Automatic tuning and configuration of metaheuristics for Inventory Routing Problem

Polytechnic University of Milan

&

Université Libre de Bruxelles

Master thesis

Author:
Antonio Maria Fiscarelli

Advisors:
Prof. Pier Luca Lanzi
Prof. Thomas Stützle

Contents

Chapter 1

Inventory Routing Problem

The Inventory Routing Problem (IRP) can be described as the combination of vehicle routing and inventory management problems, where a supplier has to deliver products to several customers, subject to side constraints. It provides integrated logistics solutions by simultaneously optimizing vehicle routing, inventory management and delivery scheduling [3]. Several applications of the IRP have been documented. Some of them arise in maritime logistics, namely in ship routing and inventory management. Problems arising in the chemical components industry and in the oil and gas industries are also a frequent source of applications in a maritime environment. Non-maritime applications of the IRP arise in a large variety of industries, including the distribution of gas using tanker trucks, road-based distribution of automobile components and of perishable items. Other applications include the transportation of groceries, cement, fuel, blood, and waste organic oil.

1.1 IRP classification

The IRP integrates inventory management, vehicle routing and delivery scheduling decisions. The supplier has to deal with three different problems: when to serve a given customer, how much to deliver to this customer when it is served, how to combine customers into delivery routes. Many variants of the IRP have been described over the past 30 years. There is no standard version of the problem, but IRPs can be classified according to 7 criteria:

- The horizon for IRP model can either be finite or infinite.
- The number of suppliers and customers may vary. The structure can be one-to-one when there is only one supplier serving one customer, one-to-many with one supplier and several customers, or many-to-many with several suppliers and several customers.
- Routing can be direct when there is one customer per route, multiple when there are several customers per route, or continuous when there is no central depot.
- Inventory policies define pre-established rules to replenish customers. Under the maximum level (ML) policy, the replenishment level can vary and it's bounded by each customer's capacity. Under the order-up-to level (OU) policy, the replenishment level is that to fill its inventory capacity.
- Inventory decisions determine how inventory management is modeled. If the inventory is allowed to become negative, then back-ordering occurs and the corresponding demand will be served at a later stage; if there are no back-orders, then the extra demand is considered as lost sales. In both cases there may exist a penalty for the stockout. In deterministic contexts, the inventory is restricted to be non-negative.

- The fleet can either be homogeneous or heterogeneous.
- The number of vehicles available may be fixed at one, many, or be unconstrained.

Another classification refers to the time at which information on demand becomes known. If it is fully known to the decision maker at the beginning of the planning horizon, the problem is then deterministic; if its probability distribution is known, the problem is then stochastic (SIRP); if demand is not fully known in advance, but is gradually revealed over time, the problem is then dynamic (DIRP). For the DIRP, one can still exploit its statistical distribution: in this case the problem is a Dynamic and Stochastic Inventory-Routing Problem (DSIRP). [3]

1.2 Different IRP's in literature

Since the particular IRP faced in this thesis was proposed only this year and there is a huge number of different IRPs in the literature, there was no published work available on this specific problem. Some papers were reviewed to have an idea about how to approach it and the most used techniques.

Paul Shaw [6] proposed an approach based on constraint programming and local search methods for solving capacitated vehicle routing problem (CVRP). The CVRP they examine has the following characteristics:

- Finite horizon.
- One-to-many structure.
- Multiple routes.
- Inventory policy not specified.
- Inventory decision not specified.
- Homogeneous fleet.
- Unconstrained number of vehicles.

The objective is to minimize the total distance traveled by all vehicles.

Goel, Furman, Song and El-Bakry [7] propose a Large Neighborhood Search (LNS) for Liquefied Natural Gas Inventory Routing (LNGIR). The LNGIR they examine has the following characteristics:

- Finite horizon.
- Many-to-many structure.
- Multiple routes.
- ML policy.
- No back-orders, lost sales penalties.
- Heterogeneous fleet.
- Number of vehicles fixed to many.

The objective is o minimize the lost production, stockout and unmet demands.

Ropke and Pisinger [8] propose an Adaptive LNS for the Pickup and Delivery Problem with Time Windows (PDPTW). The Adaptive LNS they examine has the following characteristics:

- Finite horizon.
- Many-to-many structure.
- Multiple routes.
- OU Policy.
- Deterministic context.
- Heterogeneous fleet.
- Number of vehicles fixed to many.

The objective is to minimize the total distance travelled, the total time spent by all vehicles, and the number of requests not scheduled.

Aksen, Kaya, Salman and Tuncel [9] propose an Adaptive LNS for a Selective and Periodic Inventory Routing Problem (SPIRP) for recovery and reuse of waste oil. The Adaptive LNS they examine has the following characteristics:

- Infinite horizon.
- Many-to-many structure.
- Multiple routes.
- ML policy.
- Back-orders, lost sales penalties.
- Homogeneous fleet.
- Number of vehicles fixed to many-to-many.

The objective is to minimize all transportation, number of vehicles operating, inventory holding and oil purchasing costs.

1.3 Air Liquid Inventory Routing Problem

This thesis deals with a problem called Air Liquide Inventory Routing Problem (Air Liquide IRP), proposed by the French company Air Liquide for the ROADEF/EURO 2016 challenge. The problem consists of planning bulk gas distribution in order to minimize total distribution cost and maximize quantity of gas delivered over long term.
This can be achieved by building delivery shifts to match the demand requirements subject to given resources and technical constraints.

1.3.1 Position of the problem in the Operations Research world

The main features of the problem, as described by the ROADEF/EURO 2016's committee in the model description [4], are:

- Product and sourcing:
  - One product: Liquid Oxygen at cryogenic temperature (approximately -220ÂřC).
  - One single production site (source) with unlimited product available 24/7 (no capacity. constraints on production and storage).
  - Safety first: fixed loading times at sources to be able to safely handle cryogenic products.
- Customers:
  - Vendor management inventory (VMI) customers only, available 24/7 for delivery (no orders/call-in customers).
  - Customer Consumption forecast is known in advance for the whole horizon at a hourly time stamp.
  - Safety levels inside the cryogenic tanks are determined for the whole horizon to always guarantee high enough level of oxygen. No run-out are allowed before safety level.
  - Safety first: fixed unloading operation time per customer to be able to safely access site and perform all tasks.
- Transportation resource:
  - All the transportation resources are located at one single base (which may be located at a different location to the production source).
  - Several drivers with different availability time windows.
  - A driver can only drive one trailers.
  - Each trailer has a specific capacity.
- Shift definition:
  - Each shift has only one driver and only one trailers.
  - A driver has to start from the base and come back to the base.
  - Driver will be paid from the beginning of the shift at the base to the end of the shift(within the drivers availability hours).
  - Safety first: abide by legal regulations, particularly limitations on cumulative driving time.
- Objective function:
  - The objective function is to minimize the logistic ratio, which is equal to the time and distance cost divided by the total quantity delivered over the whole horizon. The time cost is proportional to the duration of the shift(mainly related to the salary of the driver), and includes driving time, the idle time, and the loading/unloading times. The distance cost is proportional to the distance traveled by a vehicle(mainly related to the fuel consumption).
- Instances size:

- 1-4 trailers/drivers, 50 to 200 customers, time horizons from 1 week to 1 month.

This problem can be classified as an IRP and several mathematical formulations already exists in literature, but it has several features that make it unique:

- the objective is rational: the goal is to minimize the logistic ratio (cost per unit delivered), or equivalently the average quantity of gas delivered per km driven and hour spent.
- the solution must satisfy specific business-related constraints that are generally not taken into consideration in the literature
  - Drivers' bases (also called sink or depot) and sources(also called pickup nodes or production terminals) are not always co-located
  - No prior assignment of drivers to trailers
  - Accurate modeling of time: Air Liquide IRP model considers an effectively continuous time (accurate to the minute) for the timing of operations and discrete time (accurate to the hour) for inventory control
  - Multi-trip problem: a shift may be composed by several trips, i.e. it can alternate loading and deliveries.

1.3.2 Problem setting

1.3.2.1 Units of measure for quantity

For quantities, the weight(Kg) has been used.

1.3.2.2 Time representation and horizon

Let us consider the scheduling horizon T. Two discrete time breakdowns of the interval [0:T[have been used: H (hours) and M (minutes). Inventory management's time granularity is defined by hourly timesteps, while Drivers and Customers' time granularity is defined by minute timesteps.

1.3.2.3 Drivers master data

The set of drivers are referred as **Drivers**. By convention, indices referring to drivers are denoted by *d*. It represents a driver with his/her characteristics. Each driver $d \in$ **Drivers** is defined by:

- TimeWindows(d): the set of availability intervals of driver *d*, each included in [0, T[(in minutes).
- TimeCost(d): the cost per working time unit of driver *d* (in €/minute).
- MaxDrivingDuration(d): the maximum driving duration for driver *d*, before ending the shift at the base (in minutes).
- MinIntershiftDuration(d): the minimum time interval for driver d between 2 shifts (in minutes).
- Trailer(d): the only trailer which can be driven by driver *d*.

1.3.2.4 Trailers master data

The set of trailers is denoted as **Trailers**. By convention, indices referring to a trailer are denoted by *tl*. Each trailer *tl*∈**Trailers** is defined by:

1.3.2.5 Locations master data

A location denoted p (or alternatively q when the distance is calculated between two locations) may be a base, a source or a customer:

- **Bases** are the starting and ending locations of any shifts.
- **Sources** and **Customers** are loading and delivery locations respectively. For sources, quantities loaded to the trailer have a negative sign, while for customers delivered quantities have positive sign, as do their consumption forecasts.

Since all customers considered in this problem are VMI customers(forecastable customers), Air Liquide forecasts their tank levels and decides to deliver product and whenever necessary and cost efficient. By convention, the numbering of the locations p starts with the bases (e.g. 0, 1, 2...) then the sources (e.g. 3, 4, 5...) and eventually the customers. For this version of the problem, only one base(location 0) and a source(location 1) and then the customers are be present.

- **All locations(base, source and customers) common characteristics**
  - DistMatrix(p, q): distance between locations p and q (in km).
  - TimeMatrix(p, q): travel time from locations p to q (in minutes).
- **Source and customers characteristics**
  - SetupTime(p): the fixed part of loading/delivery time for point p (delivery for a customer or loading for a source) (in minutes).
- **Customers-only characteristics**
  - SafetyLevel(p): the level at the customer tank must always be above the safety level to avoid product shortage.
  - Forecast(p, h): the amount of product in mass (kg) that is used by the customer at location p during the timestep h.
  - InitialTankQuantity(p): the amount of product in mass (kg) available in the customers tank at the beginning of the horizon.
  - Capacity(p): the maximal amount of product in mass (kg) that can be delivered to a customer at location p.
  - AllowedTrailers(p): set of trailers allowed to supply the customer p.

1.3.3 Solution

A solution of the problem is a set of shifts(denoted by **Shifts**). The following decision variables are defined on each s∈**Shifts**:

- driver(s): the driver for shift s.
- trailer(s): the trailer (and associated tractor) for shift s.
- start(s): the starting time for the shift s (within [0, T[in minutes).

- Operations(s): it's a list of operations (loading, deliveries) performed during the shift s.

Each operation $o \in$ **Operations(s)** is defined by:

- arrival(o): the arrival time of operation o (within [0, T[in minutes).
- point(o): the location at which operation o takes place either sources or customers
- quantity(o): the quantity to be delivered or loaded in operation o. It is negative for loading operations at sources, positive for delivery operations at customers(in kg).

1.3.4 Constraints

1.3.4.1 Bounding constraints

All unary constraints on variables (e.g. non-negativity, inclusion in [0, T[, lower and upper bounds specified in previous section) must be satisfied.

1.3.4.2 Constraints related to drivers

[DRI01 | Inter-Shifts duration]
For each driver d, two consecutive shifts assigned to d must be separated by a duration of MinIntershiftDuration(d).

```
For all $d \in$ Drivers,
    For all $s_1$, $s_2 \in$ shifts(d),
        Start($s_2$) > end($s_1$) + MinIntershiftDuration(d) OR
        start($s_1$) > end($s_2$) + MinIntershiftDuration(d)
    EndFor
EndFor
```

[DRI03 | Respect of maximal driving time]
For each operation associated with a shift s (including the final operation performed at the base), the cumulated driving time is the total travel time on the shift up to the previous operation plus the travel time from the location of the previous operation to the location of the current operation.

```
For a given $s \in$ Shifts,
    For all operations $o \in$ Operations(s),
        If Operations(s) is not final(s)
            cumulatedDrivingTime(o) = cumulatedDrivingTime(prev(o) + timeMatrix(prev(o), o))
        else
            cumulatedDrivingTime(o) = cumulatedDrivingTime(prev(o) + timeMatrix(o, final(o)))
        EndIf
    EndFor
EndFor
```

Then, the constraint formulation is: for each operation, the cumulated driving time of the shift cannot exceed (by law) the maximum allowed driving time. While there are similar regulatory constraints on the driver's continuous working time from the beginning of the shift to the end, the cumulated driving time is generally the limiting constraint - and is therefore the only constraint considered in this simplified problem.

```
For all $s \in$ Shifts,
    For all operations $o \in$ Operations(s) and {final(s)},
        cumulatedDrivingTime(o) = cumulatedDrivingTime(prev(o) + timeMatrix(prev(o), o))
    EndFor
EndFor
```

[DRI08 | Time windows of the drivers]
For each shift s, the interval [start(s), end(s)] must fit in one of the time-windows of the selected driver.

```
For all $s \in$ Shifts
    It exists at least a $tw \in$ TimeWindows(Drivers(s)), $start(s) \geqslant start(tw)$ AND $end(tw) \geqslant end(s)$
EndFor
```

1.3.4.3 Constraints related to trailers

[TL01 | Different shifts of the same trailer cannot overlap in time]
Consider any two shifts s_1 and s_2 that use the same trailer. Then, either s_1 ends before the start of s_2 or s_2 ends before the start of s1.

```
For all $tl \in$ Trailers,
    For all $s_1$, $s_2 \in$ shifts(tl),
        $start(s_2) > end(s_1)$ OR $start(s_1) > end(s_2)$
    EndFor
EndFor
```

[TL03 | The trailer attached to a driver in a shift must be compatible]
For each shift s, the assigned trailer must be the trailer that can be driven by the driver.

```
For all $s \in$ Shifts,
    trailer(s) = Trailer(driver(s))
```

1.3.4.4 Constraints related to the sites

[DYN01 | Respect of tank capacity for each site]
For each site p, the tank quantity at each time step h must be contained into the interval [0, Capacity(p)]

```
For all $p \in$ Customers,
    For all $h \in [0, H[$
        $0 \leqslant$ tankQuantity(p, h) $\leqslant$ Capacity(p)
    EndFor
EndFor
```

For each customer p, the tank quantity at each time step h is equal to the tank quantity at time step h - 1, minus the forecasted consumption at h, plus all deluveries performed at h.
Let us remind that the values of Forecast(s, h) and $\sum_{h \in Operations(p,h)} quantity(o)$ are positive for customers. This allows writing the same basic inventory dynamic equation for sources and customers.

```
For all $p \in \{c \in Customers\}$
    tankQuantity(p, -1) = InitialTankQuantity(p)
EndFor
For all $h \in [0, H[$
    Given that dyn = tankQuantity(p, h-1) - Forecast(p, h) + $\sum_{h \in Operations(p,h)} quantity(o)$
    tankQuantity(p, h) = max(dyn, 0)
EndFor
```

Note: we assume that the entire quantity delivered in an operation is available in the customer tank as soon as the truck arrives at the customer (even if the trailer should stay a fixed time to complete a delivery, as explained in constraint SHI03).

1.3.4.5 Constraints related to shifts

[SHI02 | Arrival at a point requires traveling time from previous point]

```
For all $o \in$ Operations(s),
    For all shift s
        $arrival(o) \geqslant departure(prev(o)) + TimeMatrix(prev(o), o)$
    EndFor
EndFor
```

Since each shift ends at the base, we need to take into account the travel time from the last operation to the base:

```
For all shift s,
    $arrivals(s) \geqslant departure(last(Operations(s))) + TimeMatrix(last(Operations(s)), point(final(s)))$
```

Note: the waiting time of the drivers (at the gate of a source or a customer) is not defined as such in this model. As a consequence, there is no maximum to this waiting time. Therefore, an arbitrary long "idle time" (where the driver rests at the door of the customer or source doing nothing) can precede any operation belonging to a shift, as long as all of the constraints are respected.

[SHI03 | Loading and delivery operations take a constant time]

departure(o) = arrival(o) + SetupTime(point(o))

[SHI05 | Delivery operations require the customer to be accessible for the trailer]

```
For all $s \in$ Shifts, $o \in$ Operations(s):
    If $point(o) \in$ Customers
        $trailer(s) \in AllowedTrailers(point(o))$
    EndIf
EndFor
```

[SHI06 | trailerQuantity cannot be negative or exceed capacity of the trailer]

```
For a given shift $s \in$ Shifts,
    For all $o \in$ Operations(s) with {final(s)},
        $trailerQuantity(o) = trailerQuantity(prev(o))$ - $quantity(o)$
        $trailerQuantity(o) \geqslant 0$
        $trailerQuantity(o) \leqslant Capacity(trailer(s))$
    EndFor
EndFor
```

[SHI07 | Initial quantity of a trailer for a shift is the end quantity of the trailer following the previous shift]

endTrailerQuantity(s) = trailerQuantity(last(Operations(s)))

From these definitions we can derive this constraint:

```
If s = first(shifts(s))
    startTrailerQuantity(s) = InitialQuantity(trailers(s))
else
    startTrailerQuantity(s) = endTrailerQuantity(prev(s, shifts(trailer(s))))
EndIf
```

[SHI11 | Some product must be loaded or delivered]
For each source, some product (a negative quantity) must be loaded and for each customer, some product (a positive quantity) must be delivered.

```
For all $p \in$ Customers,
    For all $h \in$ [0, H-1],
        For all $o \in$ Operations(p, h)
            $quantity(o) \geqslant 0$
        EndFor
    EndFor
EndFor
For all $p \in$ Sources,
    For all $h \in$ [0, H-1],
        For all $o \in$ Operations(p, h)
            $quantity(o) < 0$
        EndFor
    EndFor
EndFor
```

1.3.4.6 Constraints related to quality of service

[QS02 | Run-out avoidance]
For each VMI customer p, the tank level must be maintained at a level greater than or equal to the safery level SafetyLevel(p), at all times.

```
For all $p \in$ Customers,
    For all $h \in$ [0, H-1]
        $SafetyLevel(p) \leqslant tankQuantity(p, h)$
    EndFor
EndFor
```

1.3.5 Optimization goal

1.3.5.1 Objective function

The goal is to minimize the distribution costs required to meet customer demands for product over the long term horizon. In order to tend to that final goal, we will minimize the **logistic ratio** The logistic ration is defined as the total cost of the shifts divided by the total quantity delivered on those shifts:

$$LR = \frac{\sum_{s \in shifts} Cost(s)}{TotalQuantity}$$

The cost of a shift represents the **distribution cost** related to the shift, including:

- the **distance cost**, applied to the total length of the shift, which is generally related to the trailer used (covering fuel consumption and maintenance)
- the **time cost** applies to the total duration of the shift which is generally related to the driver (covering the river salary and charges)

$$Cost(s) = (DistanceCost(trailer(s)) * TravelDist(s) + TimeCost(driver(s)) * (end(s) - start(s)))$$

Where:

$$travelDist(s) = \sum_{o \in Operations(s)} DistMatrix(prev(o), o) + DistMatrix(last(Operations(s)), point(final(s)))$$

The total quantity delivered over all shifts is computed as follows:

$$TotalQuantity = \sum_{s \in Shifts} \sum_{\substack{o \in Operations(s),\\ quantity(o) > 0}} quantity(o)$$

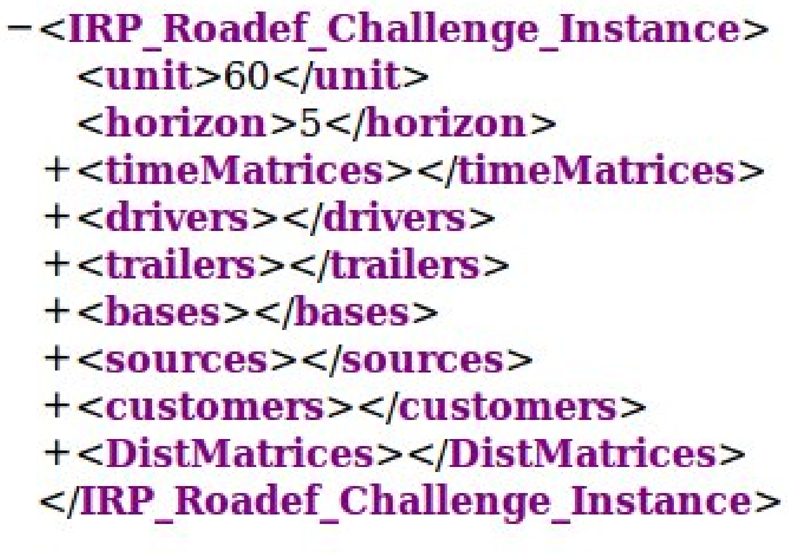
Note that if $TotalQuantity = 0$, we will consider also $LR = 0$

1.3.5.2 Time integrations over scheduling optimization horizons

The objective of an IRP problem is to minimize distribution costs over an enough long period (horizon) covering one or more replenishment cycles for all the customers.
If the time horizon is too short, the optimization can be short-sighted because of the "end-of-period side effect": customers that do not strictly require delivery within the horizon, which might increase the risk of shortage just beyond the horizon.
Considering a long period makes the relative impact of this effect negligible, but complexifies the problem and requires linger forecast on the customers, which could be far from the reality.
A "good" value fro the time horizon cannot be generalized as it depends on many factors, particularly the "confidence" in the customer forecast at various points over that horizon.3Cap

1.3.6 Instance representation

There are 11 instances of the Air Liquide IRP made available by ROADEF/EURO commission. The instances for the Roadef/Euro Challenge 2016 are in xml format. The xml lists in a hierarchical view all the variables as described in 1.3.2.



```
-<IRP_Roadef_Challenge_Instance>
   <unit>60</unit>
   <horizon>5</horizon>
  +<timeMatrices></timeMatrices>
  +<drivers></drivers>
  +<trailers></trailers>
  +<bases></bases>
  +<sources></sources>
  +<customers></customers>
  +<DistMatrices></DistMatrices>
  </IRP_Roadef_Challenge_Instance>
```

Figure 1.1: XML format of Air Liquide IRP instances

The instances available for the Roadef/Euro challenge 2016 are the following:

- Instance_V_1.1: this instance is inspired by a previous formulation of the problem.
  - 1 base.

  - 1 source, located at the base.
  - 12 customers, with daily linear forecast.
  - 2 drivers.
  - 2 trailers, each with capacity 23000 Kg.
  - horizon: 720

- Instance_V_1.2: This is a variant of the instance 1. The most noticeable differences are the capacities of the customers' tanks.
  - 1 base.
  - 1 source, located at the base.
  - 12 customers, with daily linear forecast.
  - 2 drivers.
  - 2 trailers, each with capacity 23000 Kg.
  - horizon: 720 hours.
- Instance_V_1.3: This instance is an excerpt of a real world problem, although very small. However, it enables to test many of the characteristics of the mathematical model and get prepared for harder instances.
  - 1 base.
  - 1 source.
  - 53 customers, with hourly variable forecast.
  - 1 driver.
  - 1 trailers, with capacity 22680 Kg.
  - horizon: 240 hours.
- Instance_V_1.4: It is a variant of the instance 3, with an additional driver and trailer.
  - 1 base.
  - 1 source.
  - 53 customers, with hourly variable forecast.
  - 2 drive
  - 2 trailers, with capacity 22680 Kg and 6500 KG.
  - horizon: 240 hours.
- Instance_V_1.5: This instance is an excerpt of a real problem. It features one trailer, with two drivers which are available 12 hours each in order to make it possible to run the trailer all the day. There is no superposition between drivers availabilities.
  - 1 base.
  - 1 source.
  - 54 customers, with hourly variable forecast.
  - 2 driver.
  - 1 trailers, with capacity 22840 Kg.
  - horizon: 240 hours
- Instance_V_1.6: It is a variant of the instance 5, but with an enlarged optimization horizon.
  - 1 base.
  - 1 source.
  - 54 customers, with hourly variable horizon.

  - 2 driver.
  - 1 trailers, with capacity 22840 Kg.
  - horizon: 840 hours.

- Instance_V_1.7.
  - 1 base.
  - 1 source.
  - 99 customers, with hourly variable forecast.
  - 2 driver.
  - 3 trailers, with capacity 9000 Kg, 22860 Kg and 13350 Kg.
  - horizon: 240 hours.
- Instance_V_1.8.
  - 1 base.
  - 1 source.
  - 99 customers, with hourly variable forecast.
  - 6 driver.
  - 3 trailers, with capacity 9000 Kg, 22860 Kg and 13350 Kg.
  - horizon: 240 hours.
- Instance_V_1.9.
  - 1 base.
  - 1 source.
  - 99 customers, with hourly variable forecast.
  - 6 driver.
  - 3 trailers, with capacity 9000 Kg, 22860 Kg and 13350 Kg.
  - horizon: 840 hours.
- Instance_V_1.10: this instance is an excerpt of a real problem. The dispatching region is more extended than the other test cases. There is only one driver per trailer, available 12 hours per day.
  - 1 base.
  - 1 source.
  - 89 customers, with hourly variable capacity.
  - 3 driver.
  - 3 trailers, with capacity 12620 Kg, 22940 Kg and 5380 Kg.
  - horizon: 240 hours.
- Instance_V_1.11: this instance is an excerpt of a real problem. The dispatching region is more extended than the other test cases. There is only one driver per trailer, available 12 hours per day.
  - 1 base.
  - 1 source.
  - 89 customers, with hourly variable capacity.
  - 3 driver.
  - 3 trailers, with capacity 12620 Kg, 22940 Kg and 5380 Kg.
  - horizon: 840 hours.

For all instances the triangular inequality holds. This means that, given customer p_i and p_j, and given the cheapest single-arc route (p_i, p_i), there is no route (p_i, p_k, p_j) that has cheaper cost. In practice, the cheapest way to go from p_i to p_j is $min_j\{TimeMatrix(i, j)\}$

It's interesting to see that some customers have enough initial product in their tanks to satisfy their demand. So no deliveries can virtually be performed for them since constraint **DYN01** is already satisfied. For simplicity, customers will be dived in two subgroups:

- **Demanding customers (DC)**: customers for which at least a delivery has be performed to satisfy constrain **DYN01**.
- **Non demanding customers (NDC)**: all customers not belonging to DM.

Notice that, even though it's not necessary, delivering product to non demanding customers can improve the quality of a solution. On one hand, the total amount of product delivered will increase with an improvement in the objective function, on the other hand the travel distance and the time distance will increase alongside with costs.

.

Chapter 2

Metaheuristics

2.1 Combinatorial problems

Combinatorial problems find several applications in the areas of computer science and applied sciences, where computational methods are used to find groupings or assignments of a finite set of objects that satisfies certain constraints.

2.1.1 Problems and solutions

Stützle and Hoos [1] make a distinction between problems and problem instances.

- A *problem* is general and abstract, for instance the Traveling Salesman Problem (TSP): "Given a set of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". The solution of this problem is an algorithm that, given a problem instance, determines a solution for that instance.
- An *instance* of the problem would be to find the shortest possible route for a specific set of cities. The solution of this problem instance would be a specific shortest route connecting the given set of cities.

There is also another important distinction between candidate solutions and solutions.

- *Candidate Solutions* are potential solutions that may be encountered during an attempt to solve the given problem instance and that don't satisfy all the conditions from the problem definition. For the TSP example, typically any valid route of any length connecting the given set of cities would be a candidate solution.
- *solutions* satisfies all constraints instead. In this case, only candidate routes with minimal length are considered solution

2.1.2 Decision problem

Many combinatorial problems are also considered *decision problems*: the solution of a given instance then will be specified by a set of logical conditions. It is important to distinguish between two variants:

- the *search variant* where, given a problem instance, the objective is to find a solution (or determine whether a solution exists).
- the *decision variant*, in which for a given problem instance, one wants to answer the question whether or not a solution exists.

Algorithms able to solve the search variant can always solve the decision variant. The converse also holds for many combinatorial problems.

2.1.3 Optimization problem

Many practical combinatorial problems are optimization problem rather than decision problems. *Optimization problems* can be seen as generalization of decision problems, where the solutions are additionally evaluated by an *objective function* and the goal is to find solutions with optimal objective function value.
Any combinatorial problem can be stated as *minimization problem* or *maximization problem.* A maximization problem can be translated to a minimazion problem and vice versa, hence they are equivalent. Many combinatorial problems are defined based on an objective function as well as on logical conditions. In this case:

- *feasible solutions* are candidate solutions satisfying all logical conditions.
- *optimal solutions* are, among candidate solutions, those ones with best objective function value.

2.2 Computational complexity

For a given algorithm, the complexity of a computation is characterized by the functional dependency between the size of an instance and the time and space required to solve this instance [1]. Instance size refers to the length of a reasonably concise description: for a TSP instance, its size corresponds to the number of cities to be visited or equivalently to the size of the underlying graph. The complexity of a problem can be defined as the complexity of the best algorithm used to solve this problem.

2.2.1 NP-hard and NP-complete problems

There are two relevant complexity classes:

- *class P*, the class of problems that can be solved by a *deterministic* algorithm in polynomial time
- *class NP*, the class of problems that can be solved by a *nondeterministic algorithm* in polynomial time.

P is included in NP, since a deterministic behavior can be simulated by a nondeterministic algorithm, but proving that also NP is included in P and then P = NP is one of great interest for the theoretical computer science community. This so called *P vs NP problem* in not only of theoretical interest, since many extremely-relevant problems are in NP and have huge problem size.
Many of these hard problems from NP can be translated into each other by a deterministic algorithm in polynomial time. A problem that is at least as hard as any other problem in NP (in the sense that can be polynomially reduced to it) is called NP-hard. NP-hard problems that are contained in NP are called *NP-complete.*
The TSP is known to be NP-hard [2] and, as we will see later, the Air Liquide IRP contains the TSP, hence the Air Liquide IRP is NP-hard.

2.3 Search paradigms

The fundamental idea behind the search approach is to iteratively generate and evaluate candidate solutions [**?**] In the case of combinatorial decision problems, evaluating a candidate solution means to decide whether it is an actual solution, while in the case of an optimization problem, it typically involves determining the respective objective function value.
The fundamental difference between search algorithms are in the way in which candidate solutions are generated, which can have a very significant impact on the algorithms' properties and performance. For this reason different search algorithms exist:

- *Perturbative search methods* can, given a solution composed of several *solution components*, modify one or more solution components to obtain a new candidate solution. Applied to the TSP, the perturbative search would start with one complete solution and, for example, exchange the order two cities are visited.

- *Constructive search methods* search in the space of *partial candidate solutions* instead. For the TSP a partial candidate solution would be a route that visits a subset of the cities once and terminates to the starting city. The task of generating complete candidate solutions by iteratively extending partial candidate solutions can be formulated as a search problem in which the goal is generate candidate solutions with a "good" objective value; the choise is led by heuristic and local information. For the TSP, a constructive search method would start at a randomly chosen city, and then iteratively add cities with minimal traveling distance between the current city and the one not visited yet, and terminate at the starting city.
- *Systematic search algorithms* explore the whole search space systematically and guarantees that an optimal solution will be found or, alternatively, will determine that no solution exists. This typical property of algorithms is called *completeness*. For the TSP, a systematic search would consider all permutations of the cities that have to be visited and considering the ending city as the starting one
- *Local search algorithms* start at some point of the search space and move from the current point to a neighboring point in the search space. Such moves are determined by a decision based on heuristic local knowledge only. Typically, local search algorithm are *incomplete*. In many cases a local search is perturbative based.

2.4 Stochastic local search

Stochastic local search algorithms (SLS) consist of search methods that make use of randomized choices to generate or select candidate solutions for a given combinatorial problem instance.

2.4.1 A general definition of stochastic local search

For a given instance of a combinatorial problem, the search for solutions takes place in the space of candidate solutions. The local search process is started by selecting an initial candidate solution, and then proceeds by iteratively moving from one candidate solution to a neighboring candidate solution. The decisions are based on heuristic local information but may also be randomized. [1]
A SLS is defined by Stützle and Hoos [1] in the following way:

Definition 1 (Stochastic Local Search algorithm). *Given a combinatorial problem* Π*, a stochastic local search algorithm for solving an arbitrary problem instance* $\pi \in \Pi$ *is defined by the following components:*

- *the* ***search space*** *$S(\pi)$ of instance π, which is a finite set of candidate solutions $s \in S$;*
- *a* ***set of (feasible) solutions*** *$S'(\pi) \subseteq S(\pi)$;*
- *a* ***neighborhood relation*** *on $S(\pi), N(\pi), \subseteq S(\pi) \times S(\pi)$;*
- *a finite* ***set of memory states*** *$M(\pi)$, which, in the case of SLS algorithms that do not use memory, may consist of a single state only;*
- *an* ***initialization function*** *$init(\pi) : \emptyset \rightarrow D(S(\pi) \times M(\pi))$, would specify a probability distribution over initial search positions and memory states;*
- *a* ***step function*** *$step(\pi) : S(\pi) \times M(\pi) \rightarrow D(S(\pi) \times M(\pi))$ mapping each search position and memory state onto a probability distribution over its neighboring search position and memory states;*
- *a* ***terminal predicate*** *$terminat(\pi) : S(\pi) \times M(\pi) \rightarrow D(\top, \bot)$ mapping each search position and memory state to a probability distribution over truth values which indicates the probability with which the search is to be terminated upon reaching a specific point in the search space and memory state.*

In the above, D(S) denotes the set of probability distributions over a given set S, where formally, a probability distribution $D \in D(S)$ is a function $D : S \rightarrow R_0^+$ that maps elements of S to their respective probabilities

A SLS algorithm realises a Markov process, since its behavior in a given state (s,m) does not depend on the search history.

The following is a general description of a SLS algorithm for a minimization problem, with f as objective function and $\hat{s}$ as best current solution:

```
procedure SLS-Minimization(π')
input problem instance $\pi' \in \Pi'$
output: solution $s \in S'(\pi')$ or $\emptyset$
$(s, m) := init(\pi', m))$;
$\hat{s} := s$
while not $terminate(\pi', s, m)$ do
    $(s, m) := step(\pi', s, m)$
    if $f(\pi', s) < f(\pi', \hat{s})$ then
        $\hat{s} := s$;
    end
end
if $\hat{s} \in S'(\pi')$ then
    return $\hat{s}$
else
    return $\emptyset$
end
end SLS-Minimization
```

2.4.2 Iterative Local Search

Given the first three components of a SLS algorithm, search space, solution set, and neighborhood relation, some *search strategies* can be defined by the definition of initialization and step functions. Ideally, search strategies should be independent from search space, solution set and neighborhood relation.
The definition of search steps and search trajectories is the following:

Definition 2 (**Search step, search trajectory**). *Let* Π *be a combinatorial problem, and let* $\pi \in \Pi$ *be an arbitrary instance of* Π. *Given an SLS algorithm A for* Π, *a search step (also called move) is a pair* $(s, s') \in S \times S$ *of neighborhood search positions such as the probability for A moving from s to s' is greater than 0, that is, N(s, s') and step(s)(s')* $>$ *0.*
A search trajectory is a finite sequence $(s_0, s_1, ..., s_k)$ *of search positions* $s_i (i = 0, ..., k)$ *such that for all* $i \in \{1, ..., k\}$, (s_{i-1}, s_i) *is a search step and the probability of initializing the search at* s_0 *is greater than zero, that is,* $init()(s_0, m) > 0$ *for some* $m \in M$.

A very simple ILS strategy is the *Uninformed Random Walk.* It does not use memory and is based on an initialization function that returns the uniform distribution over the entire search space. Its step function maps each point in S to an uniform distribution over all its neighborhood $N \subseteq S \times S$. Since it does not provide any mechanism to guide the search towards better solution, it's quite ineffective.

Definition 3 (**Evaluation function**). *Given an* $\pi \in \Pi$ *an instance of a decision problem* Π, *an evaluation function* $g(\pi)(s) : S(\pi) \mapsto \mathbb{R}$ *that maps each search position onto a real number in such a way that the global optima for* π *corresponds to the solutions of* π.

In most cases the evaluation function is problem specific, and for combinatorial optimization problems the objective function is used as evaluation function.
A SLS that uses the objective function to guide the search towards better solutions is known as Iterative Local Search (ILS).

2.4.3 Local minima and escape strategies

The definition of local minimum is the following:

Definition 4 (Local minimum, strict local minimum)**.** *Given a search space S, a solution set $S' \subseteq S$, a neighborhood relation $N \subseteq S \times S$ and an evaluation function $g : S \mapsto \mathbb{R}$, a local minimum is a candidate solution $s \in S$ such that for all $s' \in N(s), g(s) \leq g(s')$. We call a local minimum s a strict local minimum if for all $s' \in N(s), g(s) < g(s')$.*

Local minima are positions in the search space from which no search can achieve an improvement. Since local minima are not of sufficient high quality, some techniques for avoiding or escaping from local minimam have to be defined. There are two main ways to ecape from local minima:

- *Restart strategy*: the search is reinitialized every time a local minima is encountered.
- *Non-improving step*: allow non improving moves when a local minima is encountered.

This strategies are very effective, but a balance between randomization and greediness must be found. This trade-off is often called "*diversification* vs *intensification*":

- *diversification strategies* prevent search stagnation making sure that the search does not get stuck in confined regions that does not contain high quality solution.
- *intensification strategies* greedily improve solution quality or the chances to fin a solution in the near future.

2.4.4 SLS methods

ILS is the simplest SLS algorithm. It is quite effective but yet has several limitations. There are several ways to improve the ILS.
One is to use different *pivoting rules*:

- **Best improvement** randomly selects at each search step one of the neighboring solutions that achieves a maximal improvement in the evaluation function.
- **First improvement** select the first neighboring solution encountered that achieves an improvement in the objective function The first improvement avoids evaluating all neighbors to reduce time complexity, but the order in which they are evaluate also affects the efficiency of the search.

Considering several neighborhood relations can also improve the performance of a SLS. A solution that is optimal w.r.t a neighborhood relation N_1 may not be optimal for a different neighborhood relation N_1.
Variable Neighborhood Descent (VND) considers k neighborhood relations $N_1, N_2, ..., N_k$ ordered according to increasing size. The algorithm starts the search with neighborhood N_1 until a local minimum is reached. Whenever no more improvement steps are possible for a neighborhood N_i, the VND continues the search in N_{i+1}.
The ILS always always accepts, according to its pivoting rule, a neighboring solution with a better evaluation function value. This is the simplest case of **acceptance criterion** but other exists:

- *metropolis acceptance*: accept a new neighboring solution with probability:

$$p(T, s, s') = \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ e^{\frac{f(s)-f(s')}{T}} & \text{if } f(s') > f(s) \end{cases}$$

- *simulated annealing acceptance*: a variant of metropolis acceptance with variable temperature, using the following parameters:
  - initial temperature.
  - final temperature.
  - step: the increment in temperature.
  - frequency: how many search steps are performed before updating the temperature .

There are also different *termination criteria*:

- *local minimum termination*: the search ends when a local minimum in encountered
- *maximum steps*: the search ends when a maximum number of steps

2.5 Initial solution

The first step of a SLS algorithm is to generate an initial solution. There are two main way to generate an initial solution:

- **constructive heuristics**: the initial solution is built using a greedy function that iteratively add solution components to a partial solution. A greedy function is used to chose the next component or a random component is added with a certain probability.
- **relaxation**: a mathematical model is built for the initial problem. Some constraints are deleted to relax the problem and exact methods are used to generate a feasible initial solution for the relaxed problem. The solution generate will be unfeasible for the initial problem.

In general, the way an initial solution is generated and its solution quality can significantly affect the performance of the SLS.

2.5.1 Initial solutions for IRP in literature

Goel, Furman, Song and El-Bakry [7] propose a constructive heuristic to generate an initial solution for a LNS: the construction heuristic has a greedy approach that tries to deliver as much gas as possible in the shortest time possible, while minimizing the lost production and stockouts by prioritizing the most urgent demand. The heuristic calculates the closing inventory level at terminal k for time period t, that is used to identity the set of shifts $\mathrm{V_k}$ that can load/unload and leave terminal k at time t. The heuristic then identifies, as destination terminal, the terminal where at least one of the ships in $\mathrm{V_k}$ is allowed to load/unload and that has the most urgent need. The heuristic selects the ship with largest loading/unloading capacity in $\mathrm{V_k}$. This approach is repeated until no more departures can be scheduled or all terminals' demand is satisfied.

Aksen, Kaya, Salman and Tuncel [9] use exact methods and local search to generate a solution for their Adaptive LNS. A good quality initial solution is found solving a relaxed version of the problem, then some of the following route improvement heuristics are applied to improve the solution quality:

- **Intraroute 2-Opt**: two edges are removed from a tour and the two segments are reconnected.
- **Intraroute 3-Opt**: three edges are removed from a tour and the three segments are reconnected in every possible way.
- **Interroute 2-Opt**: two edges from different routes are removed and replaced by new edges.
- **Interroute 1-0 move**: a node is moved from a route to another.
- **Interroute 1-1 and 2-2 exchange**: two nodes or two pair of nodes from different routes are exchanged.
- **Interroute 1-1-1 rotation**: considering three routes, a node from each route is shifted to the following route in a cyclic way.

2.5.2 Initial solutions proposed for the Air Liquide IRP

Three different constructive heuristics are proposed for the Air Liquide IRP:

- **Greedy**: constructs an initial solution using heuristics.
- **Greedy Randomized**: constructs several initial solutions using heuristics and choose one of them according to their objective function value.
- **Partial Greedy Randomized**: constructs a solution component by component. It builds several candidate components using heuristics and add one of them according to the improvement in the objective function value, until a complete solution is built.

All of them use two functions:

- *constructSolution*: this function generates a solution where product is delivered only to customers belonging to DC. This is the strict minimum necessary in order to satisfy constraint **QS02**.

- *extendSolution*: this function extends the solution adding more shifts and delivering the product to any customer.

constructSolution has the following parameters:

- *initialSolution*
- *timeWeight (w_t)*: controls the importance of urgency time in the greedy function to select the next customer to be served.
- *quantityWeight (w_t)*: controls the importance of product demand of customers in the greedy function to select the next customer to be served.
- *ties*: used to break ties in the greedy function.
- *servingRatio*: controls the quantity of product delivered to a customer.
- *refuelRatio*: determines when a vehicle has to go back to the source to refuel.
- *randomPick*: used the control the degree of randomness introduced in the algorithm.
- *urgencyPolicy*: used to chose which greedy function to use.
- *maxShifts*: the solution will contain at most *maxShifts* shifts.

constructSolution accepts a solution that may be empty or partial, in the sense that some customers' demand may be no satisfied yet, and builds a solution with the following steps:

- the function computes the current state of a solution, consisting of the following variables:
  - time windows still available
  - slack capacity of trailers
  - tank level at customers' sites

  all other variables such as the total demand of product left for the remaining horizon, can be derived from these.
- two greedy functions can be used to compute the urgency for every customer. The first greedy function considers a long term profit, since it considers the total amount of demand over the whole horizon.

$$1.0 + \frac{w_{\mathrm{t}} * runouTime + w_{\mathrm{q}} * (1.0 - |totalDeman - totalDeliveredProduct|)}{2}$$

  - *horizon*: the horizon in hours.
  - *runoutTime*: time at which the tank is expected to go below the minimum capacity. It depends on the tank level at customer's site.
  - *totalDemand*: total demand of product of a customer. It depends on the customer's forecast.
  - *totalDeriveredProduct*: the total amount of product delivered to a customer in the current solution. It depends on the tank level at customers' sites and on the forecast.

  The second greedy function considers a short term profit, since it considers the cost of the next delivery.

$$1.0 + \frac{w_{\mathrm{t}} * runouTime + w_{\mathrm{q}} * Cost(p, p')}{2}$$

$$Cost(p, p') = (DistanceCost(trailer(s)) * TravelDist(p, p') + TimeCost(driver(s)) * timeDist(p, p'))$$

  - *TravelDist(p, p')*: the travel distance (in Km) required to go from customer p to customer *p'*.
  - *timeDist(p, p')*: the time distance (in min) to go from customer p to customer *p'*.

Ties are broken favoring on of the two terms.
To reduce computation, a matrix that contains all distance and time costs for going from customer p to p' is computed offline and accessed every time the greedy function needs to be recomputed. Since costs are different, a matrix for each pair (driver, trailer) is computed. The maximum values is assigned where $p = p'$.
Customers are then inserted in a priority list alongside with their greedy value and sorted in an ascending order (from best to worse). Base and source sites are not considered.

- The first customer according to the greedy function used is then the next candidate to be inserted, unless one of the following hold:
  - serving the customer would violate constraints **DRI01**, **DRI03**, **DRI08**, **TL01**, **SHI05** (see 1.3.4).
  - trailer needs to go back to source to refuel.
  - total product demand for the customer is already satisfied

  If the current customer cannot be inserted, it's discarded and the next one is considered.

- once a customer has been chosen, state variables are updated. In particular, the quantity of product to serve is calculated in the following way:
  - constraints **DYN01**, **SHI11** are satisfied, meaning that it's not possible to deliver more product than the quantity available in the trailer and the quantity of product delivered will not bring tank level above the maximum capacity. For the source there is no restriction on site capacity (see 1.3.4).
  - in case the quantity of product is bigger than the total demand of product left for the customer, only the quantity need to satisfy the demand will be delivered.
  - the quantity of product delivered is then determined according to the parameter *servingRatio*. For *servingRatio* = 1 the OU policy will be used, meaning that the maximum amount of product possible is served, otherwise the ML policy will be used,meaning that only a fraction of the total amount of product is served.

  When no deliveries can no longer be scheduled in a time window, the next time window will be used.

The function iterates until all customers' demands are satisfied or there are no time windows available. The function returns a solution can, in the best case, satisfy all customers' demand.

extendSolution has the following parameters:

- *servingRatio*: controls the quantity of product delivered to a customer.
- *refuelRatio*: determines when a vehicle has to go back to the source to refuel.
- *maxShifts*: the solution will contain at most *maxShifts* shifts.

extendSolution accepts a solution that may be empty or partial, in the sense that some customers' demand may not be satisfied yet and "extends" the solution with more deliveries, regardless their demand of product. In practice, it accepts the solution returned by the first function that is already supposed to satisfy all customer's demands and add more operations to improve the objective function value with the following steps:

- the function computes the current state of a solution, consisting of the following variables:
  - time windows still available
  - slack capacity of trailers
  - tank level at customers' sites

  all other variables such as the total demand of product left for the remaining horizon, can be derived from these.

- a greedy function is to compute the urgency for every customer:

$$\frac{DistanceCost(trailer(s)) * TravelDist(p, p') + TimeCost(driver(s)) * TimeDist(p, p')}{tankQuntity(p') - Capacity(p')}$$

  this greedy function considers a short term profit, given by time and distance costs of serving customer *p'* when the current location is *p* and the slack capacity of site *p'* when the trailer is suppose to reach the site. Notice that now, since there is no longer demand of gas, tank levels will not lower. Once full, the customer can no longer be served. If the denominator is equal to zero, the maximum value will be assigned to the function. These terms are very close to the terms in the objective function in order to optimize costs and total amount of product delivered. Customers are then inserted in a priority list alongside with their greedy value and sorted in an ascending order (from best to worse). Base and source sites will not be considered.

- The first customer according to the greedy function is then the next candidate to be inserted, unless one of the following hold:
  - serving the customer would violate constraints **DRI01**, **DRI03**, **DRI08**, **TL01**, **SHI05** (see 1.3.4).
  - trailer needs to go back to source to refuel.
  - site's tank is full

  If the current customer cannot be inserted, it's discarded and the next one is considered.

- once a customer has been chosen, state variables are updated. In particular, the quantity of product to serve is calculated in the following way:
  - constraints **DYN01**, **SHI11** are satisfied, meaning that it's not possible to deliver more product than the quantity available in the trailer and the quantity of product delivered will not bring tank level above the maximum capacity. For the source there is no restriction on site capacity (see 1.3.4).
  - the quantity of product delivered is then determined according to the parameter *servingRatio.*

  When no deliveries can no longer be scheduled in a time window, the next time window will be used.

The process iterates until there are no time windows available.

All the three constructive heuristics use, in sequence, the *constructSolution* function to generate a (possibly) feasible solution (that satisfy all customer's demands) and the second*extendSolution* function to extend the solution and improve even more it's objective function value.
Greedy builds solutions trying different set of values (*timeWeight*, *quantityWeight*, *ties*) and returns it as a feasible solution is found, otherwise it returns an unfeasible solution.

```
For all $w_t \in$ {0, 0.1, ... 1}
    For all $w_q \in$ {0, 0.1, ... 1}
        For all $t \in$ {-1, 1}
                s = NULL
                s = greedyInitialSolution(s, $w_t$, $w_q$, t)
                s = extendSolution(s)
                If(objectiveValue(s) < 1)
                    return s
        EndFor
    EndFor
EndFor
```

Greedy Randomized builds solutions for every set of values (*timeWeight*, *quantityWeight*, *ties*) and keep them in a list alongside with their objective function values. A vector of probabilities is build according the objective function values and a solution is randomly returned, the solution with worst objective function value being more likely to be selected. In fact, this heuristic favors unfeasible solutions. This heuristic is propose because starting the search only from feasible solution may be too restrictive and search in a too little area of the search space, getting stuck in the same local minima.

```
candidateList = $\emptyset$
For all $w_t \in$ {0, 0.1, ... 1}
    For all $w_q \in$ {0, 0.1, ... 1}
        For all t $\in$ {-1, 1}
            s = NULL
            s = greedyInitialSolution(s, $w_t$, $w_q$, t)
            s = extendSolution(s)
            obj <- evaluateSolution(s)
            candidateList = append(candidateList, (s, obj))
        EndFor
    EndFor
EndFor
probVector <- computeProbabilities(candidateList)
r <- rand()
index <- 0
While(r > probVector[index] and index < size(probVector - 1)) Do
    index++
EndWhile return(candidateList[index])
```

Partial greedy randomized builds solutions for every set of values (*timeWeight*, *quantityWeight*, *ties*) adding only one shift at time. These partial solutions are kept in a list alongside with their new objective function values due to the new shift inserted. A vector of probabilities is build according the objective function values and a solution is randomly for the next step. Once a complete solution has been built or no time windows are available, the solution is returned.

```
candidateList = $\emptyset$
partialSolution = NULL
masShifts = 1
While maxShifts < size(timeWindows) Do
    For all $w_t \in$ {0, 0.1, ... 1}
        For all $w_q \in$ {0, 0.1, ... 1}
            For all t $\in$ {-1, 1}
                s = NULL
                s = greedyInitialSolution(partialSolution, $w_t$, $w_q$, t, masShifts)
                obj <- evaluateSolution(s)
                candidateList = append(candidateList, (s, obj))
                probVector <- computeProbabilities(candidateList)
            EndFor
        EndFor
    EndFor
    r <- rand()
    index <- 0
    While(r > probVector[index] and index < size(probVector - 1))
        index++
    EndWhile
    partialSolution <- candidateList[index]
    maxShifts <- maxShifts + 1
EndWhile
```

2.6 Neighborhood definition

The definition of a neighborhood relation can affect the performance of the SLS as well. There are standard and well known neighborhood definition in literature that can be applied to many different problems, but often the neighborhood definition is problem specific. A neighborhood can be represented by a *Neighborhood graph* G_n: given vertex representing a solution *s*, there is an edge reaching another vertex representing a solution *s'* if and only if the relation (*s*, *s'*) ∈ N. Some of its properties are:

- most neighborhood relations are symmetric: $\forall s, s' \in S : (N(s,s') \Leftrightarrow N(s',s))$, meaning that the neighborhood graph is undirected. This is property is needed by a SLS to directly reverse the search steps.
- given a solution *s* and its vertex in the neighborhood graph, the degree of the vertex corresponds to the neighborhood size of *s*.
- The diamater of the neighborhood graph gives a worst-case lower bound on the number of steps needed to reach an optimal solution from an arbitrary point of the search space.

2.6.1 Neighborhood definition in literature

Paul Shaw [6] proposes a Large Neighborhood Search that uses a tree search to evaluate the cost of a move, select a move and check feasibility. Using a relatedness function, a set of visits are removed and a branch and bound technique combined with constraint propagation and branching heuristics is used to find the minimum cost re-insertion.
A relatedness function is defined to select visits that are highly related. In this way visits that are geographically close and that make use of the same vehicle are more likely to be chosen. The start number of visits to be removed is 1 and, after *a* unsuccessful attempts to improve the cost, it is increased.
The re-insertion process uses a branch and bound technique that in the simplest case examines all the search tree. Additional techniques are used to make the process faster:

- **Constraint propagation**: for a visit, some insertion points may be classified as illegal. There is a propagation rule for loads and for serving time.
- **Branching heuristic**: for a visit *v*, a set of insertion points $I_v = \{p_1,...,p_n\}$ are defined with the associated increase of cost C_v of the routing plan due to the insertion of the visit at that point. Then the cheapest insertion point $C_v \in I_v$ is defined for *v*. The farthest insertion heuristic is then used, choosing visit *v* to be inserted for which C_v is maximized and trying to insert it in each of its positions from cheapest to most expensive.
  The LNS that is proposed is computationally more expensive than other local search methods, so less moves for second can be evaluated. However, these moves are more powerful and move the search further at each step.

Goel, Furman, Song and El-Bakry [7] propose a three-step Large Neighborhood Search (LNS) consisting on:

- **Construction heuristic**: described in 2.4
- **Time window improvement heuristic**: the time-window improvement heuristic defines the search neighborhood as the set of feasible solutions in which a ship departure is delayed or advanced by at most *m* days.
- **Two-ship improvement heuristic**: The two-ship improvement heuristic selects a pair of ships and define the search neighborhood as the set of feasible solutions where the pairs of ships is rescheduled. All the other ships' schedules are fixed while a sub problem is solved for the ship pair rescheduling. Ship-pair selection sequence can affect the heuristic's performance. Two different schemes are proposed:

– **Lexicographic selection**: ships are ordered and ship pairs are selected in the following order: (s_1, s_2), (s_1, s_3), ..., (s_{n-1}, s_n).
– **Metric based selection**: this metric estimates the potential reduction of objective function if the ship pair (s_i, s_j) is rescheduled. The improvement in the objective function π_{v_i} due to the reschedule of ship v_i is estimated as the total production and stockouts that would be eliminated if the re-optimized schedule would not involve any demurrage. The objective reduction $\pi_{v_{i,j}}$ due to the rescheduling of (s_i, s_j) is then estimated as the greatest of π_{v_i} and π_{v_j}

Ropke and Pisinger [8] propose a LNS similar to the one proposed by Paul Shaw [6]. A parameter $q \in [0, ..., n]$ determines the number of requests removed and reinserted, hence the neighborhood size.
It has several differences compared to the LNS proposed by Shaw:

- Shaw propose only one removal and reinsertion heuristic, while here several are used and the selection mechanism uses statistics gathered during the search.
- Shaw propose an optimal method, the branch and bound, for the reinsertion, while here much simpler heuristics are used.

Notice that a parameter p introduces a degree of randomization for the heuristics. The proposed removal heuristics are the following:

- **Shaw removal heuristic**: a relatedness function is used to identify similar solutions that will more likely lead to a better solution. This function considers travel distance, time distance, vehicle capacity and if two requests involve the same vehicle.
- **Random removal**: q requests are selected randomly and removed.
- **Worst removal**: the cost of a request is computed as $cost(i, s) = f(s) - f_{\text{-i}}(s)$ where $\text{f}_{\text{-i}}(\text{s})$ is the objective value of the initial solution without request i. Then the request with highest *cost(i, s)* is removed.

The proposed reinserton heristics are the following:

- **Basic greedy heuristic**: Let $\Delta f_{\text{i, k}}$ be the cost of inserting request i at position k. Then let $c_i = min_{\text{k}} \Delta f_{\text{i, k}}$ be the cost of inserting request i at the minimum cost position. This heuristic chooses the request with lowest c_i and insert it at the best position. In this way, given that the request $i \in \{1, ..., n\}$ previously at position k is reinsert at position h, the objective value of the new function can be computed as $f - \Delta f_{\text{i,k}} + \Delta f_{\text{i,h}}$ without having to recompute the cost of each other route. This way the complexity is O(1) instead of O(n). For the Air Liquide IRP a similar technique will be used to reduce computational complexity.
- **Regret heuristic**: this heuristic implements a look-ahead mechanism to select the request to insert. Let $x_{\text{i k}} \in 1, ..., m$ be a variable that indicates the route for which request i has the kth lowest insertion cost, then $c_{\text{i}}^{*} = \Delta f_{\text{i, x}_{\text{i2}}} - \Delta f_{\text{i, x}_{\text{i1}}}$. This means that the regret value is the difference in the cost of inserting the request in its best route and its second best route. This heuristic then chooses the request with maximal regret value.

Aksen, Kaya, Salman and Tuncel [9] propose an Adaptive LNS that consists of 3 main steps:

- destroy move
- repair move
- adaptive weight adjustment

The move to be performed is selected with the same roulette-wheel mechanism used by Ropke and Pisinger [8]

Moves of the algorithm: Several moves are performed $\rho \in [1, 2, 3]$ times with probability *prob*(ρ) = $\{5/9, 3/9, 1/9\}$. The nodes chosen for a move are picked up from 4 different sets:

- *Subset 1*: nodes not included in the solution (not visited).

- *Subset 2*: nodes included once.
- *Subset 3*: nodes included at least once in the solution but not in each period.
- *Subset 4*: nodes included in all periods.

Repair moves: two common steps of repair moves are:

- **Route improvement step (RIS)**: if the total amount collected in the period t is lower than the total vehicles capacity with one vehicle less, then the move tries to reduce the number of vehicles used in the solution by 1.
- **Source node insertion step (SNIS)**: if there is a vehicle with enough slack capacity to collect the amount of a node i in the period t, then dispatch the vehicle to serve node i at the best position in time period t.

Five repair moves are proposed:

- **Repair 1**: used when a node i belonging to subset 3 or 4 is removed from period t (and appears in period t'). The amount collected by i in period t is moved to period t'. Nodes and routes in t are updated. Since the total amount collected in t decreased, RIS is applied to period t.
- **Repair 2**: used when a node i belonging to subset 3 or 4 is inserted in a period t. The new amount collected brought by the insertion of i in period t is taken from a period $t' > t$. Since the total amount collected increased at period t and decreased at t', SNIS is applied to period t and RIS is applied to period t'.
- **Repair 3**: used when a node i belonging to subset 2, 3 o 4 is deleted from all periods. Nodes and routes are updated for those periods t in which node i was present. Then RIS is applied.
- **Repair 4**: used when a node i belonging to subset 1 is inserted in a period t. Then SNIS is applied for period t.
- **Repair 5**: used when all nodes in period t and period t' are exchanged. Periods t and t' are reconstructed from scratch with the new nodes and routes. Then all other periods are reconstructed in the same way.

Destroy moves: eleven destroy moves are proposed:

- **Randomly remove ρ visits**: randomly remove ρ nodes belonging subset 3 or 4 from a random period t.
- **Randomly insert ρ visits**: randomly insert ρ nodes belonging to subset 2 or 3 in a random period t. Nodes are inserted using Repair 2.
- **Remove the worst source node**: consider period t. Compute the difference in objective value obtained by removing a node i, then remove the most cost-effective node. After that, apply Repair 3.
- **Insert the best source node**: select a node belonging to subset 1. Evaluate the difference in objective value obtained by inserting node i in a position, then insert node i in the position that will improve the most the objective value.
- **Shaw removal**: randomly select a period t and a node i contained in period t. Considering the distance from node i to the closest node $dist_{min}$, remove all nodes that belongs to subset 3 and 4, located within the range [0, $dist_{min}$] and present in t.
- **Shaw insertion**: randomly select a period t and a node i contained in period t. Considering the distance from node i to the closest node $dist_{min}$, insert all nodes that belongs to subset 2 and 3, located within the range [0, $dist_{min}$] and not present in t.
- **Remove ρ source nodes**: randomly select ρ times a node belonging to subset 2,3 or 4 ad remove it from all periods. Then apply Repair 3.
- **Insert ρ source nodes**: randomly select ρ times a node belonging to subset 1 ad insert it in a random period t. Then apply Repair 4.
- **Empty one**: randomly select a period t and remove all nodes in $\mathbf{t}$. For nodes belonging to subset 2 apply Repair 3,for all nodes belonging to subset 3 and 4 apply Repair 1

- **Swap routes**: randomly select two periods t and t' and swap their nodes. Then apply Repair 5.
- **Randomly move ρ visits**: randomly select a period t and a node i belonging to subset 2 or 3 and present in t. Remove node i from period t and insert in a different random period t'. Then apply Repair 2 on period t' and Repair 1 on period$t$

2.6.2 Neighborhood definition for the Air Liquide IRP

A solution is represented as a list of shifts, with each shift represented as a list of operations, and neighborhood relations has been defined accordingly to this representation. Several neighborhood operators has been defined for the Air Liquide IRP:

- **exchange operator**: two operations belonging to the same shift are exchanged.
- **insert operator**: a new operation is inserted in a shift at any position.
- **remove operator**: an operation is removed from a shift.
- **refuel operator**: a different inventory policy (see 1.1) is used, meaning that the trailer will go back to the source when below a certain level and a fraction of the trailer capacity will be delivered to customers.

All the operators provides the following functions:

- a *begin* function that initializes all parameters
- a *computeStep* function that applies the operator (perform the move):
  - update the parameters
  - apply the operator at the current position of the search. Only the current shift will be modified
  - delete all following shifts.
  - rebuild the following shifts with *constructSolution*
  - extend the solution with *extendSolution*
  - evaluate the neighboring solution using the delta evaluation function
- a *random* function that performs a perturbation applying the operator at a random position
- a *reset* function that resets all parameters

The exchange operator applies to two contiguous operations belonging to the same shift and exchanges their position. The previous operations in the shift remain the same, while for the following operations may happen that:

- the two operations can be inserted in an inverted order: all the following will be inserted in the same order. If, at a certain point, one of the operations cannot be inserted, this operation with all the following will be discarded.
- the first or the second operation cannot be inserted due to constraints violation: the exchange is not performed and the shift will remain the same.

This operator has the following properties:

- *Commutativity*:

$$Exchange(s,i,j) = Exchange(s,j,i), \quad i,j \in Operations(shift), shift \in Shifts(s)$$

- it's *Symmetry*:

$$Exchange(Exchange(s,i,j),i,j) = s, \quad i,j \in Operations(shift), shift \in Shifts(s)$$

The insert operator inserts an operation in any shift, at any position. The previous operations in the shift remain the same, while for the following operations may happen that:

- the new operation can be inserted: all the following operations will be reinserted in the same order. If, at a certain point, one of the operations cannot be inserted, this operation with all the following will be discarded.
- the new operation cannot be inserted due to constraints violation: the operation will not be inserted and shift will remain unchanged.
- the new operation is inserted before or after an operation that involves the same customer: the insertion is not performed and the shift will remain the same.

This operator has the following properties:

- *Idempotence*:

$$Insert(Insert(s,i),i) = Insert(s,i), \quad i \in \{1,...,|Operations(shift)|\}, shift \in Shifts(s)$$

Given customers *p* and *p'*, the operation inserted between them will involve the customer *p'* such that the traveling time for the route (*p*, *p'*, *p"*) is minimized. A random factor allows random customers to be inserted with a certain probability.

The remove operator removes an operation from any shift, at any position. The previous operations in the shift remain the same, while for the following operations may happen that:

- the operations is removed: all the following operations will be reinserted in the same order. If, at a certain point, one of the operations cannot be inserted, this operation with all the following will be discarded.

The refuel operator rebuild a solution, starting from any shift, using a certain inventory policy. The previous shifts the same, while all the following will be rebuilt with the *constructSolution* using the new inventory policy: This operator has the following properties:

- *Idempotence*:

$$Refuel(Refuel(s, shift, sr, rr), shift, sr, rr) = Refuel(s, shift, sr, rr)$$

Exchange, insert and remove operators have the following parameters:

- *randomFactor*: used to control the degree of randomness introduced in the algorithm.
- *urgencyPolicy*: used to choose which greedy function to use.
- *refuelFactor*: determines when a vehicle has to go back to the source to refuel.
- *deliveredQuantityFactor*: controls the quantity of product delivered to a customer.

Refuel operator has the following parameters:

- *randomFactor*: used to control the degree of randomness introduced in the algorithm.
- *urgencyPolicy*:used to choose which greedy function to use.
- *refuelStep*: determines the step for the parameter *refuelFactor* during the search.
- *deliveredQuantityStep*: determines the step for the parameter *deliveredQuantityFactor* during the search.

2.7 Objective function and Delta evaluation

The objective function needs to be computed at each step of the search, this means that it may be computationally expensive and that more efficient ways to compute it can speed up considerably the search.

2.7.1 Delta evaluation function

Considering a solution *s* and one of its neighboring solutions *s'* obtained with a perturbation, a *Delta evaluation function* allows to calculate the difference in objective function between *s* and *s'*. This way the new objective function value does not have to be recalculated from scratch. Most of the times only one or few components of the solution are affected by the perturbation, so the delta evaluation function works only on these.

$$f(s') = f(s) + \Delta(s, s')$$
$$\Delta(s, s') = f_c(s') - f_c(s)$$

2.7.2 Delta evaluation function for the Air Liquide IRP

The objective function has already been shown in 2.7. Considering n the number of components of a solution, in this case shifts, the complexity of the objective function is O(n).
A delta evaluation function has been defined for the Air Liquide IRP. Has been noticed that, for this kind of problem, even a small change in the solution can strongly affects it. To exploit the delta evaluation function, solutions has been built in a way that the following holds: given a solution s and a component $c \in \{1, ...n\}$, a change in the component c will only affect the shifts $\{s_{\mathrm{c}+1}, ..., s_{\mathrm{n}}\}$ while the previous ones will remain the same.
This way the delta evaluation function can be defined as:

$$\Delta(s', s) = \sum_{k=c}^{n} f_k(s') - \sum_{k=c}^{n} f_k(s)$$

When a solution is evaluated for the first time, for example at the beginning of each ILS step, for each shift s the following information is kept:

- the total quantity of product delivered until shift s
- the total cost until shift s, given by:
  - the total distance traveled by all trailers until shift s
  - the total time distance traveled by all drivers until shift s

With this information, the state of a solution at each component can be computed. The objective function for the perturbed solution *s'* can be rewritten as:

$$LR(s') = \frac{\sum_{k \in Shifts(s')} Cost(k)}{TotalQuantity} = \frac{\sum_{k \in \{1,...c-1\} \subseteq Shifts(s)} Cost(k)}{TotalQuantity} + \frac{\sum_{k \in \{c,...n\} \subseteq Shifts(s')} Cost(k)}{TotalQuantity}$$

$$LR(s') = \frac{\sum_{k \in \{1,...c-1\}} Cost(k)}{oldQuantity} + \Delta(s', s)$$

- The first summatory, that consider the shift unchanged, can be easily computed in O(1) since the state of the solution until that component is known.
- the delta evaluation function can be computed in $O(n - c), c \in \{0, n - 1\}$.

At the end, calculating the objective function of the solution *s'* is:

$$O(1) + O(n - c) \simeq O(n - c)$$

- Worst case scenario, the function is completely perturbed: c = 0

$$O(n)$$

- Average case scenario: $c = \frac{1}{2}n$

$$O(n - \frac{1}{2}n) = O(\frac{1}{2}n) = \frac{1}{2}O(n)$$

- Best case scenario, only the last component changed: c = n-1

$$O(n - (n - 1)) = O(1)$$

2.7.3 Penalty for unfeasible solutions

Notice that the objective function does not consider the feasibility of a solution. This means that an unfeasible solution can have a better objective function value than a feasible solution, and it is actually often the case since feasible solutions are more "constrained". To guide the search towards feasible solutions areas, a penalty factor for unfeasibility has been introduced.

Since solutions are generated in a way that all the constraints but **[QSO2]** are satisfied, they are not checked to reduce computation cost.
Consider runout_{p} the time at which customer p is running out of product, the penaly factor will be calculated as:

$$penalty(s) = \frac{min_{\text{p} \in \text{Customers}} runouts(\text{p})}{horizon}$$

$$f_{\text{penalty}}(s) = 1.0 + \frac{f(s)}{penalty(s)}$$

The penalty will depend on how soon one of the customers is running out of product. This way, if the search starts from an unfeasible solution, it will try to push the run out as later as possible, possibly reaching a feasible solution. Given two different unfeasible solutions, one cannot exactly state which one is more promising. The first one might be a very high quality solution (in terms of quantity of product delivered and travel time) but does not satisfy constraint **[QS02]** at the very first shifts (due to distance time), while the other one is lower quality but breaks the constraint only at the last shifts.
An unfeasible solution will always have a worse objective function value than a feasible solution. Still, an unfeasible solution might be more promising since it could lead to an unexplored and very promising area of the search space.

Chapter 3

Automatic Configuration and Parameter Tuning

3.1 Automatic configuration and parameters tuning in literature

Ropke and Pisinger [8] propose a roulette wheel selection mechanism to dynamically chose what heuristics to use during the search phase. All the heuristics proposed are used. To select one of them, weights are assigned with a roulette wheel selection principle: given k heuristics with weight $w_{\mathrm{j}} \in \{1, 2, ..., k\}$, a heuristic is chosen with probability:

$$\frac{w_{\mathrm{j}}}{\sum_{i=1}^{k} w_{\mathrm{i}}}$$

Insertion heuristic is selected independently of the remove heuristic. The weights are set to zero and are increased by different value σ_1, σ_2 or σ_3 according their performance:

- σ_1: the last remove-insert operation led to a new global best solution.
- σ_2: the last remove-insert operation led to a solution that has not been accepted before with a better cost.
- σ_3: the last remove-insert operation led to a solution that has not been accepted before with a worse cost.

The three different scores balances intensification and diversification. Weight are updated every period with the following formula:

$$w_{\mathrm{i,\ j+1}} = w_{\mathrm{i,\ j}}(1 - r) + r\frac{\pi_{\mathrm{i}}}{\theta_{\mathrm{i}}}$$

where r represents the reaction factor, π_{i} represents the score obtained and θ_{i} represents the number of times the heuristics has been used in a period.

Aksen, Kaya, Salman and Tuncel [9] propose a similar approach similar to the one used by Ropke and Pisinger

3.2 The Irace Package

Irace is a tool that allows to automatically configure optimization algorithms, given a set of tuning instances of an optimization problem, by finding the best settings. Irace performs an *offline configuration*, that is usually addressed ad-hoc by algorithm designers with a trial-and-error approach by running a candidate algorithm on a set of instances. Other computational methods also exist and involve evolutionary algorithms, local search and regression methods. In particular sequential parameters optimization (SPO) uses statistical models for finding optimal parameters of an optimization algorithm. The main difference between SPO and Irace is that the former analyzes the impact and interaction of parameters of an optimization algorithm run on a single instance, while latter analyzes the impact of a configuration on a set of several instances.

3.2.1 The algorithm configuration problem

An algorithm is configurable if is has a number of parameters that can be set by the user. No single optimal setting for every possible application of the algorithm exists and it depends on the specific problem and its set of instances. Some of the parameters used for the algorithm components used to tackle the Air Liquide IRP are described in 2.5.2 and 2.6.2.
A formal definition of the algorithm configuration problem is given by Birattari [5]:

Definition 5. *Given a parametrized algorithm with N^{param} parameters $X_d, d = 1, ..., N^{param}$, each of them that may take different values: a configuration of the algorithm $\theta = \{x_1, ..., x_{N_{param}}\}$ is a unique assignment of values to parameters, and Θ is the possibly infinite set of all configurations of the algorithm.*

The set of possible instances of a problem may be seen as a random variable I from which instances to be solved are sampled. A cost measure $C(\theta, i)$ assigns a value to each configuration when applied to instance i. When the algorithm is stochastic, this cost is a random variable and $c(\theta, i)$ is a realization of the random variable $C(\theta, i)$. This cost may be the best objective function value found within a given computation time or the deviation from the optimum value if known. The criterion that wants to be optimized when configuring an algorithm for a problem is a function c_θ of the cost for a configuration θ with respect to the distribution of that random variable I.
A usual definition of c_θ it the expected cost of θ and it determines how configurations are ranked. The exact value of c_θ is often unknown and can be estimated by sampling: several realization $c(\theta, i)$ of the random variable $C(\theta, i)$ are performed, by evaluating an algorithm configuration on instances sampled from I.

3.2.2 Iterated racing

Iterated racing is a method for automatic configuration that consists of:

- sampling configurations according to a certain distribution.
- selecting the best configuration by means of racing.
- updating the sampling distribution to bias next sampling towards the best configurations.

Each configurable parameter has an independent sampling distribution, which is either a normal distribution for numerical parameters or a discrete distribution for categorical parameters. The update of the sampling distribution consists of modifying mean and standard deviation for normal distribution and the discrete probability values for discrete distribution.
A race starts with a finite set of candidate configurations. At each step, the candidate configurations are evaluated on a subset of instances. After each step, configurations that perform statistically worse than at least another one are discarded, and the race continues with the survived configurations. The procedures stops when is reached a minimum number of survived configurations, a maximum number of instances or a pre-defined budget (often defined as number of experiments, where an experiment is the the application of a configuration on an instance).

```
procedure Iterated Racing
input $I = \{I_1, I_2, ...\}$, parameter space: X, cost measure: $C(\theta, i)$, tuning budget: B
output: $\Theta^{elite}$
$\Theta_1 \sim SampleUniform(X)$
$\Theta^{elite} := Race(\Theta_1, B_1)$
$j := 2$
while $B_{used} \leq B$ do
    $\theta^{new} \sim Sample(X, \Theta^{elite})$
    $\Theta_j := \Theta^{new} \cup \Theta^{elite}$
    $\Theta^{elite} := Race(\Theta_j, B_j)$
    $j := j + 1$
end while
end Iterated Racing
```

3.2.3 The irace package

The program **irace** requires three inputs:

- a description of the parameters space X.
- a set of training instances {I_1, I_2, ...} representative sample of *I*.
- a set of options that define irace's configuration.

Irace uses the **hookRun** function to apply a certain configuration θ to an instance i and return the corresponding cost value $c(\theta, i)$. See figure 3.1 for a schematic description.

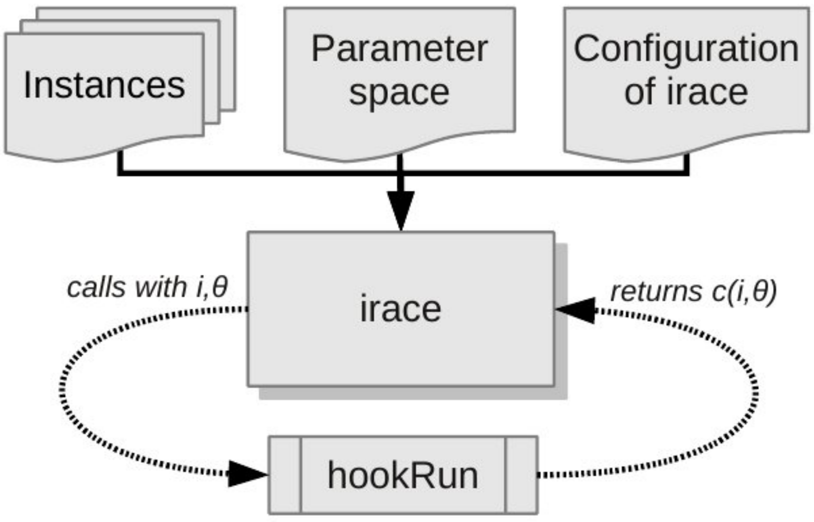


Figure 3.1: Scheme of **irace** flow information

Parameters are described in the following way:

<name> <label> <type> <range> [| <condition>]

Each parameter can be of four types:

- *real*: can take any floating-point values within the range **(<lower bound>,<upper bound>)**. The precision of a real parameter is given by **digits**.
- *integer*: can take only integer values within the range **(<lower bound>,<upper bound>)**.
- *categorical*: can be defined by a set of possible values **(<value_1>, ..., <value_n>)**. The values are strings and they make use of **<condition>** to determine whether it is enabled or not.
- *ordinal*: are an ordered set of categorical parameters. An integer parameter is assigned that corresponds to an index of the values.

Chapter 4

Experimental analysis

4.1 Instances

The description of each instance can be found at 1.3.6, but size and complexity of an instance manly depend on number of customers, number of drivers, number of trailers and horizon. See table 4.1 for details.

Table 4.1: Instances

	Customers	Drivers	Trailer	Horizon (h)
Instance_V_1.1	12	2	2	720
Instance_V_1.2	12	2	2	720
Instance_V_1.3	53	1	1	240
Instance_V_1.4	53	2	2	240
Instance_V_1.5	54	2	1	240
Instance_V_1.6	54	2	1	840
Instance_V_1.7	99	2	3	240
Instance_V_1.8	99	6	3	240
Instance_V_1.9	99	6	3	840
Instance_V_1.10	89	3	3	240
Instance_V_1.11	89	3	3	840

4.2 Software and hardware

Some building blocks, written in C++, were available: metaheuristics, local searches, acceptance critera and termination criteria. They are problem independent. Some other building blocks, written in C++, have been developed during this thesis work: initial solution methods, neighborhood operators and perturbations methods. They are problem specific and exploit some of the problems' characteristics.
The **Irace package**, written in R and available in the CRAN package, has been used for configurating and tuning the algorithm.
The experimental analysis have been performed on rack 5 of the Iridia Cluster: it consists of 4 computational nodes having the following characteristics:

- 4 Intel Xeon E5-2680 v3 (24 cores each, 2.5GHz, 2x 16MB L2/L3 cache).
- 128GB RAM
- 2TB harddisk
- 2x Gigabit ethernet

- RAM per job: 2.4GB
- the following queues: each queue has as many slot as the number of cores in that node:
  - <node>.short: jobs run in this queue at nice-level 2 (higher priority than the long queue) for maximum 24h of CPU time.
  - <node>.long: jobs run in this queue at nice-level 3 (lower priority than the short one) for maximum 168h of CPU time.

The cluster runs the Linux distribution Rocks 6.2, based on CentOS 6.2.
The experiments have been run in parallel and independent from one another. Each experiment has been run on a single CPU and has been submitted to the long queue.

4.3 Automatic tuning and configuration with Irace

All the building blocks, used to build the optimization algorithm to solve the Air Liquide IRP are the following:

- **SLS algorithms**:
  - *Iterated Local Search (ILS)*: iteratively apply a local search and use restart or perturbation to escape from local optima.
  - *Variable Neighborhood Descent (VND)*: iteratively apply a local search and switch to different neighborhood relations to escape from local optima.
- **Local Search**:
  - *first improvement*: evaluate neighbors in fixed order, choose first improving solution encountered.
  - *best improvement*: evaluate neighbors in fixed order, choose best improving solution encountered.
- **Termination**
  - *local minimum (Locmin)*: stop the search when a local minimum is found
  - *maximum steps*: stop the search when a certain number of steps have been performed
- **Initial Solution** (see 2.5.2 for a detailed description)
  - *greedy*: use heuristics to build a solution
  - *greedy randomized*: use heuristic to build several solution and select one according to a distribution probability on their objective value functions.
  - *GRASP*: generate partial solutions and select one according to a distribution probability on their objective value functions until a complete solution is built.
- **Neighborhood** (see 2.6.2 for a detailed description)
  - *exchange*: perform an exchange between two operations.
  - *insert*: insert a new operation according to an heuristic.
  - *remove*: remove an operation.
  - *refuel*: use a different inventory policy
- **Perturbation**:
  - *random move*: a random search step in a certain neighborhood.
- **Acceptance**
  - *simulated annealing (SA)*: use metropolis acceptance criterion with start and end temperature, step and frequency.
  - *metropolis (Metro)*: use metropolis acceptance criterion with fixed temperature

– *improve*: accept the solution if there is an improvement in the objective function value.
– *always*: always intensify or always diversify.

The first experiments has been performed using an arbitrary ad-hoc configuration, where the choice was based on trial-and-error runs of several possible configurations and parameter values. All other tests have been performed using the configuration obtained with the Irace package with different values for budget (number of configurations) and computation time. See table 4.2 for details.

Table 4.2: Irace settings

	Budget	**Computation time**
Conf 1	-	-
Conf 2	5000	300
Conf 3	5000	1800
Conf 4	20000	300
Conf 5	20000	1800

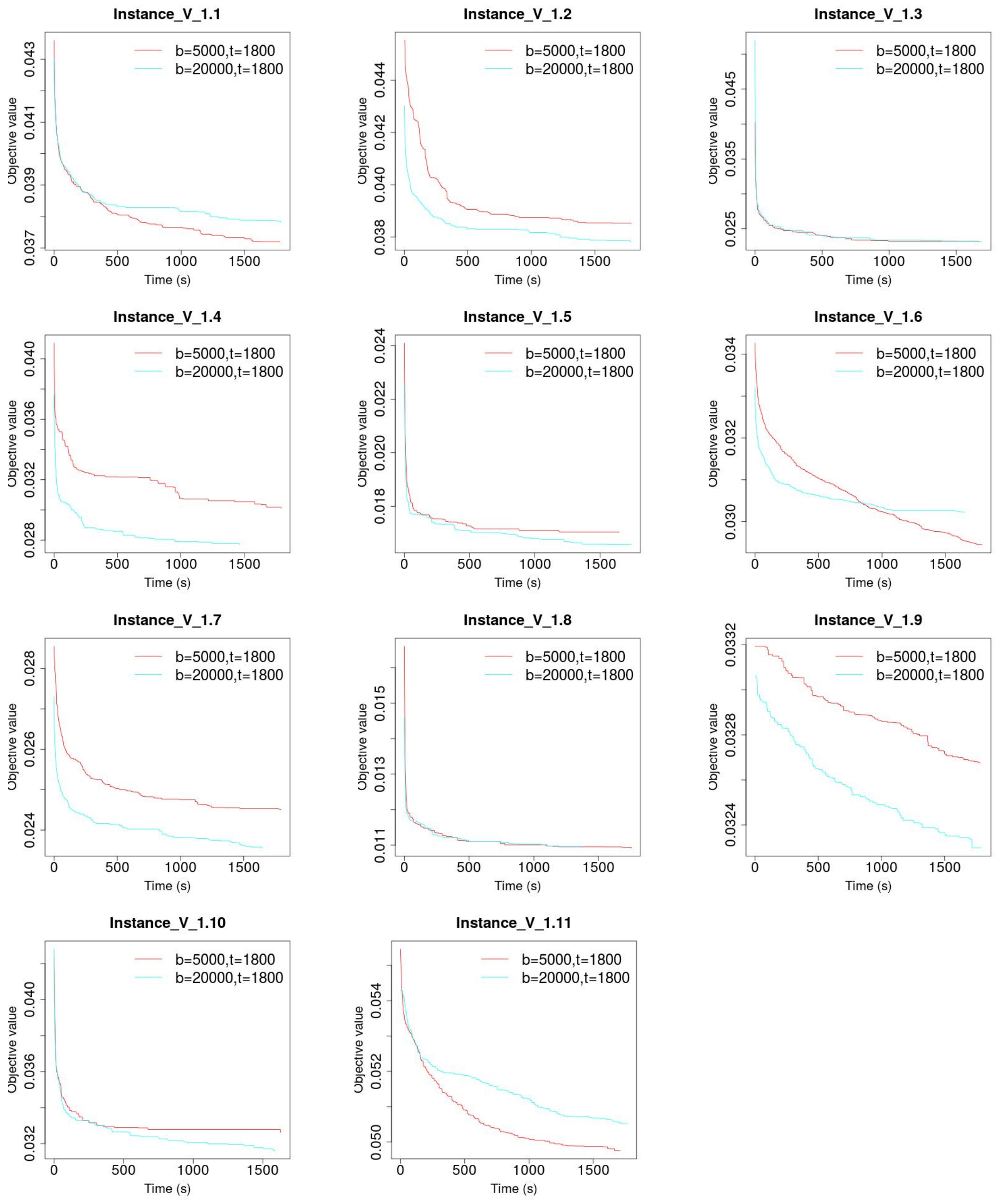
The arbitrary configuration and all configurations obtained with Irace are shown in table 4.3. It can be noticed that:

- ILS is chosen at each run of Irace as SLS algorithm.
- First improvement is chosen over best improvement as pivoting rule.
- Local minimum is always chosen as termination criterion.
- Greedy is chosen on most cases as initial solution, while GRASP is not chosen at all.
- Exchange and Remove are the most chosen operators.
- Simulated annealing is the most chosen acceptance method.

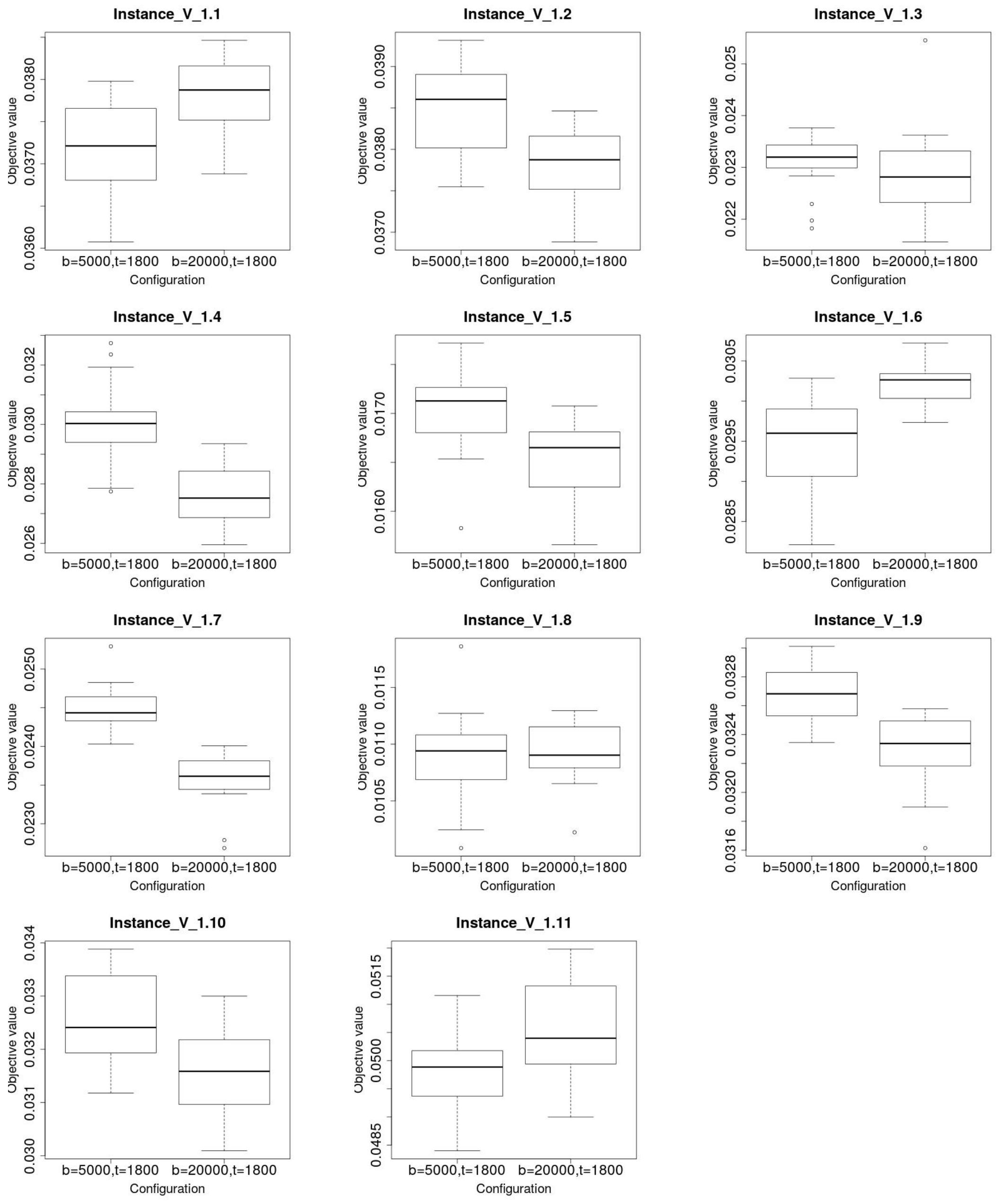
Table 4.3: Configurations

	Conf 1	**Conf 2**	**Conf 3**	**Conf 4**	**Conf 5**
ILS	ILS	ILS	ILS	ILS	ILS
LS	First	First	First	First	First
Termination	Locmin	Locmin	Locmin	Locmin	Locmin
Initial solution	Greedy	Greedy	Greedy	Greedy randomized	Greedy
Neighborhood	Exchange	Remove	Remove	Exchange	Exchange
Perturbation	Insert	Exchange	Remove	Remove	Remove
Acceptance	Metro	SA	Diversify	SA	SA

Some of the configurations obtained with different runs of Irace have been compared to check how budget and computation time affect Irace's performance. Each experiment have been run 15 times.



(a) RTD plots using different budget values



(a) RTD boxplots using different budget values

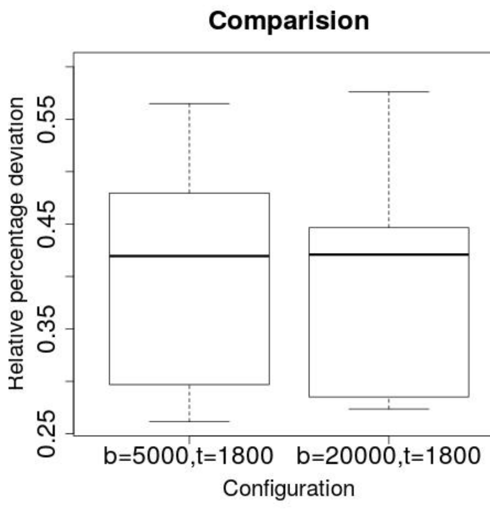
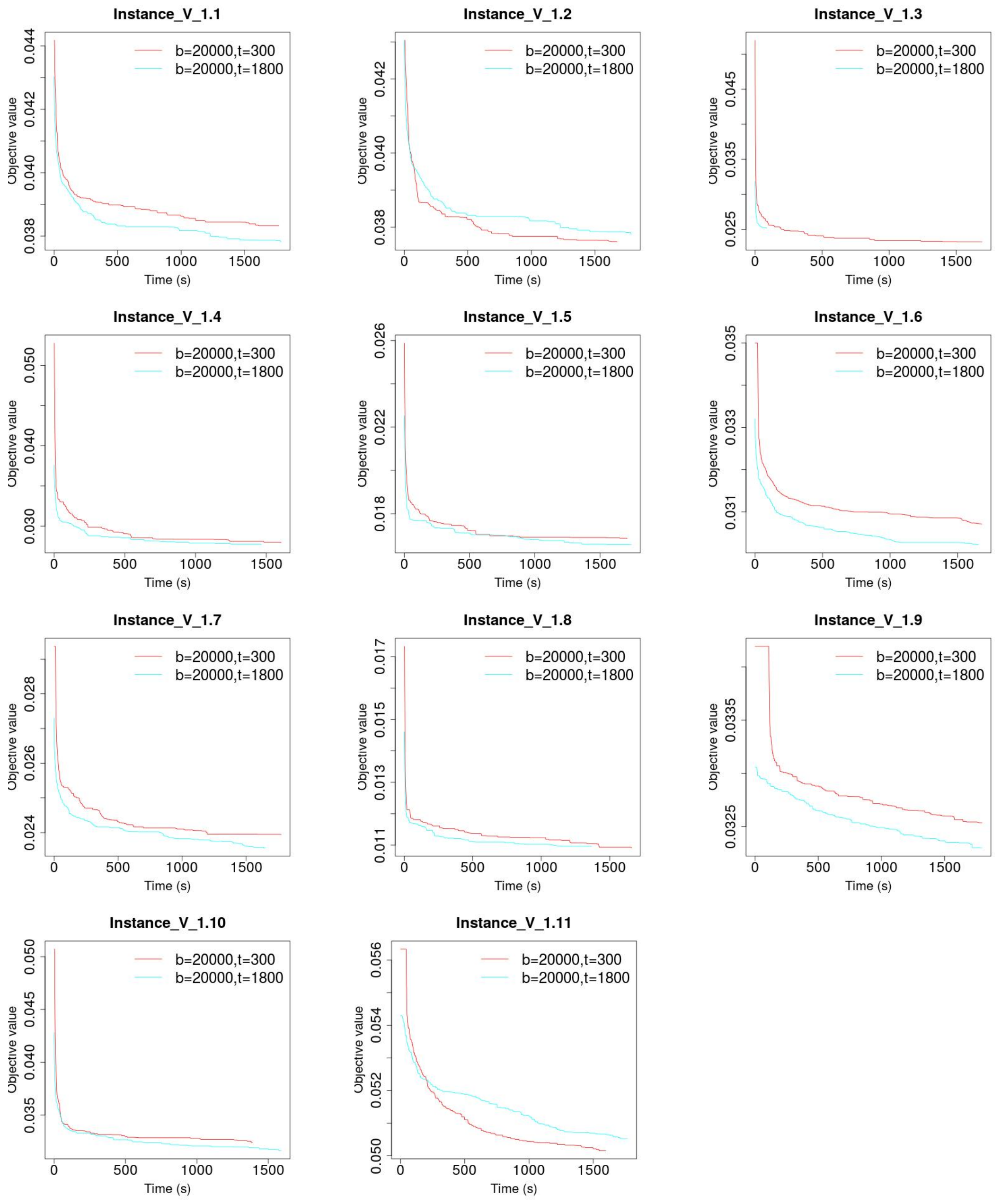
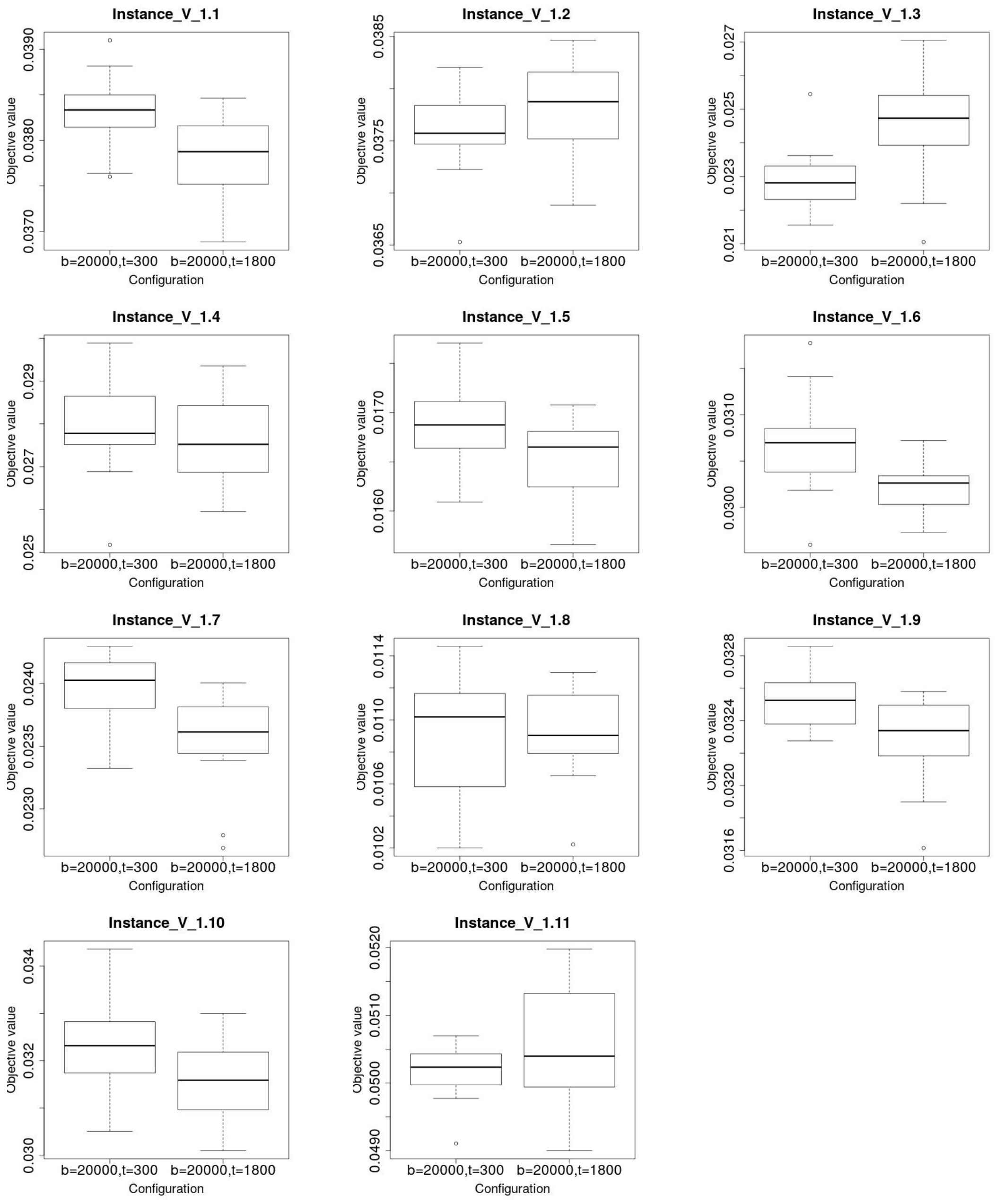


Figure 4.3: Performance boxplots using different budget values.

The first comparison is between the configurations obtained using different budget values (configuration 3 and configuration 5). Plots 4.1a show the run time distribution (RTD) of the two algorithms on every instance, while boxplots 4.2a show the distribution of the best objective function values found for 30 minutes runs on every instance. Boxplot 4.3 show the relative standard deviation (RSD) of the two algorithms with respect to the best objective function values known.
It can be noticed that the configuration obtained running Irace with a higher budget outperforms the one with lower budget for seven instances over eleven, and the RSD with respect to the best objective values known also shows that. Overall, there is a slight improvement, so it is worth using a higher budget.



(a) RTD plots using different computation time



(a) RTD boxplots using different computation time

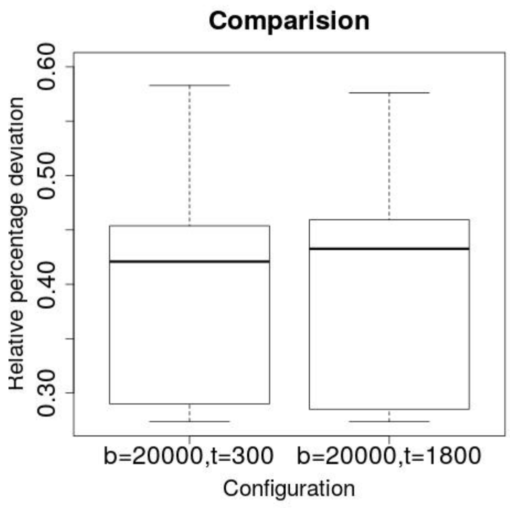
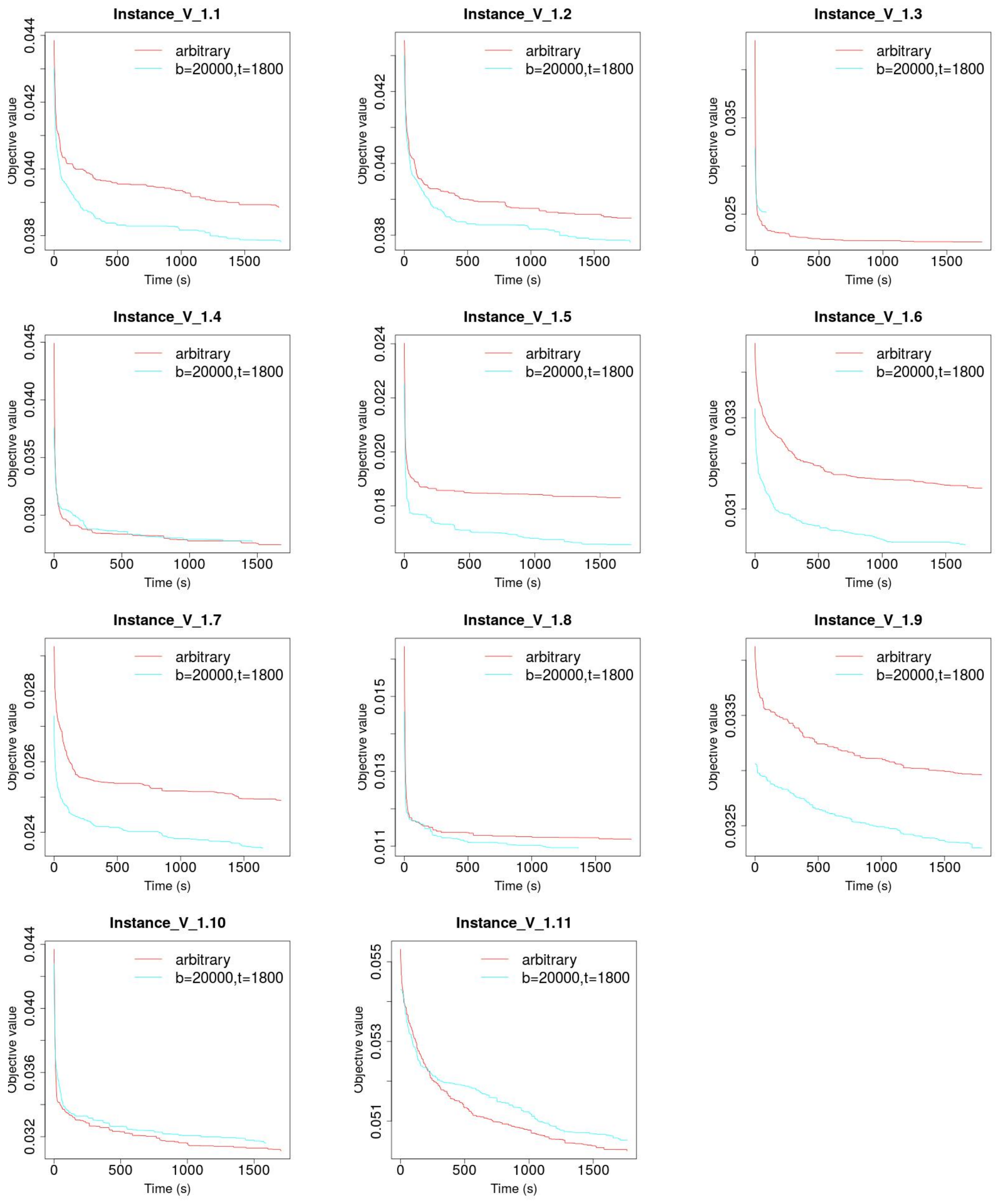
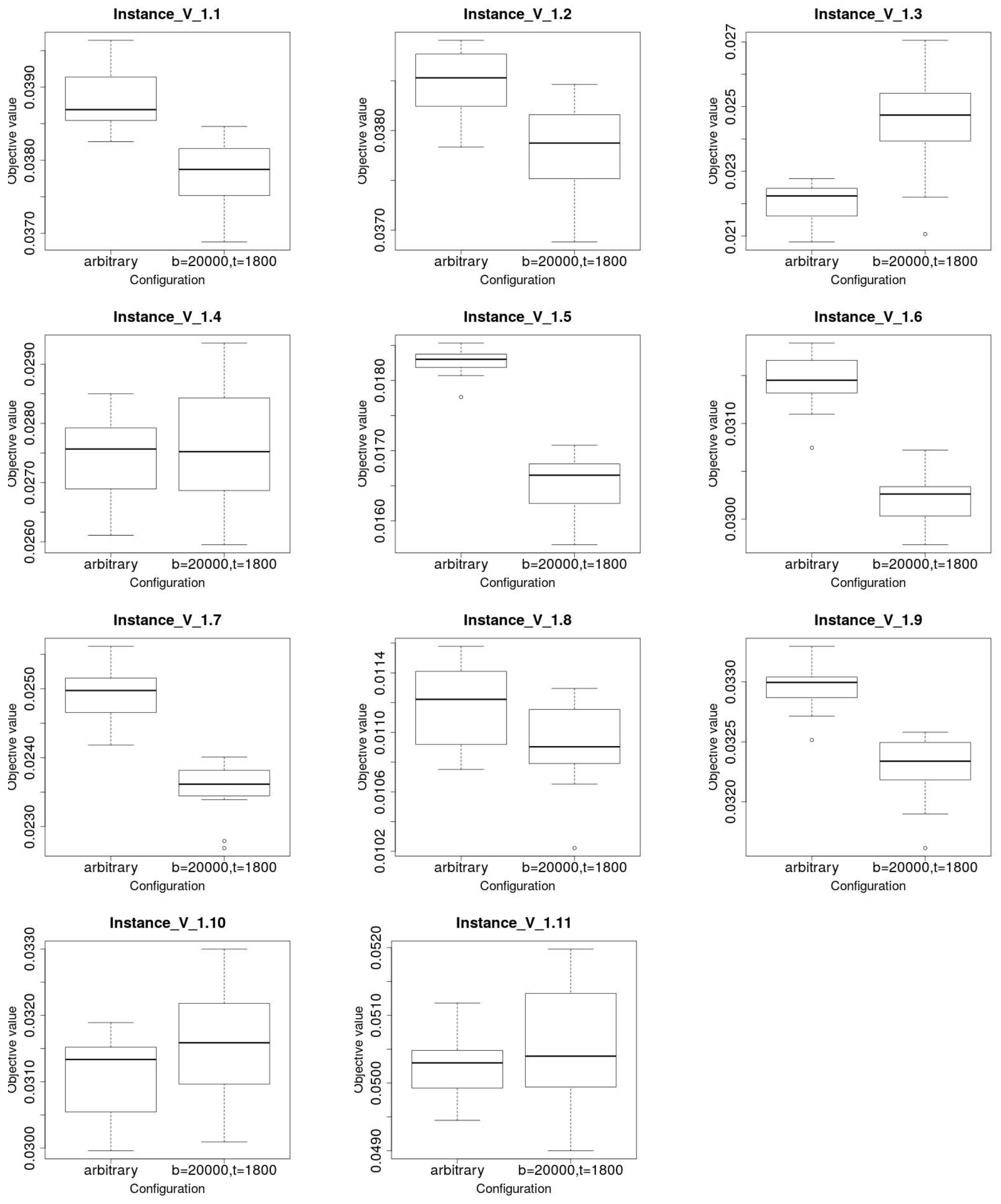


Figure 4.6: Performance boxplots using different computation time

The second comparison is between the configurations obtained using different computation time (configuration 4 and configuration 5). Plots 4.7a show the run time distribution (RTD) of the two algorithms on every instance, while boxplots 4.8a show the distribution of the best objective function values found for 30 minutes runs on every instance. Boxplot 4.3 show the relative standard deviation (RSD) of the two algorithms with respect to the best objective function values known.
It can be noticed that the configuration obtained running Irace with a higher computation time outperforms the one with lower budget for seven instances over eleven, and the RSD with respect to the best objective values known also shows that. Overall, there is a slight improvement, so it is worth using a higher computation time.



(a) RTD plots using different computation time



(a) RTD boxplots using different computation time

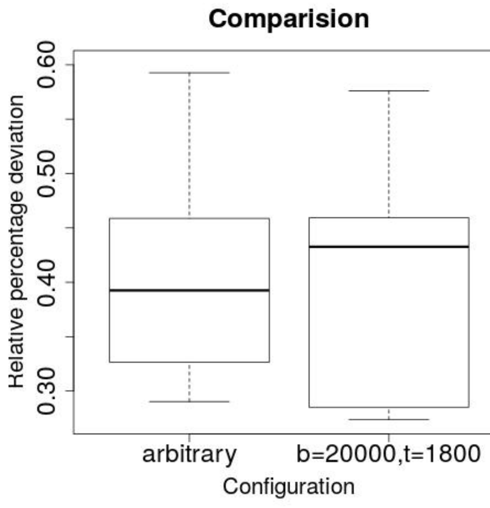


Figure 4.9: Performance boxplots using different computation time

The last comparison is between the arbitrary configuration and the configuration obtained running Irace with higher budget and computation time (configuration 1 and configuration 5).Plots 4.7a show the run time distribution (RTD) of the two algorithms on every instance, while boxplots 4.8a show the distribution of the best objective function values found for 30 minutes runs on every instance. Boxplot 4.3 show the relative standard deviation (RSD) of the two algorithms with respect to the best objective function values known.
It can be noticed that the configuration obtained with Irace outperforms the arbitrary configuration for seven instances over eleven, and the RSD with respect to the best objective values known also shows that. So the automatic tuning and configuratiom performed by Irace outperform the ad-hoc configuration.

Bibliography

[1] Holger H. Hoos Thomas Stützle *Stochastic Local Search - Foundations and Applications*, Morgan Kaufmann Publisher, San Francisco, CA June 2005

[2] Michael R. Garey, David S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco, CA, USA 1979

[3] Leandro C. Coelho, Jean-FranÃğois Cordeau, Gilbert Laporte, *Thirty Years of Inventory-Routing*, CIRRELT-2012-52, September 2012.

[4] *Inventory Routing Problem Description for ROADEF/EURO 2016 challenge*, http://challenge.roadef.org/2016/en/sujet.php

[5] Mauro Birattari, *Tuning Metaheuristics: a Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence*, Springer-Verlang.

[6] Paul Shaw, *Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problem*, April 1998.

[7] Vikas Goel, Kevin C. Furman, Jin-Hwa Song, Amr S. El-Bakry *Large Neighborhood Search for LNG Inventory Routing*, ExxonMobil.

[8] Stefan Ropke, David Pisinger, *An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows*, November 2016.

[9] Deniz Aksen Onur Kaya F. Sibel Salman Özge Tüncel *An Adaptive Large Neighborhood Search Algorithm for a Selective and Periodic Inventory Routing Problem*, June 2014.