaves the end of the shift unconstrained. |
| `max_duration` | Maximum route duration. `Inf` = unconstrained. |
| `unit_distance_cost`, `unit_duration_cost` | Variable cost per unit of distance and of duration for this type. Varying these (and `capacity`, `fixed_cost`) across calls enables a **heterogeneous fleet**. |
| `depot` | Index (1-based) of the depot vehicles of this type start from and return to. Shortcut to set `start_depot` and `end_depot` together. |

`start_depot`, `end_depot`
: Indices (1-based) of the start and end depots, in the order of `add_depot()`. Varying them across types enables the **MDVRP** (multiple depots).

`reload_depots`
: Indices (1-based) of depots where vehicles of this type may reload/empty mid-route, enabling **multi-trip** routes. Empty (default) = no reloading.

`max_reloads`
: Maximum number of reloads per route. `Inf` = unconstrained.

## Details

Call `add_vehicle_type()` several times for a fleet with multiple vehicle types (different capacities, costs, shifts or depots).

## Value

The updated `vrpr_model`.

---

`cost` *Cost of a result or solution*

---

## Description

Cost of a result or solution

## Usage

```
cost(x, ...)
```

## Arguments

`x`
: A `vrp_solve()` result or a `vrp_solution()`.

`...`
: Unused.

## Value

The objective cost (a `numeric` scalar); `Inf` if no feasible solution was found.

`ils_params` *ILS solver parameters*

## Description

ILS solver parameters

## Usage

```
ils_params(
  num_neighbours = 20L,
  min_perturbations = 1L,
  max_perturbations = 25L,
  init_load = 20,
  init_tw = 6,
  init_dist = 6,
  history_length = 300L,
  num_iters_no_improvement = 150000L,
  exhaustive_on_best = TRUE
)
```

## Arguments

| | |
|---|---|
| `num_neighbours` | Granular neighbourhood size (k neighbours per client). |
| `min_perturbations`, `max_perturbations` | Range of perturbations per iteration. |
| `init_load`, `init_tw`, `init_dist` | Initial penalties. |
| `history_length` | Length of the late-acceptance history (> 0). Default 300, as in PyVRP. |
| `num_iters_no_improvement` | Iterations without improvement before restarting from the best. |
| `exhaustive_on_best` | Refine each new best with an exhaustive search? |

## Value

A list of parameters.

## plot.vrpr_model *Plot a VRP model (depots and clients only)*

### Description

Plot a VRP model (depots and clients only)

### Usage

```
## S3 method for class 'vrpr_model'
plot(x, ...)
```

### Arguments

| | |
|---|---|
| x | A vrp_model(). |
| ... | Unused. |

### Value

A ggplot object.

## plot.vrpr_result *Plot the solution of a VRP result*

### Description

Draws depots, clients and the routes (one colour per route) over the instance coordinates. Unvisited optional clients (prize-collecting) appear as hollow circles.

### Usage

```
## S3 method for class 'vrpr_result'
plot(x, show_clients = TRUE, ...)
```

### Arguments

| | |
|---|---|
| x | A vrp_solve() result. |
| show_clients | Reserved; clients are always drawn. |
| ... | Unused. |

### Value

A ggplot object.

## read_solomon *Read a VRPTW instance in Solomon format*

### Description

Reads VRPTW instances in Solomon (and Gehring-Homberger) format, with the `VEHICLE` section (number and capacity) and the `CUSTOMER` table (coordinates, demand, time window and service time). Customer 0 is the depot.

### Usage

```
read_solomon(path, num_vehicles = NULL)
```

### Arguments

| | |
|---|---|
| `path` | Path to the file. |
| `num_vehicles` | Number of vehicles; if `NULL`, uses the value from the file. |

### Value

A `vrp_model()` ready for `vrp_solve()`.

## read_vrplib *Read an instance in VRPLIB / TSPLIB format*

### Description

Reads CVRP (and VRPTW) instances in VRPLIB/CVRPLIB format (extended TSPLIB), such as the X set by Uchoa et al. Supports Euclidean coordinates (`EDGE_WEIGHT_TYPE : EUC_2D`); time-window and service-time sections are read when present.

### Usage

```
read_vrplib(path, num_vehicles = NULL)
```

### Arguments

| | |
|---|---|
| `path` | Path to the `.vrp` file. |
| `num_vehicles` | Number of available vehicles. If `NULL`, uses the `VEHICLES`/`TRUCKS` field, the `-k<n>` suffix in the name, or – as a last resort – the number of clients (always feasible). |

### Value

A `vrp_model()` ready for `vrp_solve()`.

## routes *Routes of a solution, in long (tidy) format*

### Description

Routes of a solution, in long (tidy) format

### Usage

```
routes(x, ...)
```

### Arguments

| | |
|---|---|
| x | A vrp_solution(). |
| ... | Unused. |

### Value

A tibble with one row per visit: route_id, depot (start depot, 1-based), position, client, vehicle_type, start_service (start of service) and wait (waiting time). The last two are only meaningful with time windows (VRPTW); depot varies in the MDVRP.

## solution_cost *Cost of a solution*

### Description

Cost of a solution

### Usage

```
solution_cost(solution, cost_evaluator = NULL)
```

### Arguments

| | |
|---|---|
| solution | A vrp_solution(). |
| cost_evaluator | A vrp_cost_evaluator(). If NULL, uses an evaluator with unit penalties. |

### Value

The penalised cost (a numeric scalar). For feasible solutions this is the objective cost; the feasible attribute reports feasibility.

## summary.vrpr_result *One-row summary of a result (tibble)*

### Description

One-row summary of a result (tibble)

### Usage

```
## S3 method for class 'vrpr_result'
summary(object, ...)
```

### Arguments

| | |
|---|---|
| object | A vrp_solve() result. |
| ... | Unused. |

### Value

A one-row tibble with cost, feasibility, number of routes, iterations and runtime.

## unvisited_clients *Unvisited optional clients*

### Description

In prize-collecting problems, clients with required = FALSE may be left out if the prize does not offset the routing cost.

### Usage

```
unvisited_clients(x, ...)
```

### Arguments

| | |
|---|---|
| x | A vrp_solve() result. |
| ... | Unused. |

### Value

An integer vector of the (1-based) client numbers not visited.

## vrp_cost_evaluator *Cost evaluator (CostEvaluator)*

### Description

Creates a penalised-cost evaluator. Penalties multiply constraint violations (load, time window, maximum distance) to form the smoothed cost the solver minimises. For a *feasible* solution, the penalised cost equals the objective cost.

### Usage

```
vrp_cost_evaluator(load_penalties = 1, tw_penalty = 1, dist_penalty = 1)
```

### Arguments

| | |
|---|---|
| load_penalties | Penalty per unit of excess load, per load dimension. A scalar is recycled across all dimensions. |
| tw_penalty | Penalty per unit of time warp (time-window violation). |
| dist_penalty | Penalty per unit of distance above the maximum. |

### Value

A vrpr_cost_evaluator object.

## vrp_model *Build a vehicle routing (VRP) model*

### Description

vrp_model() creates an empty model to which depots, clients and vehicle types are added via the pipe (|>). It is the tidy equivalent of PyVRP's Model class – the data boundary uses tibbles, not one object at a time.

### Usage

```
vrp_model()
```

### Value

A vrpr_model object.

## Examples

```
clients <- tibble::tibble(
  x = c(10, 25, 40), y = c(5, 30, 12),
  demand = c(10, 15, 8)
)
m <- vrp_model() |>
  add_depot(x = 0, y = 0) |>
  add_clients(clients) |>
  add_vehicle_type(num_available = 5, capacity = 100)
m
```

---

`vrp_problem_data` *Assemble the problem data (ProblemData) from a model*

---

## Description

Builds PyVRP's C++ `ProblemData` structure from a `vrp_model()`. Locations follow PyVRP's convention: depots first (low indices), then clients.

## Usage

```
vrp_problem_data(model, distance = NULL, duration = NULL)
```

## Arguments

| | |
|---|---|
| `model` | A `vrp_model()` with at least one depot and one vehicle type. |
| `distance`, `duration` | Matrices (`numeric`, `n x n`, locations in depots-then-clients order) of distance and duration. If `NULL`, they are computed as the rounded Euclidean distance between coordinates; `duration` defaults to `distance`. |

## Details

Integer measures (distance, duration, cost, load) travel as R `numeric` with integer semantics; non-integer values are rejected at the C++ boundary. Use `Inf` for "unconstrained" limits (e.g. `tw_late`).

## Value

A `vrpr_problem_data` object (a wrapper around a C++ external pointer).

## vrp_random_solution *Generate a random solution*

### Description

Generate a random solution

### Usage

```
vrp_random_solution(problem_data, seed = 42L)
```

### Arguments

| | |
|---|---|
| problem_data | A vrp_problem_data(). |
| seed | Integer seed. |

### Value

A vrpr_solution object.

## vrp_solution *Build a solution from explicit routes*

### Description

Build a solution from explicit routes

### Usage

```
vrp_solution(problem_data, routes)
```

### Arguments

| | |
|---|---|
| problem_data | A vrp_problem_data(). |
| routes | A list of integer vectors; each vector is a route given as *client numbers* (1..n_clients), in visit order. All routes use the first vehicle type. |

### Value

A vrpr_solution object.

`vrp_solve` *Solve a VRP model*

## Description

Runs the iterated local search (ILS) solver on a model, using PyVRP's vendored C++ core.

## Usage

```
vrp_solve(model, stop, seed = 42L, params = ils_params(), display = TRUE)
```

## Arguments

| | |
|---|---|
| `model` | A `vrp_model()` or an already-assembled `vrp_problem_data()`. |
| `stop` | A stopping criterion (see vrpr_stop), e.g. `max_runtime(10)`. |
| `seed` | Integer seed for reproducibility. |
| `params` | Solver parameters (see `ils_params()`). |
| `display` | Show progress via `{cli}`? |

## Value

A `vrpr_result` object with the best solution, cost, routes and run statistics. Use `cost()`, `routes()` and `summary()` to inspect it.

## Examples

```
clients <- tibble::tibble(
  x = c(10, 25, 40, 15), y = c(5, 30, 12, 22),
  demand = c(10, 15, 8, 12)
)
res <- vrp_model() |>
  add_depot(x = 0, y = 0) |>
  add_clients(clients) |>
  add_vehicle_type(num_available = 3, capacity = 50) |>
  vrp_solve(stop = max_iterations(200), display = FALSE)

cost(res)
routes(res)
```

vrpr_stop *Solver stopping criteria*

## Description

Control when the iterated local search loop should terminate. Each function returns a callable object (closure) that the solver invokes every iteration, receiving the cost of the current best solution and returning TRUE to stop.

## Usage

```
max_runtime(seconds)

max_iterations(max_iters)

no_improvement(n)

first_feasible()
```

## Arguments

| | |
|---|---|
| seconds | Maximum run time, in seconds. |
| max_iters | Maximum number of iterations. |
| n | Number of consecutive iterations without improvement before stopping. |

## Details

These are the R equivalent of PyVRP's pyvrp.stop module.

## Value

An object of class vrpr_stop: a function function(best_cost, feasible) returning TRUE/FALSE.

# Index