VRPy

Release 0.1.0

Romain Montagné, David Torres Sanchez

Sep 18, 2021

USER GUIDE

1 Disclaimer **3**

2 Authors **5**

3 Contributors **7**

4 Table of contents **9**

4.1 Getting started . . . 9
4.2 Using *VRPy* . . . 9
4.3 Vehicle Routing Problems . . . 10
4.4 Solving Options . . . 15
4.5 Examples . . . 17
4.6 API . . . 33
4.7 Mathematical Background . . . 35
4.8 Performance profiles . . . 37
4.9 Bibliography . . . 38

Bibliography **39**

Python Module Index **41**

Index **43**

VRPy is a python framework for solving instances of different types of Vehicle Routing Problems (VRP) including:

- the Capacitated VRP (CVRP),
- the CVRP with resource constraints,
- the CVRP with time windows (CVRPTW),
- the CVRP with simultaneous distribution and collection (CVRPSDC),
- the CVRP with heterogeneous fleet (HFCVRP).

Check out section *Vehicle Routing Problems* to find more variants and options.

VRPy relies on the well known NetworkX package (graph manipulation), as well as on cspy, a library for solving the resource constrained shortest path problem.

CHAPTER

ONE

DISCLAIMER

There is no guarantee that VRPy returns the optimal solution. See section *Mathematical Background* for more details, and section *Performance profiles* for performance comparisons with OR-Tools.

CHAPTER

TWO

AUTHORS

Romain Montagné (r.montagne@hotmail.fr)

David Torres Sanchez (d.torressanchez@lancs.ac.uk)

CHAPTER

THREE

CONTRIBUTORS

@Halvaros

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 Getting started

4.1.1 Installation

You can install the latest release of VRPy from PyPi by:

```
pip install vrpy
```

4.1.2 Requirements

The requirements for running VRPy are:

- cspy: Constrained shortest path problem algorithms [TS19].
- NetworkX: Graph manipulation and creation [HSSC08].
- numpy: Array manipulation [Oli06].
- PuLP: Linear programming modeler.

4.2 Using *VRPy*

In order to use the VRPy package, first, one has to create a directed graph which represents the underlying network.

To do so, we make use of the well-known *NetworkX* package, with the following input requirements:

- Input graphs must be of type `networkx.DiGraph`;
- Input graphs must have a single *Source* and *Sink* nodes with no incoming or outgoing edges respectively;
- There must be at least one path from *Source* to *Sink*;
- Edges in the input graph must have a `cost` attribute (of type `float`).

For example the following simple network fulfills the requirements listed above:

```
>>> from networkx import DiGraph
>>> G = DiGraph()
>>> G.add_edge("Source", 1, cost=1)
>>> G.add_edge("Source", 2, cost=2)
>>> G.add_edge(1, "Sink", cost=0)
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge(2, "Sink", cost=2)
>>> G.add_edge(1, 2, cost=1)
>>> G.add_edge(2, 1, cost=1)
```

The customer demands are set as `demand` attributes (of type `float`) on each node:

```
>>> G.nodes[1]["demand"] = 5
>>> G.nodes[2]["demand"] = 4
```

To solve your routing problem, create a `VehicleRoutingProblem` instance, specify the problem constraints (e.g., the `load_capacity` of each truck), and call `solve`.

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G, load_capacity=10)
>>> prob.solve()
```

Once the problem is solved, we can query useful attributes as:

```
>>> prob.best_value
3
>>> prob.best_routes
{1: ["Source", 2, 1, "Sink"]}
>>> prob.best_routes_load
{1: 9}
```

`prob.best_value` is the overall cost of the solution, `prob.best_routes` is a *dict* object where keys represent the route ID, while the values are the corresponding path from *Source* to *Sink*. And `prob.best_routes` is a *dict* object where the same keys point to the accumulated load on the vehicle.

Different options and constraints are detailed in the *Vehicle Routing Problems* section, and other attributes can be queried depending on the nature of the VRP (see section *API*).

4.3 Vehicle Routing Problems

The *VRPy* package can solve the following VRP variants.

4.3.1 Capacitated Vehicle Routing Problem (CVRP)

In the capacitated vehicle routing problem (CVRP), a fleet of vehicles with uniform capacity must serve customers with known demand for a single commodity. The vehicles start and end their routes at a common depot and each customer must be served by exactly one vehicle. The objective is to assign a sequence of customers (a route) to each truck of the fleet, minimizing the total distance traveled, such that all customers are served and the total demand served by each truck does not exceed its capacity.

```
>>> from networkx import DiGraph
>>> from vrpy import VehicleRoutingProblem
>>> G = DiGraph()
>>> G.add_edge("Source", 1, cost=1)
>>> G.add_edge("Source", 2, cost=2)
>>> G.add_edge(1, "Sink", cost=0)
>>> G.add_edge(2, "Sink", cost=2)
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge(1, 2, cost=1)
>>> G.nodes[1]["demand"] = 2
>>> G.nodes[2]["demand"] = 3
>>> prob = VehicleRoutingProblem(G, load_capacity=10)
>>> prob.solve()
```

Note that whether the problem is a distribution or a collection problem does not matter. Both are modeled identically.

4.3.2 CVRP with resource constraints

Other resources can also be considered:

- maximum duration per trip;
- maximum number of customers per trip.

Taking into account duration constraints requires setting `time` attributes on each edge, and setting the `duration` attribute to the maximum amount of time per vehicle.

Following the above example:

```
>>> G.edges["Source",1]["time"] = 5
>>> G.edges["Source",2]["time"] = 4
>>> G.edges[1,2]["time"] = 2
>>> G.edges[1,"Sink"]["time"] = 6
>>> G.edges[2,"Sink"]["time"] = 1
>>> prob.duration = 9
>>> prob.solve()
```

Similarly, imposing a maximum number of customers per trip is done by setting the `num_stops` attribute to the desired value.

```
>>> prob.num_stops = 1
```

4.3.3 CVRP with Time Windows (CVRPTW)

In this variant, deliveries must take place during a given time-window, which can be different for each customer.

Such constraints can be taken into account by setting `lower` and `upper` attributes on each node, and by activating the `time_windows` attribute to `True`. Additionally, service times can be taken into account on each node by setting the `service_time` attribute.

Following the above example:

```
>>> G.nodes[1]["lower"] = 0
>>> G.nodes[1]["upper"] = 10
>>> G.nodes[2]["lower"] = 5
>>> G.nodes[2]["upper"] = 9
>>> G.nodes[1]["service_time"] = 1
>>> G.nodes[2]["service_time"] = 2
>>> prob.time_windows = True
>>> prob.solve()
```

Note: Waiting time is allowed upon arrival at a node. This means that if a vehicle arrives at a node before the time window's lower bound, the configuration remains feasible, it is considered that the driver waits before servicing the customer.

4.3.4 CVRP with Simultaneous Distribution and Collection (CVRPSDC)

In this variant, when a customer is visited, two operations are done simultaneously. Some good is delivered, and some waste material is picked-up. The total load must not exceed the vehicle's capacity.

The amount that is picked-up is set with the `collect` attribute on each node, and the `distribution_collection` attribute is set to `True`.

Following the above example:

```
>>> G.nodes[1]["collect"] = 2
>>> G.nodes[2]["collect"] = 1
>>> prob.load_capacity = 2
>>> prob.distribution_collection = True
>>> prob.solve()
```

4.3.5 CVRP with Pickup and Deliveries (VRPPD)

In the pickup-and-delivery problem, each demand is made of a pickup node and a delivery node. Each pickup/delivery pair (or request) must be assigned to the same tour, and within this tour, the pickup node must be visited prior to the delivery node (as an item that is yet to be picked up cannot be delivered). The total load must not exceed the vehicle's capacity.

For every pickup node, the `request` attribute points to the name of the delivery node. Also, the `pickup_delivery` attribute is set to `True`. The amount of goods to be shipped is counted positively for the pickup node, and negatively for the delivery node. For example, if 3 units must be shipped from node 1 to node 2, the `demand` attribute is set to 3 for node 1, and -3 for node 2.

```
>>> G.nodes[1]["request"] = 2
>>> G.nodes[1]["demand"] = 3
>>> G.nodes[2]["demand"] = -3
>>> prob.pickup_delivery = True
>>> prob.load_capacity = 10
>>> prob.solve(cspy=False)
```

Note: This variant has to be solved with the `cspy` attribute set to False.

4.3.6 Periodic CVRP (PCVRP)

In the periodic CVRP, the planning period is extended over a time horizon, and customers can be serviced more than once. The demand is considered constant over time, and the frequencies (the number of visits) of each customer are known.

For each node, the `frequency` attribute (type `int`) is set, and the parameter `periodic` is set to the value of the considered time span (the planning period). All nodes that have no frequency are visited exactly once.

```
>>> G.nodes[1]["frequency"] = 2
>>> prob.periodic = 2
>>> prob.solve()
```

A planning of the routes can then be queried by calling `prob.schedule`, which returns a dictionary with keys day numbers and values the list of route numbers scheduled this day.

Note: The PCVRP usually has additional constraints: some customers can only be serviced on specific days of the considered time span. For example, over a 3 day planning period, a node with frequency 2 could only be visited on days 1 and 2 or 2 and 3 but not 1 and 3. Such *combination* constraints are not taken into account by *VRPy* (yet).

Note: If the parameter `num_vehicles` is used, it refers to the maximum number of vehicles available per day (and not over the time span).

4.3.7 CVRP with heterogeneous fleet (HFCVRP)

In the CVRP with *heterogeneous fleet* (or mixed fleet), there are different types of vehicles, which can differ in capacities and costs (fixed costs and travel costs). Typically, a vehicle with a larger capacity will be more expensive. The problem consists in finding the best combination of vehicles to satisfy the demands while minimizing global costs.

First, the `cost` attribute on each of the graph is now a *list* of costs, with as many items as vehicle types (even if costs are equal). For example, if there are *two* types of vehicles, the following graph satisfies the input requirements:

```
>>> from networkx import DiGraph
>>> G = DiGraph()
>>> G.add_edge("Source", 1, cost=[1, 2])
>>> G.add_edge("Source", 2, cost=[2, 4])
>>> G.add_edge(1, "Sink", cost=[0, 0])
>>> G.add_edge(2, "Sink", cost=[2, 4])
>>> G.add_edge(1, 2, cost=[1, 2])
```

When defining the `VehicleRoutingProblem`, the `mixed_fleet` argument is set to `True`, and the `load_capacity` argument is now also of type `list`, where each item of the list is the maximum load per vehicle type. For example, if the two types of vehicles have capacities 10 and 15, respectively:

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G, mixed_fleet=True, load_capacity=[10, 15])
```

Note how the dimensions of `load_capacity` and `cost` are consistent: each list must have as many items as vehicle types, and the order of the items of the `load_capacity` list is consistent with the order of the `cost` list on every edge of the graph.

Once the problem is solved, the type of vehicle per route can be queried with `prob.best_routes_type`.

4.3.8 VRP options

In this subsection are described different options which arise frequently in vehicle routing problems.

Open VRP

The *open* VRP refers to the case where vehicles can start and/or end their trip anywhere, instead of having to leave from the depot, and to return there after service. This is straightforward to model : setting distances (or costs) to 0 on every edge outgoing from the Source and incoming to the Sink achieves this.

Fixed costs

Vehicles typically have a *fixed cost* which is charged no matter what the traveled distance is. This can be taken into account with the `fixed_cost` attribute. For example, if the cost of using each vehicle is 100:

```
>>> prob.fixed_cost = 100
```

Note: If the fleet is mixed, the same logic holds for `fixed_cost`: a list of costs is given, where each item of the list is the fixed cost per vehicle type. The order of the items of the list has to be consistent with the other lists (`cost` and `load_capacity`). See example *Mixed fleet*.

Limited fleet

It is possible to limit the size of the fleet. For example, if at most 10 vehicles are available:

```
>>> prob.num_vehicles = 10
```

Note: If the fleet is mixed, the same logic holds for `num_vehicles`: a list of integers is given, where each item of the list is the number of available vehicles, per vehicle type. The order of the items of the list has to be consistent with the other lists (`cost`, `load_capacity`, `fixed_cost`).

And to enforce exactly `num_vehicles` vehicles, one can use the `use_all_vehicles`:

```
>>> prob.use_all_vehicles = True
```

Dropping visits

Having a limited fleet may result in an infeasible problem. For example, if the total demand at all locations exceeds the total capacity of the fleet, the problem has no feasible solution. It may then be interesting to decide which visits to drop in order to meet capacity constraints while servicing as many customers as possible. To do so, we set the `drop_penalty` attribute to an integer value that the solver will add to the total travel cost each time a node is dropped. For example, if the value of the penalty is 1000:

```
>>> prob.drop_penalty = 1000
```

This problem is sometimes referred to as the *capacitated profitable tour problem* or the *prize collecting tour problem.*

Minimizing the global time span

It is possible to modify the objective function in order to solve a min-max problem. More specifically, the total time span can be minimized by setting the `minimize_global_span` to `True`. Of course this assumes edges have a `time` argument:

```
>>> prob.minimize_global_span = True
```

Note: This may lead to poor computation times.

4.3.9 Other VRPs

Coming soon:

- CVRP with multiple depots

4.4 Solving Options

4.4.1 Setting initial routes for a search

By default, an initial solution is computed with the well known Clarke and Wright algorithm [CW64]. If one already has a feasible solution at hand, it is possible to use it as an initial solution for the search of a potential better configuration. The solution is passed to the solver as a list of routes, where a route is a list of nodes starting from the *Source* and ending at the *Sink*.

```
>>> prob.solve(initial_solution = [["Source",1,"Sink"],["Source",2,"Sink"]])
```

4.4.2 Returning solution from initial heuristic

It is possible to return the solution found by the Clarke and Wright algorithm by setting the `heuristic_only` argument to *True*.

```
>>> prob.solve(heuristic_only=True)
```

Note that this only possible with capacity and/or resource constraints.

4.4.3 Locking routes

It is possible to constrain the problem with partial routes if preassignments are known. There are two possibilites : either a complete route is known, and it should not be optimized, either only a partial route is known, and it may be extended. Such routes are given to the solver with the `preassignments` argument. A route with *Source* and *Sink* nodes is considered complete and is locked. Otherwise, the solver will extend it if it yields savings.

In the following example, one route must start with customer 1, one route must contain edge $(4, 5)$, and one complete route, *Source-2-3-Sink*, is locked.

```
>>> prob.solve(preassignments = [["Source",1],[4,5],["Source",2,3,"Sink"]])
```

4.4.4 Setting a time limit

The `time_limit` argument can be used to set a time limit, in seconds. The solver will return the best solution found after the time limit has elapsed.

For example, for a one minute time limit:

```
>>> prob.solve(time_limit=60)
```

4.4.5 Linear programming or dynamic programming

VRPy's `solve` method relies on a column generation procedure. At every iteration, a master problem and a sub problem are solved. The sub problem consists in finding variables which are likely to improve the master problem's objective function. See section *Mathematical Background* for more details.

The sub problem - or pricing problem - can be solved either with linear programming, or with dynamic programming. Switching to linear programming can be done by deactivating the `cspy` argument when calling the `solve` method. In this case the CBC [FRV+18] solver of COIN-OR is used by default.

```
>>> prob.solve(cspy=False)
```

The sub problems that are solved are typically computationally intractable, and using dynamic programming is typically quicker, as such algorithms run in pseudo-polynomial time. However, solving the sub problems as MIPs may also be effective depending on the data set. Also, using commercial solvers may significantly help accelerating the procedure. If one has CPLEX or GUROBI at hand, they can be used by setting the `solver` parameter to "cplex" or "gurobi".

```
>>> prob.solve(cspy=False, solver="gurobi")
```

4.4.6 Pricing strategy

In theory, at each iteration, the sub problem is solved optimally. VRPy does so with a bidirectional labeling algorithm with dynamic halfway point [TRothenbacherGI17] from the *cspy* library.

This may result in a slow convergence. To speed up the resolution, there are two ways to change this pricing strategy:

1. By deactivating the `exact` argument of the `solve` method, *cspy* calls one of its heuristics instead of the bidirectional search algorithm. The exact method is run only once the heuristic fails to find a column with negative reduced cost.

```
>>> prob.solve(exact=False)
```

2. By modifying the `pricing_strategy` argument of the `solve` method to one of the following:
    - *BestEdges1*,
    - *BestEdges2*,
    - *BestPaths*,
    - *Hyper*

```
>>> prob.solve(pricing_strategy="BestEdges1")
```

BestEdges1, described for example in [DellAmicoRS06], is a sparsification strategy: a subset of nodes and edges are removed to limit the search space. The subgraph is created as follows: all edges (i, j) which verify $c_{ij} > \alpha\ \pi_{max}$ are discarded, where c_{ij} is the edge's cost, $\alpha \in]0, 1[$ is parameter, and π_{max} is the largest dual value returned by the current

restricted relaxed master problem. The parameter α is increased iteratively until a route is found. *BestEdges2* is another sparsification strategy, described for example in [SPR18]. The β edges with highest reduced cost are discarded, where β is a parameter that is increased iteratively. As for *BestPaths*, the idea is to look for routes in the subgraph induced by the k shortest paths from the Source to the Sink (without any resource constraints), where k is a parameter that is increased iteratively.

Additionally, we have an experimental feature that uses Hyper-Heuristics for the dynamic selection of pricing strategies. The approach ranks the best pricing strategies as the algorithm is running and chooses according to selection functions based on [SZS15][FGonccalvesP17]. The selection criteria has been modified to include a combination of runtime, objective improvement, and currently active columns in the restricted master. Adaptive parameter settings found in [DOzcanB12] is used to balance exploration and exploitation under stagnation. The main advantage is that selection is done as the programme runs, and is therefore more flexible compared to a predefined pricing strategy.

For each of these heuristic pricing strategies, if a route with negative reduced cost is found, it is fed to the master problem. Otherwise, the sub problem is solved exactly.

The default pricing strategy is *BestEdges1*, with `exact=True` (i.e., with the bidirectional labeling algorithm).

4.4.7 A greedy randomized heuristic

For the CVRP, or the CVRP with resource constraints, one can activate the option of running a greedy randomized heuristic before pricing:

```
>>> prob.solve(greedy="True")
```

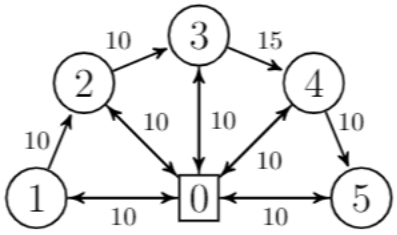
This algorithm, described in [SPR18], generates a path starting at the *Source* node and then randomly selects an edge among the γ outgoing edges of least reduced cost that do not close a cycle and that meet operational constraints (γ is a parameter). This is repeated until the *Sink* node is reached . The same procedure is applied backwards, starting from the *Sink* and ending at the *Source*, and is run 20 times. All paths with negative reduced cost are added to the pool of columns.

4.5 Examples

4.5.1 A simple example

Network definition

In this first example, we will be working with the following network:



The first step is to define the network as a `nx.Digraph` object. Note that for convenience, the depot (node 0 in the picture) is split into two vertices : the `Source` and the `Sink`.

```
# Create graph
>>> from networkx import DiGraph
>>> G = DiGraph()
>>> for v in [1, 2, 3, 4, 5]:
        G.add_edge("Source", v, cost=10)
        G.add_edge(v, "Sink", cost=10)
>>> G.add_edge(1, 2, cost=10)
>>> G.add_edge(2, 3, cost=10)
>>> G.add_edge(3, 4, cost=15)
>>> G.add_edge(4, 5, cost=10)
```

VRP definition

The second step is to define the VRP, with the above defined graph as input:

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G)
```

Maximum number of stops per route

In this first variant, it is required that a vehicle cannot perform more than 3 stops:

```
>>> prob.num_stops = 3
>>> prob.solve()
```

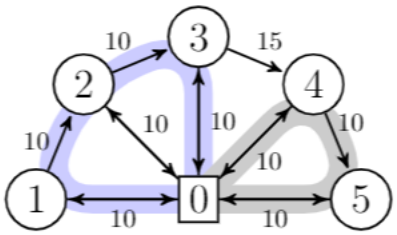
The best routes found can be queried as follows:

```
>>> prob.best_routes
{1: ['Source', 4, 5, 'Sink'], 2: ['Source', 1, 2, 3, 'Sink']}
```

And the cost of this solution is queried in a similar fashion:

```
>>> prob.best_value
70.0
>>> prob.best_routes_cost
{1: 30, 2: 40}
```

The optimal routes are displayed below:



Capacity constraints

In this second variant, we define a demand for each customer and limit the vehicle capacity to 10 units.

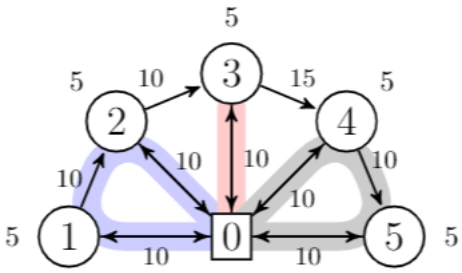
Demands are set directly as node attributes on the graph, and the capacity constraint is set with the `load_capacity` attribute:

```
>>> for v in G.nodes():
       if v not in ["Source", "Sink"]:
          G.nodes[v]["demand"] = 5
>>> prob.load_capacity = 10
>>> prob.solve()
>>> prob.best_value
80.0
```

As the problem is more constrained, it is not surprising that the total cost increases. As a sanity check, we can query the loads on each route to make sure capacity constraints are met:

```
>>> prob.best_routes
{1: ["Source", 1, "Sink"], 2: ["Source", 2, 3, "Sink"], 3: ["Source", 4, 5, "Sink"]}
>>> prob.best_routes_load
{1: 5, 2: 10, 3: 10}
```

The new optimal routes are displayed below:



Time constraints

One may want to restrict the total duration of a route. In this case, a *time* attribute is set on each edge of the graph, and a maximum duration is set with *prob.duration*.

```
>>> for (u, v) in G.edges():
       G.edges[u,v]["time"] = 20
>>> G.edges[4,5]["time"] = 25
>>> prob.duration = 60
>>> prob.solve()
>>> prob.best_value
85.0
```

As the problem is more and more constrained, the total cost continues to increase. Lets check the durations of each route:

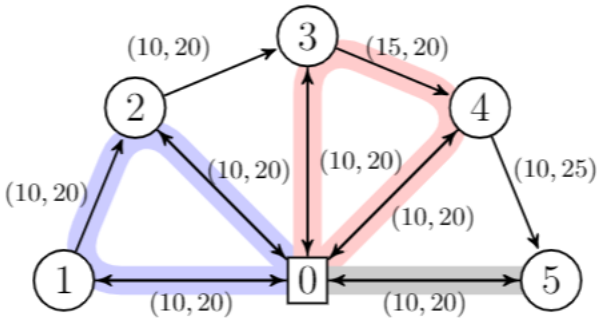
```
>>> prob.best_routes
{1: ["Source", 1, 2, "Sink"], 2: ["Source", 3, 4, "Sink"], 3: ["Source", 5, "Sink"]}
```

(continues on next page)

(continued from previous page)

```
>>> prob.best_routes_duration
{1: 60, 2: 60, 3: 40}
```

The new optimal routes are displayed below:



Time window constraints

When designing routes, it may be required that a customer is serviced in a given time window $[\ell, u]$. Such time windows are defined for each node, as well as service times.

```
>>> time_windows = {1: (5, 100), 2: (5, 20), 3: (5, 100), 4: (5, 100), 5: (5, 100)}
>>> for v in G.nodes():
        G.nodes[v]["lower"] = time_windows[v][0]
        G.nodes[v]["upper"] = time_windows[v][1]
        if v not in ["Source","Sink"]:
           G.nodes[v]["service_time"] = 1
```

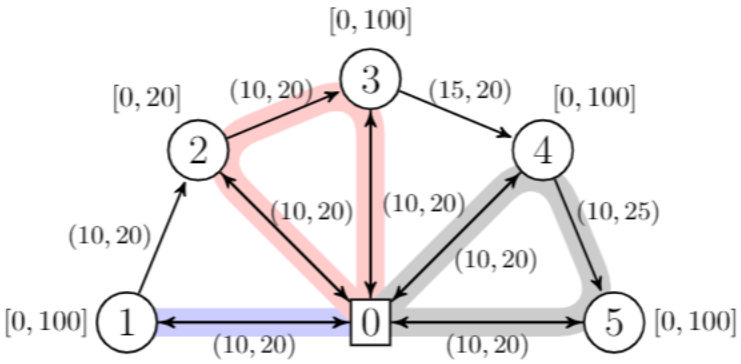
A boolean parameter `time_windows` is activated to enforce such constraints:

```
>>> prob.time_windows = True
>>> prob.duration = 64
>>> prob.solve()
>>> prob.best_value
90.0
```

The total cost increases again. Lets check the arrival times:

```
>>> prob.best_routes
{1: ["Source", 1, "Sink"], 4: ["Source", 2, 3, "Sink"], 2: ["Source", 4, "Sink"],  3: [
↪"Source", 5, "Sink"]}
>>> prob.arrival_time
{1: {1: 20, 'Sink': 41}, 2: {4: 20, 'Sink': 41}, 3: {5: 20, 'Sink': 41}, 4: {2: 20, 3:␣
↪41, 'Sink': 62}}
```

The new optimal routes are displayed below:



Complete program

```
import networkx as nx
from vrpy import VehicleRoutingProblem

# Create graph
G = nx.DiGraph()
for v in [1, 2, 3, 4, 5]:
       G.add_edge("Source", v, cost=10, time=20)
   G.add_edge(v, "Sink", cost=10, time=20)
   G.nodes[v]["demand"] = 5
   G.nodes[v]["upper"] = 100
   G.nodes[v]["lower"] = 5
   G.nodes[v]["service_time"] = 1
G.nodes[2]["upper"] = 20
G.nodes["Sink"]["upper"] = 110
G.nodes["Source"]["upper"] = 100
G.add_edge(1, 2, cost=10, time=20)
G.add_edge(2, 3, cost=10, time=20)
G.add_edge(3, 4, cost=15, time=20)
G.add_edge(4, 5, cost=10, time=25)

# Create vrp
prob = VehicleRoutingProblem(G, num_stops=3, load_capacity=10, duration=64, time_
↪windows=True)

# Solve and display solution
prob.solve()
print(prob.best_routes)
print(prob.best_value)
```

Periodic CVRP

For scheduling routes over a time period, one can define a frequency for each customer. For example, if over a planning period of two days, customer 2 must be visited twice, and the other customers only once:

```
>>> prob.periodic = 2
>>> G.nodes[2]["frequency"] = 2
>>> prob.solve()
>>> prob.best_routes
{1: ['Source', 1, 2, 'Sink'], 2: ['Source', 4, 5, 'Sink'], 3: ['Source', 2, 3, 'Sink']}
>>> prob.schedule
{0: [1, 2], 1: [3]}
```

We can see that customer 2 is visited on both days of the planning period (routes 1 and 3), and that it is not visited more than once per day.

Mixed fleet

We end this small example with an illustration of the `mixed_fleet` option, when vehicles of different types (capacities, travel costs, fixed costs) are operating.

The first vehicle has a `load_capacity` of 5 units, and no `fixed_cost`, while the second vehicle has a `load_capacity` of 20 units, and a `fixed_cost` with value 5. The travel costs of the second vehicle are 1 unit more expensive than those of the first vehicle:

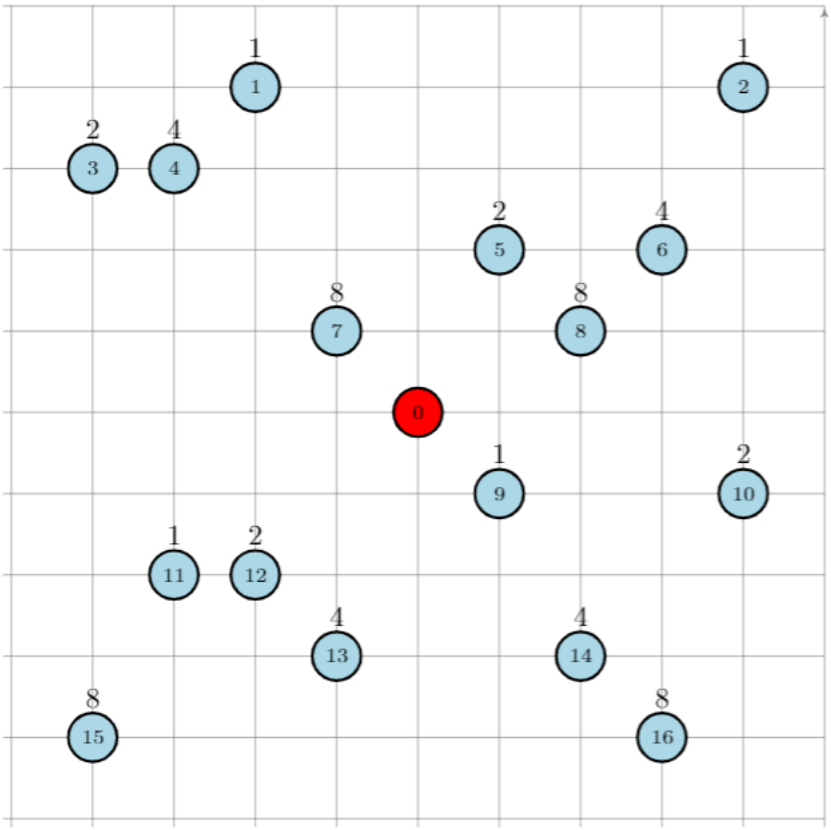
```
>>> from networkx import DiGraph
>>> from vrpy import VehicleRoutingProblem
>>> G = DiGraph()
>>> for v in [1, 2, 3, 4, 5]:
        G.add_edge("Source", v, cost=[10, 11])
        G.add_edge(v, "Sink", cost=[10, 11])
        G.nodes[v]["demand"] = 5
>>> G.add_edge(1, 2, cost=[10, 11])
>>> G.add_edge(2, 3, cost=[10, 11])
>>> G.add_edge(3, 4, cost=[15, 16])
>>> G.add_edge(4, 5, cost=[10, 11])
>>> prob=VehicleRoutingProblem(G, mixed_fleet=True, fixed_cost=[0, 5], load_capacity=[5,␣
↪20])
>>> prob.best_value
    85
>>> prob.best_routes
    {1: ['Source', 1, 'Sink'], 2: ['Source', 2, 3, 4, 5, 'Sink']}
>>> prob.best_routes_cost
    {1: 20, 2: 65}
>>> prob.best_routes_type
    {1: 0, 2: 1}
```

4.5.2 An example borrowed from *ortools*

We borrow this second example from the well known *ortools* [PF] routing library. We will use the data from the tutorial.

Network definition

The graph is considered complete, that is, there are edges between each pair of nodes, in both directions, and the cost on each edge is defined as the *Manhattan* distance between both endpoints. The network is displayed below (for readability, edges are not shown), with the depot in red, and the labels outside of the vertices are the demands:



The network can be entirely defined by its distance matrix. We will make use of the *NetworkX* module to create this graph and store its attributes:

```
from networkx import DiGraph, from_numpy_matrix, relabel_nodes, set_node_attributes
from numpy import array

# Distance matrix
DISTANCES = [
[0,548,776,696,582,274,502,194,308,194,536,502,388,354,468,776,662,0], # from Source
[0,0,684,308,194,502,730,354,696,742,1084,594,480,674,1016,868,1210,548],
[0,684,0,992,878,502,274,810,468,742,400,1278,1164,1130,788,1552,754,776],
[0,308,992,0,114,650,878,502,844,890,1232,514,628,822,1164,560,1358,696],
[0,194,878,114,0,536,764,388,730,776,1118,400,514,708,1050,674,1244,582],
[0,502,502,650,536,0,228,308,194,240,582,776,662,628,514,1050,708,274],
[0,730,274,878,764,228,0,536,194,468,354,1004,890,856,514,1278,480,502],
[0,354,810,502,388,308,536,0,342,388,730,468,354,320,662,742,856,194],
```

(continues on next page)

(continued from previous page)

```
[0,696,468,844,730,194,194,342,0,274,388,810,696,662,320,1084,514,308],
[0,742,742,890,776,240,468,388,274,0,342,536,422,388,274,810,468,194],
[0,1084,400,1232,1118,582,354,730,388,342,0,878,764,730,388,1152,354,536],
[0,594,1278,514,400,776,1004,468,810,536,878,0,114,308,650,274,844,502],
[0,480,1164,628,514,662,890,354,696,422,764,114,0,194,536,388,730,388],
[0,674,1130,822,708,628,856,320,662,388,730,308,194,0,342,422,536,354],
[0,1016,788,1164,1050,514,514,662,320,274,388,650,536,342,0,764,194,468],
[0,868,1552,560,674,1050,1278,742,1084,810,1152,274,388,422,764,0,798,776],
[0,1210,754,1358,1244,708,480,856,514,468,354,844,730,536,194,798,0,662],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], # from Sink
]

# Demands (key: node, value: amount)
DEMAND = {1: 1, 2: 1, 3: 2, 4: 4, 5: 2, 6: 4, 7: 8, 8: 8, 9: 1, 10: 2, 11: 1, 12: 2, 13:␣
↪4, 14: 4, 15: 8, 16: 8}

# The matrix is transformed into a DiGraph
A = array(DISTANCES, dtype=[("cost", int)])
G = from_numpy_matrix(A, create_using=nx.DiGraph())

# The demands are stored as node attributes
set_node_attributes(G, values=DEMAND, name="demand")

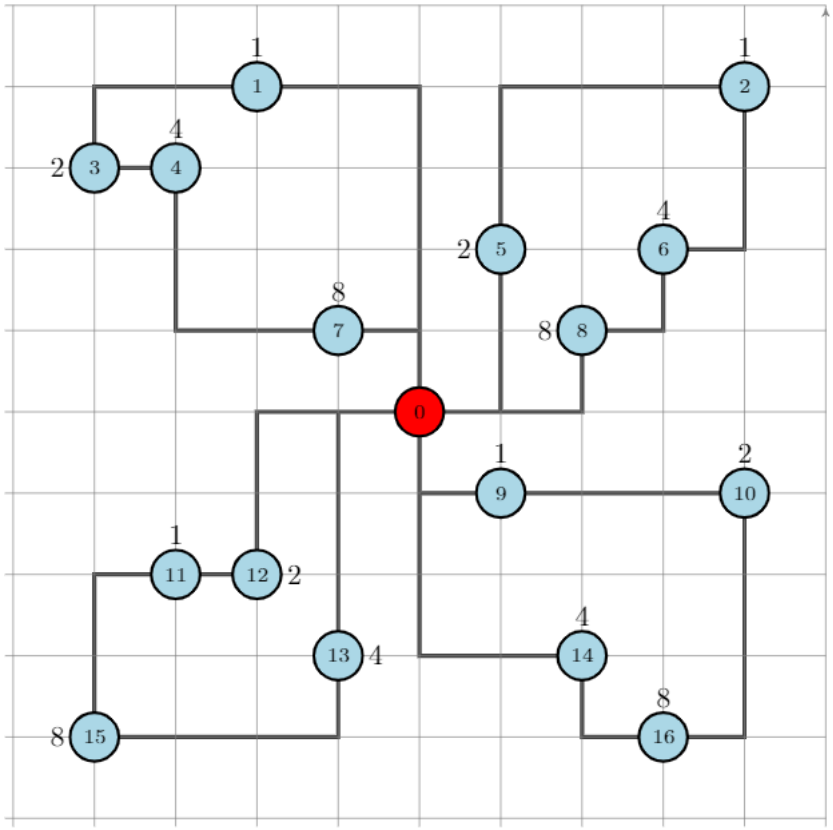
# The depot is relabeled as Source and Sink
G = relabel_nodes(G, {0: "Source", 17: "Sink"})
```

CVRP

Once the graph is properly defined, creating a CVRP and solving it is straightforward. With a maximum load of 15 units per vehicle:

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G, load_capacity=15)
>>> prob.solve()
>>> prob.best_value
6208.0
>>> prob.best_routes
{1: ['Source', 12, 11, 15, 13, 'Sink'], 2: ['Source', 1, 3, 4, 7, 'Sink'], 3: ['Source',␣
↪5, 2, 6, 8, 'Sink'], 4: ['Source', 14, 16, 10, 9, 'Sink']}
>>> prob.best_routes_load
{1: 15, 2: 15, 3: 15, 4: 15}
```

The four routes are displayed below:



CVRP with simultaneous distribution and collection

We follow with the exact same configuration, but this time, every time a node is visited, the vehicle unloads its demand and loads some waste material.

```
from networkx import DiGraph, from_numpy_matrix, relabel_nodes, set_node_attributes
from numpy import array

# Distance matrix
DISTANCES = [
[0,548,776,696,582,274,502,194,308,194,536,502,388,354,468,776,662,0], # from Source
[0,0,684,308,194,502,730,354,696,742,1084,594,480,674,1016,868,1210,548],
[0,684,0,992,878,502,274,810,468,742,400,1278,1164,1130,788,1552,754,776],
[0,308,992,0,114,650,878,502,844,890,1232,514,628,822,1164,560,1358,696],
[0,194,878,114,0,536,764,388,730,776,1118,400,514,708,1050,674,1244,582],
[0,502,502,650,536,0,228,308,194,240,582,776,662,628,514,1050,708,274],
[0,730,274,878,764,228,0,536,194,468,354,1004,890,856,514,1278,480,502],
[0,354,810,502,388,308,536,0,342,388,730,468,354,320,662,742,856,194],
[0,696,468,844,730,194,194,342,0,274,388,810,696,662,320,1084,514,308],
[0,742,742,890,776,240,468,388,274,0,342,536,422,388,274,810,468,194],
[0,1084,400,1232,1118,582,354,730,388,342,0,878,764,730,388,1152,354,536],
[0,594,1278,514,400,776,1004,468,810,536,878,0,114,308,650,274,844,502],
[0,480,1164,628,514,662,890,354,696,422,764,114,0,194,536,388,730,388],
[0,674,1130,822,708,628,856,320,662,388,730,308,194,0,342,422,536,354],
[0,1016,788,1164,1050,514,514,662,320,274,388,650,536,342,0,764,194,468],
[0,868,1552,560,674,1050,1278,742,1084,810,1152,274,388,422,764,0,798,776],
```

(continues on next page)

(continued from previous page)

```
[0,1210,754,1358,1244,708,480,856,514,468,354,844,730,536,194,798,0,662],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], # from Sink
]

# Delivery demands (key: node, value: amount)
DEMAND = {1: 1, 2: 1, 3: 2, 4: 4, 5: 2, 6: 4, 7: 8, 8: 8, 9: 1, 10: 2, 11: 1, 12: 2, 13:␣
↪4, 14: 4, 15: 8, 16: 8}

# Pickup waste (key: node, value: amount)
COLLECT = {1: 1, 2: 1, 3: 1, 4: 1, 5: 2, 6: 1, 7: 4, 8: 1, 9: 1, 10: 2, 11: 3, 12: 2,␣
↪13: 4, 14: 2, 15: 1, 16: 2}

# The matrix is transformed into a DiGraph
A = array(DISTANCES, dtype=[("cost", int)])
G = from_numpy_matrix(A, create_using=nx.DiGraph())

# The distribution and collection amounts are stored as node attributes
set_node_attributes(G, values=DEMAND, name="demand")
set_node_attributes(G, values=COLLECT, name="collect")

# The depot is relabeled as Source and Sink
G = relabel_nodes(G, {0: "Source", 17: "Sink"})
```

The *load_capacity* is unchanged, and the `distribution_collection` attribute is set to `True`.

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G, load_capacity=15, distribution_collection=True)
>>> prob.solve()
    >>> prob.best_value
6208.0
>>> prob.best_routes
{1: ['Source', 12, 11, 15, 13, 'Sink'], 2: ['Source', 1, 3, 4, 7, 'Sink'], 3: ['Source',␣
↪5, 2, 6, 8, 'Sink'], 4: ['Source', 14, 16, 10, 9, 'Sink']}
>>> prob.node_load
{1: {7: 4, 3: 5, 4: 8, 1: 8, 'Sink': 8}, 2: {8: 7, 6: 10, 2: 10, 5: 10, 'Sink': 10}, 3:
↪{14: 2, 16: 8, 10: 8, 9: 8, 'Sink': 8}, 4: {13: 0, 15: 7, 11: 5, 12: 5, 'Sink': 5}}
```

The optimal solution is unchanged. This is understandable, as for each node, the distribution volume is greater than (or equals) the pickup volume.

VRP with time windows

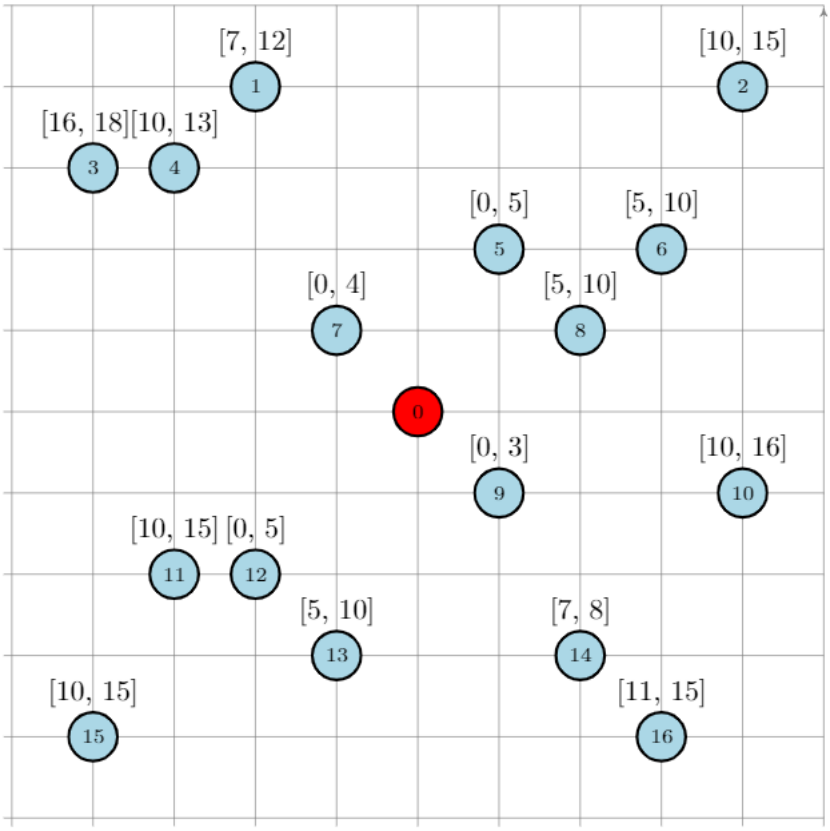
Each node must now be serviced within a time window. The time windows are displayed above each node:

This time, the network is defined by its distance matrix and its time matrix:

```
from networkx import DiGraph, from_numpy_matrix, relabel_nodes, set_node_attributes
from numpy import array

# Distance matrix
DISTANCES = [
    [0,548,776,696,582,274,502,194,308,194,536,502,388,354,468,776,662,0], # from Source
```

(continues on next page)



(continued from previous page)

```
    [0,0,684,308,194,502,730,354,696,742,1084,594,480,674,1016,868,1210,548],
    [0,684,0,992,878,502,274,810,468,742,400,1278,1164,1130,788,1552,754,776],
    [0,308,992,0,114,650,878,502,844,890,1232,514,628,822,1164,560,1358,696],
    [0,194,878,114,0,536,764,388,730,776,1118,400,514,708,1050,674,1244,582],
    [0,502,502,650,536,0,228,308,194,240,582,776,662,628,514,1050,708,274],
    [0,730,274,878,764,228,0,536,194,468,354,1004,890,856,514,1278,480,502],
    [0,354,810,502,388,308,536,0,342,388,730,468,354,320,662,742,856,194],
    [0,696,468,844,730,194,194,342,0,274,388,810,696,662,320,1084,514,308],
    [0,742,742,890,776,240,468,388,274,0,342,536,422,388,274,810,468,194],
    [0,1084,400,1232,1118,582,354,730,388,342,0,878,764,730,388,1152,354,536],
    [0,594,1278,514,400,776,1004,468,810,536,878,0,114,308,650,274,844,502],
    [0,480,1164,628,514,662,890,354,696,422,764,114,0,194,536,388,730,388],
    [0,674,1130,822,708,628,856,320,662,388,730,308,194,0,342,422,536,354],
    [0,1016,788,1164,1050,514,514,662,320,274,388,650,536,342,0,764,194,468],
    [0,868,1552,560,674,1050,1278,742,1084,810,1152,274,388,422,764,0,798,776],
    [0,1210,754,1358,1244,708,480,856,514,468,354,844,730,536,194,798,0,662],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], # from Sink
    ]

TRAVEL_TIMES = [
    [0, 6, 9, 8, 7, 3, 6, 2, 3, 2, 6, 6, 4, 4, 5, 9, 7, 0],  # from source
    [0, 0, 8, 3, 2, 6, 8, 4, 8, 8, 13, 7, 5, 8, 12, 10, 14, 6],
    [0, 8, 0, 11, 10, 6, 3, 9, 5, 8, 4, 15, 14, 13, 9, 18, 9, 9],
    [0, 3, 11, 0, 1, 7, 10, 6, 10, 10, 14, 6, 7, 9, 14, 6, 16, 8],
```

(continues on next page)

(continued from previous page)

```
    [0, 2, 10, 1, 0, 6, 9, 4, 8, 9, 13, 4, 6, 8, 12, 8, 14, 7],
    [0, 6, 6, 7, 6, 0, 2, 3, 2, 2, 7, 9, 7, 7, 6, 12, 8, 3],
    [0, 8, 3, 10, 9, 2, 0, 6, 2, 5, 4, 12, 10, 10, 6, 15, 5, 6],
    [0, 4, 9, 6, 4, 3, 6, 0, 4, 4, 8, 5, 4, 3, 7, 8, 10, 2],
    [0, 8, 5, 10, 8, 2, 2, 4, 0, 3, 4, 9, 8, 7, 3, 13, 6, 3],
    [0, 8, 8, 10, 9, 2, 5, 4, 3, 0, 4, 6, 5, 4, 3, 9, 5, 2],
    [0, 13, 4, 14, 13, 7, 4, 8, 4, 4, 0, 10, 9, 8, 4, 13, 4, 6],
    [0, 7, 15, 6, 4, 9, 12, 5, 9, 6, 10, 0, 1, 3, 7, 3, 10, 6],
    [0, 5, 14, 7, 6, 7, 10, 4, 8, 5, 9, 1, 0, 2, 6, 4, 8, 4],
    [0, 8, 13, 9, 8, 7, 10, 3, 7, 4, 8, 3, 2, 0, 4, 5, 6, 4],
    [0, 12, 9, 14, 12, 6, 6, 7, 3, 3, 4, 7, 6, 4, 0, 9, 2, 5],
    [0, 10, 18, 6, 8, 12, 15, 8, 13, 9, 13, 3, 4, 5, 9, 0, 9, 9],
    [0, 14, 9, 16, 14, 8, 5, 10, 6, 5, 4, 10, 8, 6, 2, 9, 0, 7],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # from sink
    ]

# Time windows (key: node, value: lower/upper bound)
TIME_WINDOWS_LOWER = {0: 0, 1: 7, 2: 10, 3: 16, 4: 10, 5: 0, 6: 5, 7: 0, 8: 5, 9: 0, 10:␣
↪10, 11: 10, 12: 0, 13: 5, 14: 7, 15: 10, 16: 11,}
TIME_WINDOWS_UPPER = {1: 12, 2: 15, 3: 18, 4: 13, 5: 5, 6: 10, 7: 4, 8: 10, 9: 3, 10: 16,
↪ 11: 15, 12: 5, 13: 10, 14: 8, 15: 15, 16: 15,}

# Transform distance matrix into DiGraph
A = array(DISTANCES, dtype=[("cost", int)])
G_d = from_numpy_matrix(A, create_using=DiGraph())

# Transform time matrix into DiGraph
A = array(TRAVEL_TIMES, dtype=[("time", int)])
G_t = from_numpy_matrix(A, create_using=DiGraph())

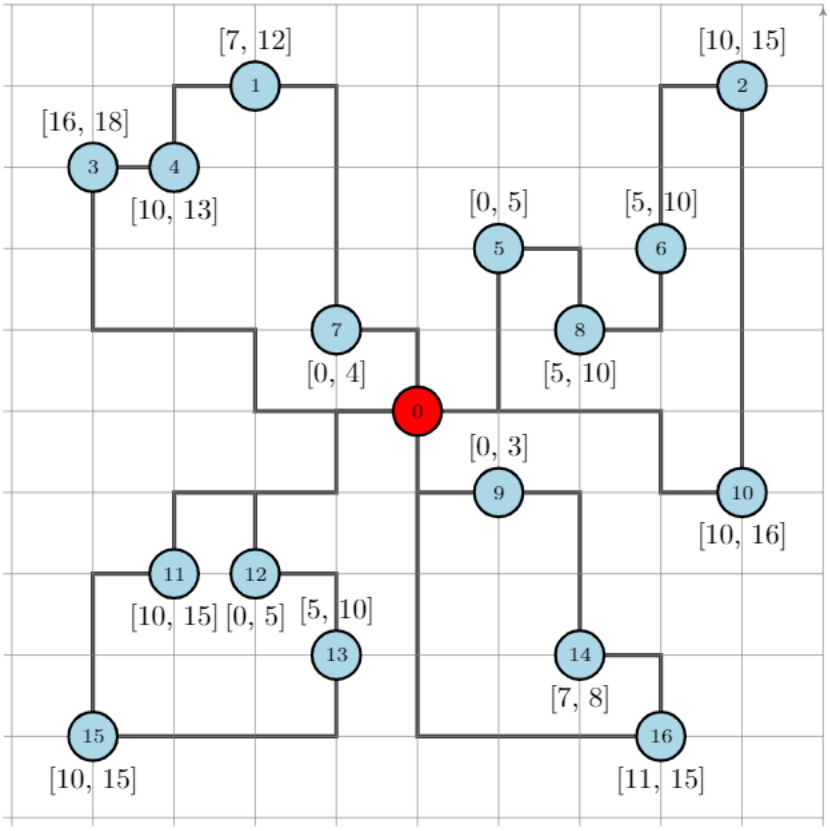
# Merge
G = compose(G_d, G_t)

# Set time windows
set_node_attributes(G, values=TIME_WINDOWS_LOWER, name="lower")
set_node_attributes(G, values=TIME_WINDOWS_UPPER, name="upper")

# The VRP is defined and solved
prob = VehicleRoutingProblem(G, time_windows=True)
prob.solve()
```

The solution is displayed below:

```
>>> prob.best_value
    6528.0
>>> prob.best_routes
    {1: ['Source', 9, 14, 16, 'Sink'], 2: ['Source', 12, 13, 15, 11, 'Sink'], 3: ['Source
↪', 5, 8, 6, 2, 10, 'Sink'], 4: ['Source', 7, 1, 4, 3, 'Sink']}
>>> prob.arrival_time
    {1: {9: 2, 14: 7, 16: 11, 'Sink': 18}, 2: {12: 4, 13: 6, 15: 11, 11: 14, 'Sink': 20},
↪ 3: {5: 3, 8: 5, 6: 7, 2: 10, 10: 14, 'Sink': 20}, 4: {7: 2, 1: 7, 4: 10, 3: 16, 'Sink
↪': 24}}
```



CVRP with pickups and deliveries

In this variant, each demand is made of a pickup node and a delivery node. Each pickup/delivery pair (or request) must be assigned to the same tour, and within this tour, the pickup node must be visited prior to the delivery node (as an item that is yet to be picked up cannot be delivered). The total load must not exceed the vehicle's capacity. The requests are displayed below:

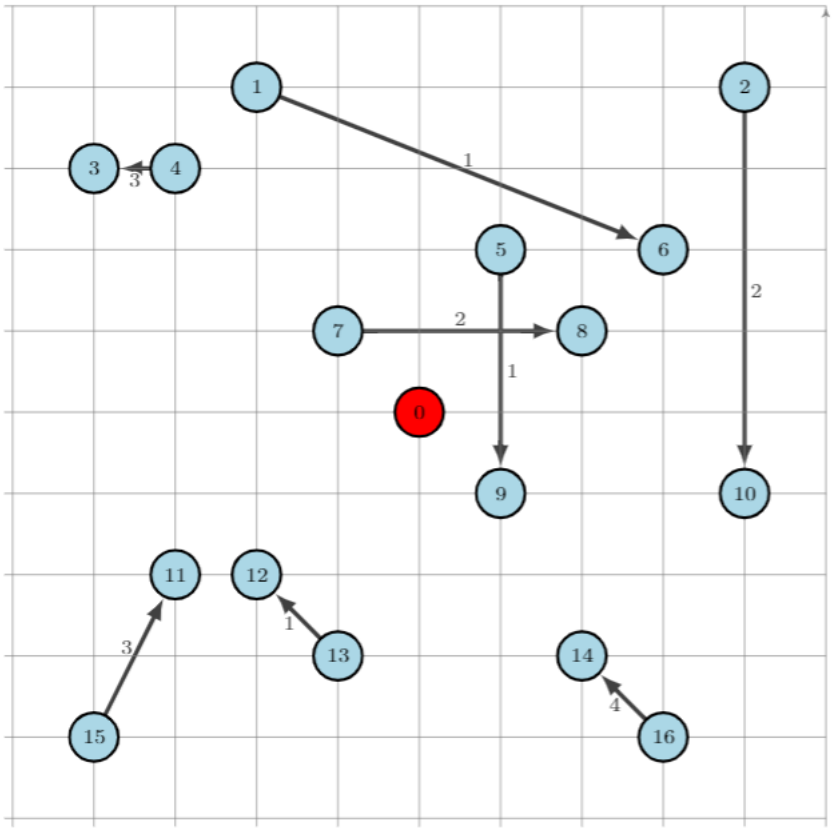
The network is defined as previously, and we add the following data to take into account each request:

```
# Requests (from_node, to_node) : amount
pickups_deliveries = {(1, 6): 1, (2, 10): 2, (4, 3): 3, (5, 9): 1, (7, 8): 2, (15, 11):␣
↪3, (13, 12): 1, (16, 14): 4}
for (u, v) pickups_deliveries:
    G.nodes[u]["request"] = v
    # Pickups are accounted for positively
    G.nodes[u]["demand"] = pickups_deliveries[(u, v)]
    # Deliveries are accounted for negatively
    G.nodes[v]["demand"] = -pickups_deliveries[(u, v)]
```

We can now create a pickup and delivery instance with a maximum load of 6 units per vehicle, and with at most 6 stops:

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G, load_capacity=6, num_stops=6, pickup_delivery=True)
>>> prob.solve(cspy=False)
>>> prob.best_value
5980.0
```

(continues on next page)



(continued from previous page)

```
>>> prob.best_routes
{1: ['Source', 5, 2, 10, 16, 14, 9, 'Sink'], 2: ['Source', 7, 4, 3, 1, 6, 8, 'Sink'], 3:␣
↪['Source', 13, 15, 11, 12, 'Sink']}
>>> prob.node_load
{1: {5: 1, 2: 3, 10: 1, 16: 5, 14: 1, 9: 0, 'Sink': 0}, 2: {7: 2, 4: 5, 3: 2, 1: 3, 6: 2,
↪ 8: 0, 'Sink': 0}, 3: {13: 1, 15: 4, 11: 1, 12: 0, 'Sink': 0}}
```

The four routes are displayed below:

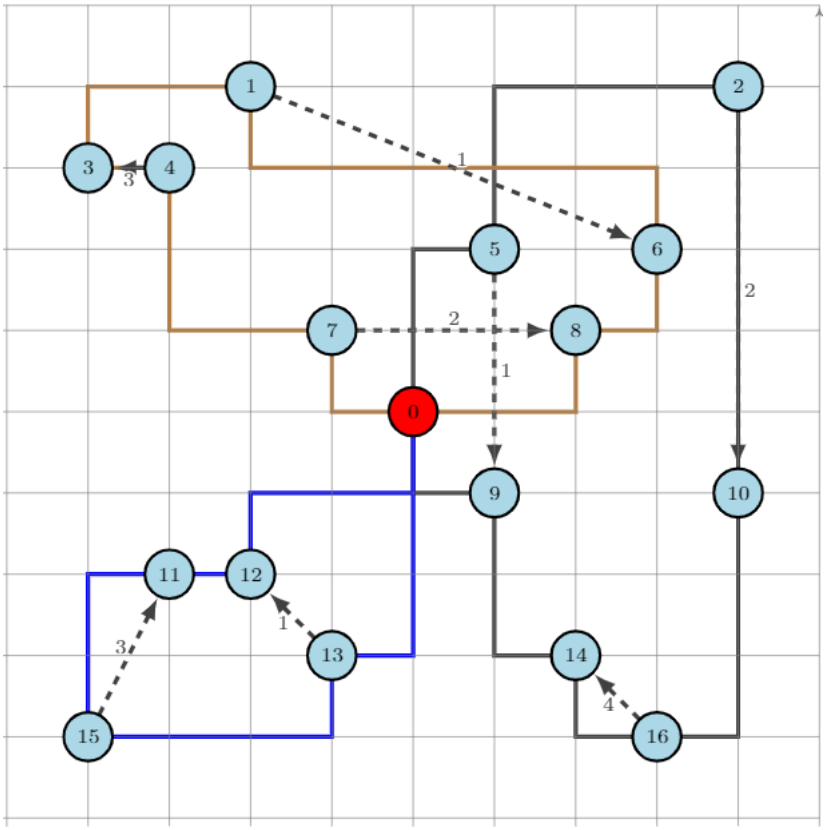
Limited fleet and dropping visits

This last example is similar to the above CVRP, except for the fact that demands have increased, and that the fleet is limited to 4 vehicles, with a 15 unit capacity (per vehicle). Since the total demand is greater than $4 \times 15 = 60$, servicing each node is not possible, therefore, we will try to visit as many customers as possible, and allow dropping visits, at the cost of a 1000 penalty.

```
from networkx import DiGraph, from_numpy_matrix, relabel_nodes, set_node_attributes
from numpy import array

# Distance matrix
DISTANCES = [
[0,548,776,696,582,274,502,194,308,194,536,502,388,354,468,776,662,0], # from Source
[0,0,684,308,194,502,730,354,696,742,1084,594,480,674,1016,868,1210,548],
```

(continues on next page)



(continued from previous page)

```
[0,684,0,992,878,502,274,810,468,742,400,1278,1164,1130,788,1552,754,776],
[0,308,992,0,114,650,878,502,844,890,1232,514,628,822,1164,560,1358,696],
[0,194,878,114,0,536,764,388,730,776,1118,400,514,708,1050,674,1244,582],
[0,502,502,650,536,0,228,308,194,240,582,776,662,628,514,1050,708,274],
[0,730,274,878,764,228,0,536,194,468,354,1004,890,856,514,1278,480,502],
[0,354,810,502,388,308,536,0,342,388,730,468,354,320,662,742,856,194],
[0,696,468,844,730,194,194,342,0,274,388,810,696,662,320,1084,514,308],
[0,742,742,890,776,240,468,388,274,0,342,536,422,388,274,810,468,194],
[0,1084,400,1232,1118,582,354,730,388,342,0,878,764,730,388,1152,354,536],
[0,594,1278,514,400,776,1004,468,810,536,878,0,114,308,650,274,844,502],
[0,480,1164,628,514,662,890,354,696,422,764,114,0,194,536,388,730,388],
[0,674,1130,822,708,628,856,320,662,388,730,308,194,0,342,422,536,354],
[0,1016,788,1164,1050,514,514,662,320,274,388,650,536,342,0,764,194,468],
[0,868,1552,560,674,1050,1278,742,1084,810,1152,274,388,422,764,0,798,776],
[0,1210,754,1358,1244,708,480,856,514,468,354,844,730,536,194,798,0,662],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], # from Sink
]

# Demands (key: node, value: amount)
DEMAND = {1: 1, 2: 1, 3: 3, 4: 6, 5: 3, 6: 6, 7: 8, 8: 8, 9: 1, 10: 2, 11: 1, 12: 2, 13:␣
↪6, 14: 6, 15: 8, 16: 8}

# The matrix is transformed into a DiGraph
A = array(DISTANCES, dtype=[("cost", int)])
```

(continues on next page)

(continued from previous page)

```
G = from_numpy_matrix(A, create_using=nx.DiGraph())

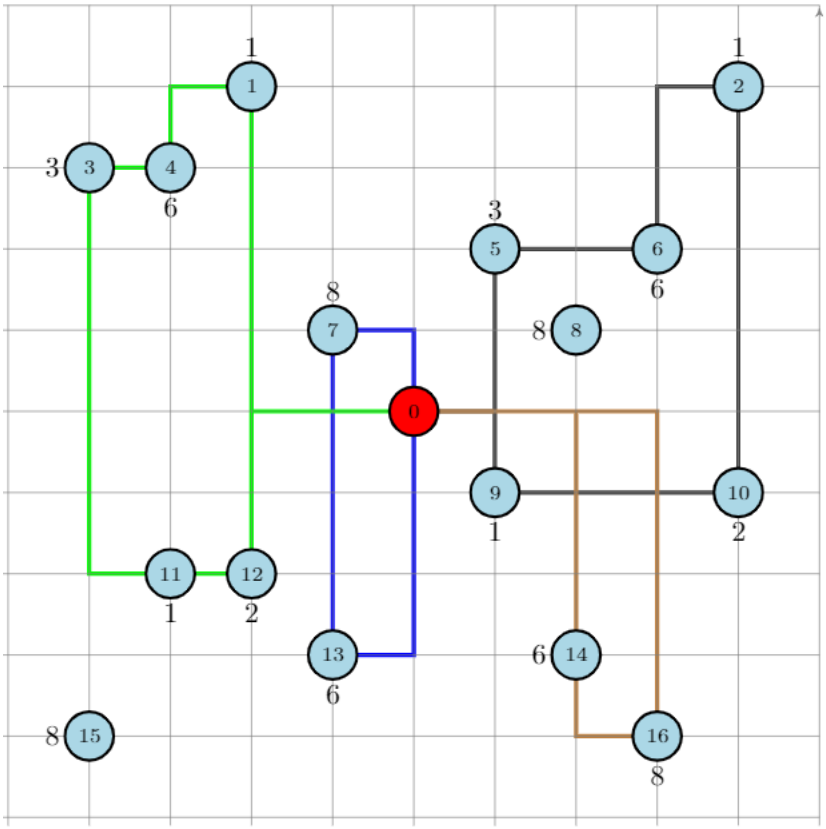
# The demands are stored as node attributes
set_node_attributes(G, values=DEMAND, name="demand")

# The depot is relabeled as Source and Sink
G = relabel_nodes(G, {0: "Source", 17: "Sink"})
```

Once the graph is properly defined, a VRP instance is created, with attributes *num_vehicles* and *drop_penalty*:

```
>>> from vrpy import VehicleRoutingProblem
>>> prob = VehicleRoutingProblem(G, load_capacity=15, num_vehicles=4, drop_penalty=1000)
>>> prob.solve()
>>> prob.best_value
7776.0
>>> prob.best_routes
{1: ['Source', 9, 10, 2, 6, 5, 'Sink'], 2: ['Source', 7, 13, 'Sink'], 3: ['Source', 14,␣
↪16, 'Sink'], 4: ['Source', 1, 4, 3, 11, 12, 'Sink']}
>>> prob.best_routes_load
{1: 13, 2: 14, 3: 14, 4: 13}
```

The solver drops nodes 8 and 15. The new optimal routes are displayed below:



4.6 API

4.6.1 vrpy.VehicleRoutingProblem

class vrpy.vrp.**VehicleRoutingProblem**(*G*, *num_stops=None*, *load_capacity=None*, *duration=None*, *time_windows=False*, *pickup_delivery=False*, *distribution_collection=False*, *drop_penalty=None*, *fixed_cost=0*, *num_vehicles=None*, *use_all_vehicles=False*, *periodic=None*, *mixed_fleet=False*, *minimize_global_span=False*)

Stores the underlying network of the VRP and parameters for solving with a column generation approach.

Parameters

- **G** (*DiGraph*) – The underlying network.
- **num_stops** (*int, optional*) – Maximum number of stops. Defaults to None.
- **load_capacity** (*list, optional*) – Maximum load per vehicle. Each item of the list points to a different capacity. Defaults to None.
- **duration** (*int, optional*) – Maximum duration of route. Defaults to None.
- **time_windows** (*bool, optional*) – True if time windows on vertices. Defaults to False.
- **pickup_delivery** (*bool, optional*) – True if pickup and delivery constraints. Defaults to False.
- **distribution_collection** (*bool, optional*) – True if distribution and collection are simultaneously enforced. Defaults to False.
- **drop_penalty** (*int, optional*) – Value of penalty if node is dropped. Defaults to None.
- **fixed_cost** (*int, optional*) – Fixed cost per vehicle. Defaults to 0.
- **num_vehicles** (*int, optional*) – Maximum number of vehicles available. Defaults to None (in this case num_vehicles is unbounded).
- **use_all_vehicles** (*bool, optional*) – True if all vehicles specified by num_vehicles should be used. Defaults to False.
- **periodic** (*int, optional*) – Time span if vertices are to be visited periodically. Defaults to None.
- **mixed_fleet** (*bool, optional*) – True if heterogeneous fleet. Defaults to False.
- **minimize_global_span** (*bool, optional*) – True if global span is minimized (instead of total cost). Defaults to False.

property arrival_time
Returns nested dict. First key : route id ; second key : node ; value : arrival time.

property best_routes
Returns dict of best routes found. Keys : route_id; values : list of ordered nodes from Source to Sink.

property best_routes_cost
Returns dict with route ids as keys and route costs as values.

property best_routes_duration
Returns dict with route ids as keys and route durations as values.

property best_routes_load
Returns dict with route ids as keys and route loads as values.

property best_value
Returns value of best solution found.

property departure_time
Returns nested dict. First key : route id ; second key : node ; value : departure time.

property node_load
Returns nested dict. First key : route id ; second key : node ; value : load. If truck is collecting, load refers to accumulated load on truck. If truck is distributing, load refers to accumulated amount that has been unloaded. If simultaneous distribution and collection, load refers to truck load when leaving the node.

property schedule
If Periodic CVRP, returns a dict with keys a day number and values the route IDs scheduled this day.

solve(*initial_routes=None*, *preassignments=None*, *pricing_strategy='BestEdges1'*, *cspy=True*, *exact=False*, *time_limit=None*, *solver='cbc'*, *dive=False*, *greedy=False*, *max_iter=None*, *run_exact=1*, *heuristic_only=False*)
Iteratively generates columns with negative reduced cost and solves as MIP.

Parameters

- **initial_routes** (*list, optional*) – List of routes (ordered list of nodes). Feasible solution for first iteration. Defaults to None.
- **preassignments** (*list, optional*) – List of preassigned routes (ordered list of nodes). If the route contains Source and Sink nodes, it is locked, otherwise it may be extended. Defaults to None.
- **pricing_strategy** (*str, optional*) – Strategy used for solving the sub problem. Options available :
  - ”Exact”: the subproblem is solved exactly;
  - ”BestEdges1”: some edges are removed;
  - ”BestEdges2”: some edges are removed (with a different strategy);
  - ”BestPaths”: some edges are removed (with a different strategy);
  - ”Hyper”: choose from the above using a hyper_heuristic (see hyper_heuristic.py);

  Defaults to “BestEdges1”.
- **cspy** (*bool, optional*) – True if cspy is used for subproblem. Defaults to True.
- **exact** (*bool, optional*) – True if only cspy’s exact algorithm is used to generate columns. Otherwise, heuristics will be used until they produce +ve reduced cost columns, after which the exact algorithm is used. Defaults to False.
- **time_limit** (*int, optional*) – Maximum number of seconds allowed for solving (for finding columns). Defaults to None.
- **solver** (*str, optional*) – Solver used. Three options available: “cbc”, “cplex”, “gurobi”. Using “cplex” or “gurobi” requires installation. Not available by default. Additionally, “gurobi” requires pulp to be installed from source. Defaults to “cbc”, available by default.
- **dive** (*bool, optional*) – True if diving heuristic is used. Defaults to False.
- **greedy** (*bool, optional*) – True if randomized greedy algorithm is used to generate extra columns. Only valid for capacity constraints, time constraints, num stops constraints. Defaults to False.

- **max_iter** (*int, optional*) – maximum number of iterations for the column generation procedure.
- **run_exact** (*int, optional*) – if a pricing strategy is selected, this parameter controls the number of iterations after which the exact algorithm is run. Defaults to 1.
- **heuristic_only** (*bool, optional*) – True if Clarke & Wright solution is return without entering column generation loop. Defaults to False.

Returns Optimal solution of MIP based on generated columns

Return type float

4.6.2 Notes

The input graph must have single *Source* and *Sink* nodes with no incoming or outgoing edges respectively. These dummy nodes represent the depot which is split for modeling convenience. The *Source* and *Sink* cannot have a demand, if one is given it is ignored with a warning.

Please read sections *Vehicle Routing Problems*, *Solving Options* and *Examples* for details and examples on each of the above arguments.

4.7 Mathematical Background

4.7.1 A column generation approach

VRPy solves vehicle routing problems with a column generation approach. The term *column generation* refers to the fact that iteratively, routes (or *columns*) are *generated* with a pricing problem, and fed to a master problem which selects the best routes among a pool such that each vertex is serviced exactly once. The linear formulations of these problems are detailed hereafter.

Master Problem

Let $G = (V, A)$ be a graph where V denotes the set of nodes that have to be visited, and A the set of edges of the network. Let Ω be the set of feasible routes. Let λ_r be a binary variable that takes value 1 if and only if route $r \in \Omega$ with cost c_r is selected. The master problem reads as follows:

$$\min \sum_{r \in \Omega} c_r \lambda_r$$

subject to set covering constraints:

$$\sum_{r \in \Omega | v \in r} \lambda_r = 1 \qquad \forall v \in V \quad (1)$$

$$\lambda_r \in \{0, 1\} \qquad \forall r \in \Omega \quad (2)$$

When using a column generation procedure, integrity constraints (2) are relaxed (such that $0 \leq \lambda_r \leq 1$), and only a subset of Ω is used. This subset is generated dynamically with the following sub problem.

Pricing problem

Let π_v denote the dual variable associated with constraints (1). The marginal cost of a variable (or column) λ_r is given by:

$$\hat{c}_r = c_r - \sum_{v \in V | v \in r} \pi_v$$

Therefore, if x_{uv} is a binary variable that takes value 1 if and only if edge (u, v) is used, *assuming there are no negative cost sub cycles*, one can formulate the problem of finding a route with negative marginal cost as follows :

$$\min \quad \sum_{(u,v) \in A} c_{uv} x_{uv} - \sum_{u | (u,v) \in A} \pi_u x_{uv}$$

subject to flow balance constraints :

$$\sum_{u | (u,v) \in A} x_{uv} = \sum_{u | (v,u) \in A} x_{uv} \qquad \forall v \in V$$

$$x_{uv} \in \{0, 1\} \qquad \forall (u, v) \in A$$

In other words, the sub problem is a shortest elementary path problem, and additional constraints (such as capacities, time) give rise to a shortest path problem with *resource constraints*, hence the interest of using the *cspy* library.

If there are negative cost cycles (which typically happens), the above formulation requires additional constraints to enforce path elementarity, and the problem becomes computationally intractable. Linear formulations are then impractical, and algorithms such as the ones available in *cspy* become very handy.

4.7.2 Does VRPy return an optimal solution?

VRPy does not necessarily return an optimal solution (even with no time limit). Indeed, once the pricing problems fails to find a route with negative marginal cost, the master problem is solved as a MIP. This *price-and-branch* strategy does not guarantee optimality. Note however that it can be shown [BSL97] that asymptotically, the relative error goes to zero as the number of customers increases. To guarantee that an optimal solution is returned, the column generation procedure should be embedded in a branch-and-bound scheme (*branch-and-price*). This is part of the future work listed below.

4.7.3 TO DO

- Embed the solving procedure in a branch-and-bound scheme:
  - branch-and-price (exact)
  - diving (heuristic)
- Implement heuristics for initial solutions.
- More acceleration strategies:
  - other heuristic pricing strategies
  - switch to other LP modeling library (?)
  - improve stabilization
  - ...
- Include more VRP variants:
  - pickup and delivery with cspy
  - ...

4.8 Performance profiles

Performance profiles are a practical way to have a global overview of a set of algorithms' performances. On the x axis, we have the relative gap (%), and on the y axis, the percentage of data sets solved within the gap. So for example, at the intersection with the y axis is the percentage of data sets solved optimally, and at the intersection with $y = 100\%$ is the relative gap within which all data sets are solved.

At a glance, the more the curve is in the upper left corner, the better the algorithm.

We compare the performances of *vrpy* and *OR-Tools* (default options):

- on Augerat's instances (CVRP),
- on Solomon's instances (CVRPTW)

Results are found *here* [link to repo] and can be replicated.

4.8.1 CVRP



We can see that with a maximum running time of 10 seconds, *OR-Tools* solves 15% of the instances optimally, while *vrpy* only solves 5% of them. Both solve approximately 43% of instances with a maximum relative gap of 5%. And both solve all instances within a maximum gap of 25%.

4.8.2 CVRPTW

Coming soon.

4.9 Bibliography

- genindex
- search

BIBLIOGRAPHY

[BSL97] Julien Bramel and David Simchi-Levi. Solving the vrp using a column generation approach. In *The Logic of Logistics*, pages 125–141. Springer, 1997.

[CW64] Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[DellAmicoRS06] Mauro Dell'Amico, Giovanni Righini, and Matteo Salani. A branch-and-price approach to the vehicle routing problem with simultaneous distribution and collection. *Transportation science*, 40(2):235–247, 2006.

[DOzcanB12] John H Drake, Ender Özcan, and Edmund K Burke. An improved choice function heuristic selection for cross domain heuristic search. In *International Conference on Parallel Problem Solving from Nature*, 307–316. Springer, 2012.

[FGonccalvesP17] Alexandre Silvestre Ferreira, Richard Aderbal Gonçalves, and Aurora Pozo. A multi-armed bandit selection strategy for hyper-heuristics. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, 525–532. IEEE, 2017.

[FRV+18] J Forrest, T Ralphs, S Vigerske, B Kristjansson, M Lubin, H Santos, M Saltzman, and others. Coin-or/cbc: version 2.9. 9. *DOI: https://doi. org/10.5281/zenodo*, 2018.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical Report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[Oli06] Travis Oliphant. NumPy: a guide to NumPy. USA: Trelgol Publishing, 2006. [Online; accessed 18/05/2020]. URL: http://www.numpy.org/.

[PF] Laurent Perron and Vincent Furnon. OR-Tools. URL: https://developers.google.com/optimization/.

[SZS15] Nasser R Sabar, Xiuzhen Jenny Zhang, and Andy Song. A math-hyper-heuristic approach for large-scale vehicle routing problems with time windows. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, 830–837. IEEE, 2015.

[SPR18] Alberto Santini, Christian EM Plum, and Stefan Ropke. A branch-and-price approach to the feeder network design problem. *European Journal of Operational Research*, 264(2):607–622, 2018.

[TRothenbacherGI17] Christian Tilk, Ann-Kathrin Rothenbächer, Timo Gschwind, and Stefan Irnich. Asymmetry matters: dynamic half-way points in bidirectional labeling for solving shortest path problems with resource constraints faster. *European Journal of Operational Research*, 261(2):530–539, 2017.

[TS19] David Torres Sanchez. cspy : A Python package with a collection of algorithms for the (Resource) Constrained Shortest Path problem. 2019. URL: https://github.com/torressa/cspy.

PYTHON MODULE INDEX

V

vrpy.vrp, 33

INDEX

A

arrival_time (*vrpy.vrp.VehicleRoutingProblem property*), 33

B

best_routes (*vrpy.vrp.VehicleRoutingProblem property*), 33
best_routes_cost (*vrpy.vrp.VehicleRoutingProblem property*), 33
best_routes_duration (*vrpy.vrp.VehicleRoutingProblem property*), 33
best_routes_load (*vrpy.vrp.VehicleRoutingProblem property*), 33
best_value (*vrpy.vrp.VehicleRoutingProblem property*), 33

D

departure_time (*vrpy.vrp.VehicleRoutingProblem property*), 34

M

module
    vrpy.vrp, 33

N

node_load (*vrpy.vrp.VehicleRoutingProblem property*), 34

S

schedule (*vrpy.vrp.VehicleRoutingProblem property*), 34
solve() (*vrpy.vrp.VehicleRoutingProblem method*), 34

V

VehicleRoutingProblem (*class in vrpy.vrp*), 33
vrpy.vrp
    module, 33