POLITECNICO DI MILANO

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

AUTOMATIC TUNING AND CONFIGURATION OF METAHEURISTICS FOR THE INVENTORY ROUTING PROBLEM

Relatore: Prof. Pier Luca LANZI
Correlatore: Prof. Thomas Stützle
Correlatore: Dott. Federico Pagnozzi

Tesi di laurea di:
ANTONIO MARIA FISCARELLI
Matr. 820571

Anno Accademico 2015 - 2016

The people who survive the sword will find favor in the desert.
[Jeremiah 31:2]

Aknowledgements

Foremost, I would like to express my sincere gratitude to my co-advisor Prof. Thomas Stützle of the Université libre de Bruxelles for the continuous support of my master thesis, for his patience and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor.
I thank my thesis advisor prof. Pier Luca Lanzi at Polytechnic University in Milan for the many discussions carried on long-distance via e-mail, and for helpful comments on the text.
I would also like to thank Federico Pagnozzi. The door to Federico's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right the direction whenever he thought I needed it.
I thank my fellow labmates at Iridia for welcoming me as a friend, the stimulating discussions and for all the fun we have had in the last year.

Ringrazio i miei genitori che hanno sempre creduto in me, mi hanno sostenuto economicamente, senza tirarsi mai indietro e senza i quali non sarebbero stati possibili i miei studi. Non ci sono parole per descrivere quello che loro fanno per me. Intendo poi ringraziare Carmen, mia sorella maggiore, perché è presente costantemente in quello che mi riguarda, aiutandomi di volta in volta nelle difficoltà. Intendo ringraziare infine mia zia Sandra e mio cugino Pierluigi, che mi sono sempre stati vicini durante la mia permanenza a Milano.
Voglio ringraziare i miei amici di Foggia per tutte le serate in piazza, le nottate in spiaggia e le risate insieme.
Voglio ringraziare Gianluca, Alice e Andreina, per avermi accettato per come sono e per avermi insegnato a non prendermi troppo sul serio.

J'aimerais remercier Franco, Laurent, Yulia et Leila. Nous sommes tous différents mais nous partageons tous la même passion, celle de crér un monde meilleur. En particulier j'aimerais remercier Isolina. Merci de m'avoir fait découvrir un monde où la compassion envers chaque petite chose vivante est possible.

Prefazione

L'"Air Liquide Inventory Routing Problem" è un problema reale presentato dalla compagnia francese Air Liquide per il 2016 ROADEF/EURO challenge. Appartiene alla classe di Inventory Routing Problems, una classe di problemi che combina aspetti di Vehicle Routing e Inventory Management. Diverse varianti di questo problema sono state affrontate data la loro diversa struttura e la diversa disponibilità di informazione riguardo la domanda dei clienti.
L'obiettivo principale di questo lavoro è quello di sviluppare delle componenti algoritmiche da integrare in un framework per lo sviluppo di algoritmi metaeuristici ed usare il pacchetto Irace per trovare la configurazione e il set di parametri ideale.
Alcuni algoritmi esatti e potenti algoritmi di ricerca stocastici sono stati sviluppati per risolvere questa classe di problemi. In particolare, questi ultimi sono molto efficaci perché possono risolvere problemi computazionalmente complessi in modo efficiente e robusto, specialmente quando le istanze di tali problemi diventano molto grandi e gli algoritmi esatti non riescono a trovare una soluzione in un tempo ragionevole.
Tre componenti per generare una soluzione iniziale sono stati sviluppati e quattro diversi operatori sono stati definiti. Una funzione delta è stata definita per velocizzare la valutazione di una soluzione. Una variante della funzione obiettivo originale è stata proposta per tenere in considerazione il fatto che alcune soluzioni violano i vincoli.
Degli esperimenti sono stati eseguiti sul cluster dell'Iridia usando l'insieme di istanze reso disponibile per la competizione. L'analisi ha mostrato che la configurazione scelta da Irace ottiene migliori risultati rispetto ad una configurazione scelta in maniera casuale a una configurazione scelta in base alla mia esperienza del problema. Inoltre, un'analisi di sensitività è stata fatta per verificare come i parametri di un algoritmo ne influenzano le performance.

Abstract

The Air Liquide Inventory Routing Problem is a real-world optimization problem proposed by the French company Air Liquide for the 2016 ROADEF/EURO challenge. It belongs to the class of Inventory Routing Problems, a class of problems that combines aspects of vehicle routing and inventory management. Many variants of this problem has been described in the literature due to their different structure and the different availability of information on the customer's demand.
The main goal of this work is to develop problem-specific algorithmic components that will be integrated in a framework for the design of general-purpose metaheuristic algorithms and use the Irace package to find the best configuration and set of parameters for the algorithm. Some exact algorithms and powerful stochastic local search methods have been developed to solve this problem. In particular, stochastic local search methods are very promising because they can solve computationally hard problems very effectively and robustly, especially when problem size increases and exacts methods fail to find a solution in a reasonable time.
Three different components to generate an initial solution have been developed and four different neighborhood operators have been defined. In order to speed-up the evaluation of a solution, a delta evaluation function has been defined. In order to consider the unfeasibility of solutions, a variant of the original objective function has been proposed.
An experimental analysis has been performed on the Iridia cluster using the set of instances available for the competition. The analysis shows how the configuration found by irace outperforms a configuration chosen randomly and an ad-hoc configuration. Furthermore, a sensitivity analysis has also been performed to investigate how the parameters affect the algorithm performance.

Contents

Chapter 1

Introduction

The problem approached in this thesis the Air Liquide Inventory Routing Problem, a real-world optimization problem proposed by the French company Air Liquide for the 2016 ROADEF/EURO challenge. It belongs to the class of Inventory Routing Problems, a class of problems that combines aspects of vehicle routing and inventory management.

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]. Many variants exist due to their different structure: they can differ in the definition of horizon, number of suppliers and customers, type of routes allowed, inventory policies, inventory management, fleet type, number of vehicles and different availability of information on the customer's demand
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. Applications of the IRP arise in a large variety of industries, including the distribution of gas using tanker truck, road-based distribution of automobile components and of perishable items. Other applications include the transportation of groceries, cement, fuel, blood, and waste organic oil.

This problem is known to be NP-hard, since it includes the Vehicle Routing Problem (VRP) as a subproblem [3]: there is no technique that can solve it in polynomial time. Some exact algorithms and powerful stochastic local search methods have been developed to solve this problem. In particular, stochastic local search methods can be very effective: they can solve computationally hard problems very effectively and robustly, especially when problem size increases and exacts methods fail to find a solution in a reasonable time. Therefore, large instances of IRP are solved using heuristic approaches that can find relatively good quality solutions in a short amount of time. While some stochastic local search methods can be simple and effective on a wide range of applications, some others are much more complex and problem specific.

Stochastic local search algorithms are a class of metaheuristics that consist of search methods that combine heuristic local information and randomization to generate or select solutions [1]. The algorithms are defined by several components and the right choice of these components can affect the algorithm's performance. In this work some problem-specific algorithmic components are developed and integrated in a framework for the design of general-purpose

metaheuristic algorithms. The Irace package is then used to find the best configuration and set of parameters for the algorithm.
Three different components to generate an initial solution have been developed. These components are constructive heuristics that build one or more solutions and select one of them according to a probability that depends on their quality.
Four different neighborhood operators have been defined for the local search. A perturbation method is also available that performs a certain number of random steps in the search space of one of the neighborhoods defined. A delta evaluation function has been defined in order to speed-up the evaluation of a solution and a variant of the original objective function has been proposed to consider the unfeasibility of solutions.

An experimental analysis has been performed on the Iridia cluster using the set of instances available for the competition: three different algorithms have been tested. For the first algorithm all components and parameters have been selected randomly. For the second algorithm all components have been selected using an approach based on personal experience and trial-and-error runs. The third algorithm has been generated using the Irace package. A statistical hypothesis test has been performed to compare the performance of the three algorithms and the one using Irace is shown to outperform the others on almost all instances. A sensitivity analysis has also been used to investigate how parameters affect the algorithm's performance. It is shown that the optimal values found are similar to the ones selected by irace.

This thesis is structured as follows: Chapter 2 presents some of the IRPs in literature and the Air Liquide IRP. Chapter 3 provides an introduction to metaheuristics and how the different IRPs in literature and the Air Liquide IRP have been approached. Chapter 4 gives an overview on automatic configuration and parameter tuning, some of the methods used in literature and the Irace package used for the Air Liquide IRP. Chapter 5 presents experimental setup, tests performed and statistical analysis. Finally, in Chapter 6 some conclusions are drawn and possible extensions plus further works are presented.

Chapter 2

Inventory Routing Problem

This chapter introduces the problem that has been faced during this thesis work. The first section gives a general description of IRP. Then some of the IRPs in literature are presented. In particular, the Air Liquide IRP is presented and all its features and constraints are described. Finally, all problem instances available are described.

2.1 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 and 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 some criteria [3]:

- 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 when one supplier serves 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 policy (ML), the replenishment level can vary and it's bounded by each customer's capacity. Under the order-up-to-level policy (OU), the replenishment level is that to fill the customer's 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 this information is fully known to the decision maker at the beginning of the planning horizon, the problem is deterministic; if its probability distribution is known, the problem is stochastic (SIRP); if demand is not fully known in advance, but is gradually revealed over time, the problem is dynamic (DIRP). For the DIRP, one can still exploit its statistical distribution: in this case the problem is a Dynamic and Stochastic IRP (DSIRP) [3].

2.2 Different IRPs in literature

Since the particular IRP faced during this work was proposed this year, there was no published work available on this specific problem. There is a huge number of different IRPs in the literature though, and some papers were reviewed.

Claudia Archetti, Luca Bertazzi, Alain Hertz and M. Grazia Speranza [6] propose a hybrid heuristic for an IRP. A supplier has to serve a set of customers over a time horizon. Each customer has a capacity constrain and the service time cannot cause any stock-out. A single vehicle with a given capacity is available. The transportation cost depends on the distance traveled, while the inventory holding cost depends on the level of the inventory at the site. The IRP they examined has the following characteristics:

- Finite horizon.
- One-to-many structure.
- Multiple routes.
- ML and OU policies.
- Lost sales.
- Homogeneous fleet.
- Single vehicle.

The objective is to minimize the sum of the transportation and inventory costs.
Paul Shaw [7] proposed an approach based on constraint programming and local search methods for solving the Capacitated Vehicle Routing Problem (CVRP). The CVRP they examined 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 [8] propose a Large Neighborhood Search (LNS) for the Liquified Natural Gas Inventory Routing Problem (LNGIRP). It is considered a special case of Maritime Inventory Routing Problem (MIRP), that combines inventory management and ship routing. Christiansen and Fagerholt [9] define a MIRP as the transportation of a single product that is produced at loading ports and consumed at unloading ports where each port has a given inventory storage capacity and a production or consumption rate. However, the LNGIR addressed in this article includes several variations concerning variable production and consumption rates, LNG specific contractual obligations and berth constraints. The LNGIR examined 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 to minimize the sum of lost production, stockout and unmet demands.

Ropke and Pisinger [10] propose an Adaptive LNS for a variant of the Pickup and Delivery Problem with Time Windows (PDPTW). A number of requests and vehicles are given. A request consists of picking up goods at one location and delivering these goods to another location. Goods must be picked up and delivered within specific time windows. Service times are assigned with each pickup and delivery. Each request is assigned a set of vehicles that are allowed to enter the location, having limited capacity. A start time indicates when a vehicle must leave the start location and an end time indicates when a vehicle must return to the end location. The PDPTW 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 [11] propose an Adaptive LNS for a Selective and Periodic Inventory Routing Problem (SPIRP) for recovery and reuse of waste oil. The problem is defined on a complete directed graph with a set of source nodes and a depot. It has a cyclic planning horizon over a period of seven days. Waste vegetable oil is accumulated at the source nodes in each period *t*. The source node can be visited at any period by a fleet of vehicles having fixed capacity, that have to collect the total amount of waste oil accumulated since the last visit (no partial collection is allowed). The facility requires a certain amount of oil in a period **t** in order to produce enough biodiesel according to the production plan. It can be produced by collecting waste oil from source nodes (like previously described), by purchasing virgin oil or by using the waste oil inventory buildup at the depot. Different costs are assigned to these activities. 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 purchasing costs.

2.3 Air Liquid Inventory Routing Problem

This thesis deals with a problem called 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 to maximize the 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.

2.3.1 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 the following:

- 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 below 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 shift can be assigned to a driver only within one of his time windows.
  - A driver can only drive one trailer.
  - Each trailer has a specific capacity.
- Shift definition:
  - Each shift has only one driver and only one trailer.
  - A driver has to start from the base and come back to the base.
  - A 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 variants already exist 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.

2.3.2 Problem setting

2.3.2.1 Units of measure for quantity

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

2.3.2.2 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.

2.3.2.3 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.

2.3.2.4 Trailers master data

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

- DistanceCost(tl): The cost per distance unit for trailer tl (in €/minute).
- Capacity(tl): the capacity of trailer tl in mass (Kg). This capacity corresponds to the usable capacity which is the quantity of product that can be loaded in the trailer and delivered to customers. It does not include the minimum quantity of product that must remain in the trailer and the volume that must remain empty for pressure considerations.
- InitialQuantity(tl): the mass (in Kg) of usable product in trailer tl at time 0. Must belong to the interval [0, Capacity(tl)].

2.3.2.5 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 whenever necessary, in a cost efficient way. 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 one 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.

2.3.3 Solution

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

- driver(s): the driver for shift s.
- trailer(s): the trailer (and associated tractor) for shift s.
- start(s): the starting time for 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 plac, either sources or customers.
- quantity(o): the quantity to be delivered or loaded in operation o. It is negative for loading operations at sources, or positive for delivery operations at customers (in kg).

2.3.4 Constraints

2.3.4.1 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.

2.3.4.2 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 ∈ Drivers
    For all $s_1$, $s_2$ ∈ 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 ∈ Shifts
    For all operations o ∈ 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.

```
For all s ∈ Shifts
    For all operations o ∈ Operations(s) ∪ {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 ∈ Shifts
    There exists a tw ∈ TimeWindows(Drivers(s)) such that start(s) ⩾ start(tw) AND end(tw) ⩾
end(s)
EndFor
```

2.3.4.3 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 s_1.

```
For all tl ∈ Trailers
    For all $s_1$, $s_2$ ∈ 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 ∈ Shifts
    trailer(s) = Trailer(driver(s))
```

2.3.4.4 Constraints related to the sites

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

```
For all p ∈ Customers
    For all h ∈ [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 time h, plus all deliveries performed at time 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 ∈ {c ∈ Customers}
    tankQuantity(p, -1) = InitialTankQuantity(p)
EndFor
For all h ∈ [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).

2.3.4.5 Constraints related to shifts

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

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

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

2.3.4.6 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 safety level SafetyLevel(p), at all times.

```
For all p ∈ Customers,
    For all h ∈ [0, H-1]
        SafetyLevel(p) ⩽ tankQuantity(p, h)
    EndFor
EndFor
```

2.3.5 Optimization goal

2.3.5.1 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, the **logistic ratio** has to be minimized. The logistic ratio 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 $LR = 0$

2.3.5.2 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 for 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.

2.3.6 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 2.3.2.

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 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, 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: It is a variant of the instance 7, with more drivers.
  - 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 It is a variant of the instance 8, with an enlarged optimization horizon.
  - 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 and q, the shortest way to go from customer p to customer q is the path $(p,\ q)$ connecting p and q. In this case, the shortest distance to go from p to q will be given by DistanceMatrix(p,q).

It's interesting to see that some customers have enough initial product in their tanks to satisfy their demand. So no deliveries needs to be performed for them since constraint **QS02** 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 **QS02**.
- **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 3

Metaheuristics

This chapter provides an understanding of metaheuristics and how they are designed to solve combinatorial problems. First sections introduce combinatorial problems, give a definition of Stochastic Local Search (SLS) and describe its components. Following sections describe some methods presented in the literature to generate initial solutions and the methods proposed for the Air Liquide IRP. Some neighborhood definitions presented in the literature for different problems and the neighborhood operators defined for the Air Liquide IRP are also presented. At the end, some considerations about the complexity of the objective function lead to the definition of a delta evaluation function to evaluate solutions of the Air Liquide IRP and an alternative objective function is defined to take into account unfeasibility of solutions.

3.1 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.

3.1.1 Problems and solutions

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

- A **problem** is general and abstract . The solution of this problem is an algorithm that, given a problem instance, determines a solution for that instance.
- A **problem instance** would be, for the Air Liquide IRP, to find the set of shifts that satisfy the demand of a specific set of customers with minimal cost.

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 Air Liquide IRP, typically any valid set of shifts of any cost would be a candidate solution.
- **Solutions** satisfies all constraints instead. In this case, only those sets of shifts satisfying the demand of all customers and all other constraints are considered solutions.

3.1.2 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.

3.1.3 Optimization problem

Many practical combinatorial problems are optimization problems 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 values.
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.

3.2 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 Air Liquide IRP instance, its size corresponds to the horizon length and the number of customers.
The complexity of a problem can be defined as the complexity of the best algorithm used to solve this problem.

3.2.1 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 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 a 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**.

3.3 Search paradigms

The fundamental idea behind the search approach is to iteratively generate and evaluate candidate solutions [1]. 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 of them to obtain a new candidate solution. Applied to the Air Liquide IRP, a simple perturbative search would start with one complete solution and, for example, exchange the order of two customers that are served.
- **Constructive search methods** search in the space of **partial candidate solutions** instead. For the Air Liquide IRP, a simple partial candidate solution would be a set of shifts that satisfy the demand of a subset of all customers. 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 to generate candidate solutions with a "good" objective value; the choice is led by heuristic and local information. For the Air Liquide IRP, a constructive search method would start at a randomly chosen customer, and then iteratively add customers with minimal traveling distance between the current customer and the one not visited yet, until the demand of all customers have been satisfied.
- **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**.
- **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 not complete and perturbative based.

3.4 Stochastic local search

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

3.4.1 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.
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*** *$N(\pi) \subseteq S(\pi) \times S(\pi)$;*
- *a finite* ***set of memory states*** *$M(\pi)$ that, in the case of SLS algorithms that do not use memory, may consist of a single state only;*
- *an* ***initialization function*** *$init(\pi) : \emptyset \to D(S(\pi) \times M(\pi))$, would specify a probability distribution over initial search positions and memory states;*
- *a* ***step function*** *$step(\pi) : S(\pi) \times M(\pi) \to D(S(\pi) \times M(\pi))$ mapping each search position and memory state onto a probability distribution over its neighboring search position and memory states;*
- *a* ***terminal predicate*** *$terminate(\pi) : S(\pi) \times M(\pi) \to D(\top, \bot)$ 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 $D : S \to R_0^+$ 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
```

3.4.2 Iterative Local Search

Given search space, solution set and neighborhood relation, some *search strategies* can be defined by the use 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 LS 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 $N \subseteq S \times S$.
A very simple SLS algorithm, instead, can be the Uninformed Rand Walk with Random Restart. But since it does not provide any mechanism to guide the search towards better solution, it's quite ineffective.

Definition 3 (**Evaluation function**). *Given $\pi \in \Pi$ an instance of a decision problem Π, an evaluation function $g(\pi)(s) : S(\pi) \mapsto \mathbb{R}$ is a function 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).

3.4.3 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 local minima or confined regions that does not contain high quality solution.
- **intensification strategies** greedily improve solution quality or the chances to improve solution quality solution in the near future.

3.4.4 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 value.
- **First improvement** select the first neighboring solution encountered that achieves an improvement in the evaluation function value.

First improvement avoids evaluating all neighbors to reduce time complexity, but the order in which they are evaluated 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 may not be optimal for a different one. **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 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 search steps have been performed.

3.5 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 by iteratively adding solution components to a partial solution. A greedy function is used to chose the next component to add 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.

3.5.1 Initial solutions for IRP in literature

Goel, Furman, Song and El-Bakry [8] propose a constructive heuristic to generate an initial solution for the Large Neighborhood Search (LNS) used to solve the LNGIRP: the constructive 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 [11] use exact methods and local search techniques to generate a solution for their Adaptive LNS for the SPIRP for recovery and reuse of waste oil. 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.

3.5.2 Initial solutions 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, then, chooses one of them according to their objective function value.

- **Partial Greedy Randomized**: constructs a solution adding one component at time, starting from an empty solution. It builds several candidate components using heuristics and adds one of them according to the improvement in the objective function value, until a complete solution is built.

All of them use two functions:

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

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

The Construct Solution function accepts a solution that may be empty or partial, in the sense that some customers' demand may be not yet satisfied, 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 for the remaining horizon, can be derived from these.
- two greedy functions are 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.
$$\frac{w_{\mathrm{t}} * runouTime + w_{\mathrm{q}} * (1.0 - |totalDeman - totalDeliveredProduct|)}{2}$$
  - *horizon*: the horizon in hours.
  - *runoutTime*: time at which the tank is expected to go below the minimum level. It depends on totalDeliveredProduct, initial tank level at customer's site and customer's forecast.
  - *totalDemand*: total demand of product for a customer. It's given by the sum of customer's forecast over the horizon.
  - *totalDeriveredProduct*: total amount of product delivered to a customer in the current solution.

  The second greedy function considers a short term profit, since it considers the cost of the next delivery.
$$\frac{w_{\mathrm{t}} * runouTime + w_{\mathrm{q}} * Cost(p,q)}{2}$$

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

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

  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 q 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 $p = q$.
  Customers are then inserted in a priority list alongside with their greedy value and sorted in an ascending order (from best to worst). 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**.
  - 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.
  - in case the quantity of product is bigger than the total demand of product left for the customer, only the quantity needed to satisfy the demand will be delivered.
  - the quantity of product delivered is then determined according to the parameter *deliveredQuantityFactor*. For *deliveredQuantityFactor* = 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 deliveries can no longer be scheduled in a time window, the next time window will be used.

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

Extend Solution function has the following parameters:

- *deliveredQuantityFactor*: controls the quantity of product delivered to a customer.
- *refuelFactor*: determines when a vehicle has to go back to the source to refuel.

Extend Solution function accepts a solution that may be empty or partial, 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 used 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. Customers are then inserted in a priority list alongside with their greedy value and sorted in an ascending order (from best to worst). 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**.
  - 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.
  - the quantity of product delivered is then determined according to the parameter *deliveredQuantityFactor*.

  When 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 Construct Solution function to generate a solution (that possibly satisfy all customer's demands) and the Extend Solution function to extend the solution and improve even more it's objective function value.
Greedy builds solutions according to different values (*timeWeight*, *quantityWeight*, *ties*) and returns the first feasible solution found, or the best unfeasible solution.

```
For all $w_t \in \{0, 0.1, \ldots 1\}$
    For all $w_q \in \{0, 0.1, \ldots 1\}$
        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 according to different values (*timeWeight*, *quantityWeight*, *ties*) and keep them in a list alongside with their objective function values. A vector of probabilities is build according to the objective function values and a solution is returned according to those probabilities: a better quality solution is more likely to be selected.

```
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, starting from an empty solution, builds solution components according to different values (*timeWeight*, *quantityWeight*, *ties*) and add them to the current partial solution. These partial solutions are kept in a list alongside with their new objective function values due to the new component added. A vector of probabilities is build according to the improvement in the objective function values due to the new component and a solution is selected for the next step according to those probabilities: a better quality solution is more likely to be selected. 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
```

3.6 Neighborhood definition

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

3.6.1 Neighborhood definition in literature

Paul Shaw [7], for solving the CVRP, 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 can move the search further at each step.

Goel, Furman, Song and El-Bakry [8], for solving the LNGIRP, propose a three-step Large Neighborhood Search (LNS) consisting on:

- **Construction heuristic**: described in 3.4
- **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 obtained by rescheduling the ship pair. 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 $\pi_{\mathrm{v_i}}$ 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 $\pi_{\mathrm{v_{i,j}}}$ due to the rescheduling of $(s_i,\ s_j)$ is then estimated as the greatest of $\pi_{\mathrm{v_i}}$ and $\pi_{\mathrm{v_j}}$.

Ropke and Pisinger [10], for the PDPTW, propose a LNS similar to the one proposed by Paul Shaw [7]. 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 proposes only one removal and reinsertion heuristic, while here several are used and the selection mechanism uses statistics gathered during the search.
- Shaw proposes an exact method 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 $\mathrm{f_{-i}(s)}$ 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 $\Delta f_{i,k}$ be the cost of inserting request i at position *k*. Then let $c_i = min_k\{\Delta f_{i,k}\}$ 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 $f - \Delta f_{\mathrm{i,k}} + \Delta f_{\mathrm{i,h}}$ 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 $x_{ik} \in 1, ..., m$ be a variable that indicates the route for which request i has the kth lowest insertion cost, then $c^*_{\mathrm{i}} = \Delta f_{i,x_{ik}} - \Delta f_{i,x_{ik-1}}$. This means that the regret value is the difference in the cost of inserting the request in its kth best route and its *k-1*th best route. This heuristic then chooses the request with maximal regret value.

Aksen, Kaya, Salman and Tuncel [11], for solving the SPIRP for recovery and reuse of waste oil, propose an Adaptive LNS that consists of several destroy and repair moves.
Moves of the algorithm: Several moves are performed $\rho \in [1, 2, 3]$ times with probability *prob*$(\rho) = \{5/9, 3/9, 1/9\}$. 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: all repair moves have two common steps:

- **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.

The repair moves proposed are the following:

- **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 **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$t$

3.6.2 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. Furthermore, solutions are built in a way that the following holds: given a solution s and a shift $s_i \in \mathbf{Shifts(s)}$, a perturbation caused by a neighborhood operator applied on shift s_i will only affect the shifts $s_{\mathrm{i}}, ..., s_{|\mathrm{Shifts(s)}|}$ while the previous ones will remain unchanged. This will significantly speed-up the search in the space of neighboring solutions.
The neighborhood operators proposed for the Air Liquide IRP are the following:

- **Exchange operator**: two contiguous operations in the same shift are exchanged.
- **Insert operator**: a new operation is inserted in a shift at a certain position.
- **Remove operator**: an operation is removed from a shift.
- **Refuel operator**: a different inventory policy 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 **Compute Step** function that applies the operator (perform the move):
  - update all 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 using the *Construct Solution* function.
  - extend the solution wusing the *Extend Solution* function.
  - evaluate the neighboring solution using a 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 unchanged.

This operator has the following properties:

- **Commutativity**:

$$Exchange(s, o_1, o_2) = Exchange(s, o_2, o_1), \quad o_1, o_2 \in Operations(s), s \in Shifts$$

- **Symmetry**:

$$Exchange(Exchange(s, o_1, o_2), o_1, o_2) = s, \quad o_1, o_2 \in Operations(s), s \in Shifts$$

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 unchanged.

This operator has the following properties:

- **Idempotence**:

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

Given customers p and q, the operation inserted between them will involve the customer r such that the traveling time for the route (p, r, q) is minimized. Customer r

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 rebuilds a solution, starting from any shift, using a certain inventory policy. The previous operations in the shifts will remain the same, while all the following will be discarded (and then rebuilt, as described previously, using the Compute Step function with 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 trailer has to go back to the source to refuel.
- *deliveredQuantityFactor*: controls the quantity of product delivered to a customer.

3.7 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.

3.7.1 Delta evaluation function

Considering a solution s and one of its neighboring solutions *s'*, 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.

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

3.7.2 Delta evaluation function for the Air Liquide IRP

The objective function for the Air Liquide IRP is the following:

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

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: it has been noticed that, for this kind of problem, even a small change in a single shift can strongly affect the solution. To exploit the delta evaluation function, solutions are built in a way that the following holds: given a solution s and a shift $s_c \in$ **Shifts(s)**, a change in the shift s_c will only affect the shifts $s_{\mathrm{c}}, ..., s_{\mathrm{n}}$ while the previous ones will remain unchanged.
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 during shift s
- the total cost of shift s, given by:
  - the total distance traveled by the assigned trailer during shift s
  - the total time distance traveled by the assigned driver during shift s

The cost of a shift c is $Cost(c)$, while the cost of all the shifts up to c is calculated as $\sum_{k=1}^{c} Cost(c)$. The quantity of product delivered during a shift k is $Quantity(k)$, while the quantity of product delivered during all shifts up to c is calculated as $\sum_{k=1}^{c} Quantity(k)$. These quantities are also computed when the solution is evaluated for the fist time.
A marginal cost and a marginal quantity can be defined as:

$$\Delta Cost_c(s', s) = \sum_{\substack{k \in \{c, ... n\} \\ \subseteq Shifts(s')}} Cost(k) \; - \sum_{\substack{k \in \{c, ... n\} \\ \subseteq Shifts(s)}} Cost(k)$$

$$\Delta Quantity_c(s', s) = \sum_{\substack{k \in \{c, ... n\} \\ \subseteq Shifts(s')}} Quantity(k) \; - \sum_{\substack{k \in \{c, ... n\} \\ \subseteq Shifts(s)}} Quantity(k)$$

The objective function for the perturbed solution *s'* can be rewritten as:

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

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

$$LR(s') = \frac{\sum_{\substack{k \in \{1, ... n\} \\ \subseteq Shifts(s)}} Cost(k) + \Delta Cost_c(s', s)}{\sum_{\substack{k \in \{1, ... n\} \\ \subseteq Shifts(s)}} Quantity(k) + \Delta Quantity_c(s', s)} = \frac{\sum_{k \in shifts(s)} Cost(k) + \Delta Cost_c(s', s)}{TotalQuantity + \Delta Quantity_c(s', s)}$$

- the delta evaluation functions can be computed in $O(n - (c-1))$.
- The summation and the total quantity are already available from the starting solution. Hence, they can be computed in O(1).

3.7.3 Penalty for unfeasible solutions

Notice that the objective function does not consider the feasibility of a solution. This means that an unfeasible solution can still have a better objective function value than a feasible one. It is 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 penalty factor will be calculated as:

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

The objective function is then calculated as:

$$LR_{penalty}(s) = \begin{cases} LR(s) & \text{if } s\, is\, feasible \\ 1.0 + \frac{LR(s)}{penalty(s)} & \text{if } s\, is\, unfeasible \end{cases}$$

This way an unfeasible solution will always have a worse objective function value than a feasible solution.
The penalty will depend on how soon one of the customers is running out of product. If the search starts from an unfeasible solution, it will try to push the run out as later as possible, possibly reaching a feasible one.

3.8 Some considerations on complexity

Given a solution s and a shift c, applying a neighborhood operator to shift c and calculating the objective function of the new solution s' obtained is:

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

- Worst case scenario, the first component has been perturbed and the solution is completely changed: c = 1
$$O(n - 0) \simeq O(n)$$
- Average case scenario : $c = \frac{1}{2}n$
$$O(n - (\frac{1}{2}n - 1)) \simeq O(\frac{1}{2}n - 1) \simeq \frac{1}{2}O(n)$$
- Best case scenario, only the last component has been perturbed: c = n
$$O(n - (n - 1)) \simeq O(1)$$

Chapter 4

Automatic tuning and configuration

This chapter provides an understanding of automatic tuning and configuration of metaheuristics. The first section gives a brief introduction. The second section describes some of the techniques presented in the literature. The third section describes instead the Irace package, developed at Iridia, that has been used for this work.

4.1 The advantage of automatic tuning and configuration

The behavior of high-performance algorithms is heavily affected by their parameters, especially for search based algorithms that make use of randomization. Automatic tuning and configuration techniques are practical for several reasons [12]:

- **Development of complex algorithms**: had-hoc tuning and configuration is a time-consuming task that requires expertise. The use of automatic tuning and configuration techniques can lead to significant time savings and better results.
- **Evaluation and comparison of algorithms**: sometimes one algorithm outperforms another not because it is fundamentally superior but because its parameters were better optimized.
- **Practical use of algorithms**: end users have often little or no knowledge about the impact of parameters on the performance of an algorithm. Automatic tuning and configuration techniques can be used to improve performance in a simple and convenient way.

Notice that the optimization of an algorithm's performance by setting its (usually numerical) parameters is often known as **parameter tuning**, while the design of algorithms by means of choosing what heuristic or component to use is often known as **algorithm configuration**. However, categorical parameters can be used to select and combine the components of an algorithm. So the more general term "Automatic configuration" will be used to address the automatization of these techniques.

4.2 Automatic configuration in literature

Ropke and Pisinger [10], for the PDPTW, propose a roulette wheel selection mechanism to dynamically choose what heuristics to use during the search phase. 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 heuristics are selected independently from remove heuristics. The weights are set to zero and are increased by different values 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 balance intensification and diversification. Weight are updated using 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 [11], for solving the Selective and Periodic Inventory Routing Problem (SPIRP) for recovery and reuse of waste oil, propose a similar approach to the one used by Ropke and Pisinger.

4.3 The Irace Package

Irace is a tool that allows to automatically configure optimization algorithms, given a set of tuning instances of an optimization problem. Irace performs an *offline configuration*, that is usually addressed by algorithm designers with a trial-and-error approach by running candidate algorithms 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 several instances.

4.3.1 The algorithm configuration problem

An algorithm is configurable if it 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. 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, $d \in \{1, ..., N^{param}\}$, 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 can 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.
A usual definition of c_θ is 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 drawn, by evaluating an algorithm configuration on instances sampled from I.

4.3.2 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
```

4.3.3 The irace package

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 the configuration of irace.

Irace uses the **hookRun** to apply a certain configuration θ to an instance i and return the corresponding cost value $c(\theta, i)$. See figure 4.1 for a schematic description.

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>)**. These 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.

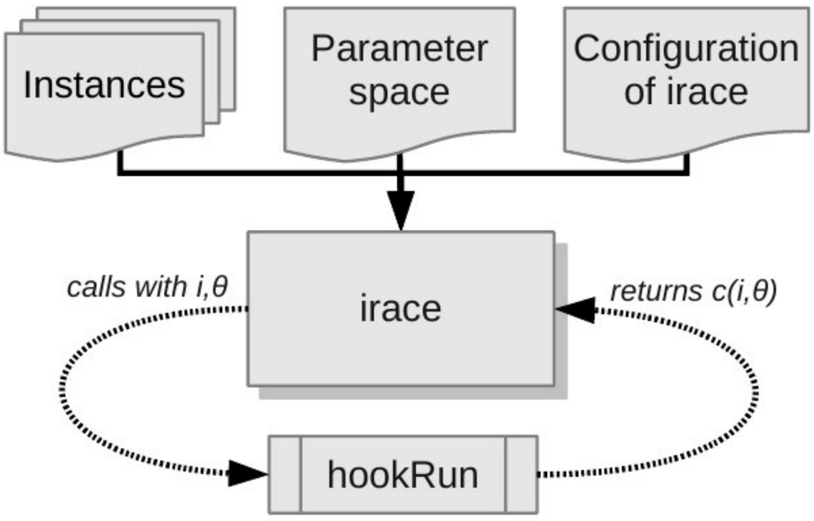


Figure 4.1: Scheme of **irace** flow information

Chapter 5

Experimental analysis

This chapter presents the experimental analysis for this thesis work. The first sections shows instance characteristics and some general aspects of the setup (software and hardware used). Then the results obtained using the Irace package will be showed, followed by statistical hypothesis tests. At the end, a sensitivity analysis is performed to investigate the connection between parameters and the behavior of the algorithm.

5.1 Instances

Size and complexity of an instance mainly depend on the number of customers and the horizon, but also on the number of drivers and trailers. See table 5.1 for details. The instances are somehow sorted according to their size in an ascending order.

Table 5.1: Instances characteristics

	Customers	**Drivers**	**Trailer**	**Horizon**
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

5.2 Software and hardware

Since they are problem dependent, some building blocks have been developed during this work and provide the following algorithmic components: initial solution methods, neighborhood operators and perturbations methods. Some general building blocks were already available and provided the following algorithmic components: metaheuristics, local searches, acceptance critera and termination criteria.
The Irace package, written in R and available in the CRAN package, has been used to configure and tune the algorithm.

The experimental analysis have been performed on rack 5 of the Iridia Cluster: it consists of 4 computational nodes, each having the following characteristics:

- 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 cluster runs the Linux distribution Rocks 6.2, based on CentOS 6.2.

5.3 Automatic tuning and configuration with Irace

The algorithmic components used to build the algorithm for the Air Liquide IRP are the following:

- **SLS algorithm**:
  - *Iterated Local Search (ILS)*: iteratively apply a local search and use restart or perturbation to escape from local optima.
  - *Variable Neighborhood Descent (VND)*: a local search that switches 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**
  - *greedy*: a constructive heuristic that builds an initial solution using a greedy function. First, customer belonging to DC are served according to one of the two greedy functions selected. Once all demands are satisfied, additional operations are added in order to improve the solution quality.
  - *greedy randomized*: construct several initial solutions as described for *greedy*. Then, it randomly selects one of them according to a probability distribution that depends on their objective function value.
  - *partial greedy randomized*: constructs a solution adding one component at time, starting from an empty solution. It builds several candidate components using the same heuristics as *greedy*. Then, it randomly adds one of the components according to a probability distribution that depends on the improvement in the objective function value.
- **Neighborhood**
  - *exchange*: two contiguous operations in the same shift are exchanged.
  - *insert*: a new operation is inserted in a shift at a certain position.
  - *remove*: an operation is removed from a shift.
  - *refuel*: a different inventory policy is used.

  Notice that ILS can choose on of the neighborhoods available, while VND can chose many in a certain order.

- **Perturbation**:
  - *random move*: a certain number of random search steps in one of the neighborhoods defined are performed.
- **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.

Irace has been run with different budgets and computation time limits to see what are the most chosen algorithmic components. See Table 5.2 for details.

Table 5.2: Irace settings

	Budget	**Computation time limit**
configuration_1	5000	300
configuration_2	5000	900
configuration_3	20000	300
configuration_4	20000	900

All configurations obtained using Irace with different settings are shown in Table 5.3. It can be noticed that:

- ILS is always chosen as SLS algorithm.
- first improvement is chosen over best improvement as pivoting rule.
- local minimum is always chosen as termination criterion.
- greedy is chosen in most cases as initial solution, while partial greedy randomized is not chosen at all.
- exchange and remove are the only operators chosen.
- simulated annealing is the most chosen acceptance criterion.

Table 5.3: Configurations obtained for different run of Irace

	configuration_1	**configuration_2**	**configuration_3**	**configuration_4**
ILS	ILS	ILS	ILS	ILS
LS	First	First	First	First
Termination	Locmin	Locmin	Locmin	Locmin
Initial solution	Greedy	Greedy	Greedy randomized	Greedy
Neighborhood	Remove	Remove	Exchange	Exchange
Perturbation	Exchange	Remove	Remove	Remove
Acceptance	SA	Diversify	SA	SA

As expected, the configuration_4 showed the best performance and it will be selected for the next experiments. It will be compared to two other configurations. They are all listed as follows:

- **random**: algorithmic components and parameters have been selected randomly.
- **arbitrary**: algorithmic components have been selected ad-hoc, based on personal experience and trial-and-error runs.

- **irace**: configuration_4 obtained using the Irace package.

The different component chosen for each configuration are shown in table 5.4.

Table 5.4: Components chosen for each configuration

	random	**arbitrary**	**configuration_4**
ILS	ILS	ILS	ILS
LS	First	Best	First
Termination	Locmin	Locmin	Locmin
Initial solution	Greedy	Greedy	Greedy
Neighborhood	Exchange	Exchange	Exchange
Perturbation	Remove	Insert	Remove
Acceptance	Metro	Metro	SA

Each component has a set of parameters that can be tuned by irace. They are shown in Table 5.5 and described in detail in sections 3.4.4, 3.5.2 and 3.6.2.

Table 5.5: Parameters used for the different configurations

	random	**arbitrary**	**irace**	*range*
Initial Solution				
randomFactor	0.9274	0.6	0.9604	[0.0, 1.0]
urgencyPolicy	1	1	2	[1, 2]
Neighborhood				
randomFactor	0.6843	0.6	0.2972	[0.0, 1.0]
urgencyPolicy	1	1	1	[1, 2]
refuelFactor	0.3037	0.0	0.1232	[0.0, 1.0]
deliveredQuantityFactor	0.9652	1.0	0.9932	[0.0, 1.0]
Perturbation				
randomFactor	0.4813	0.6	0.4684	[0.0, 1.0]
urgencyPolicy	2	1	1	[1, 2]
refuelFactor	0.2198	0.0	0.6678	[0.0, 1.0]
deliveredQuantityFactor	0.7388	1.0	0.6928	[0.0, 1.0]
steps	7	3	1	[1, 10]
Acceptance				
start	1.0477	3.5	3.8107	[1.0, 5.0]
end	0.6775	-	0.0429	[0.0, 1.0]
ratio	0.0344	-	0.0556	[0.01, 0.1]

5.3.1 Experiments

The three configurations shown in table 5.4 have been tested and compared to check how the best configuration found by Irace can improve performance. All experiments have been run in parallel and on a different CPU. Each run has been repeated 15 times for each instance with a cut-off time of 30 minutes. The performance of each configuration have been measured, calculating the Relative Percentage Deviation (RPD) with respect to the best solution known for each instance.
Consider that a much higher objective function value with respect to the optimal values known is assigned to unfeasible solutions. This means that their RPD will be very close to one.

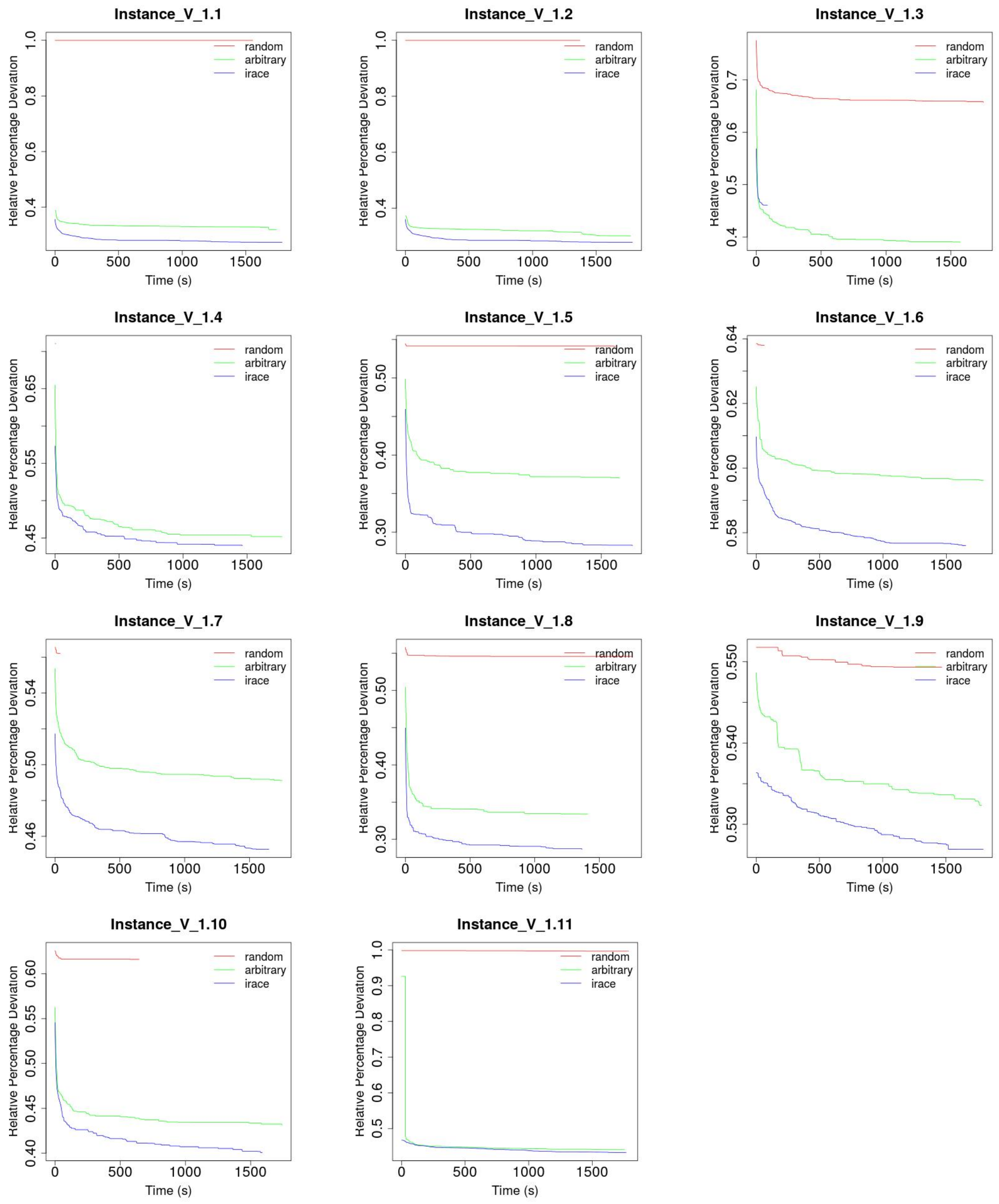


Figure 5.1: Performance plots for random, arbitrary and irace configuration, on every instance

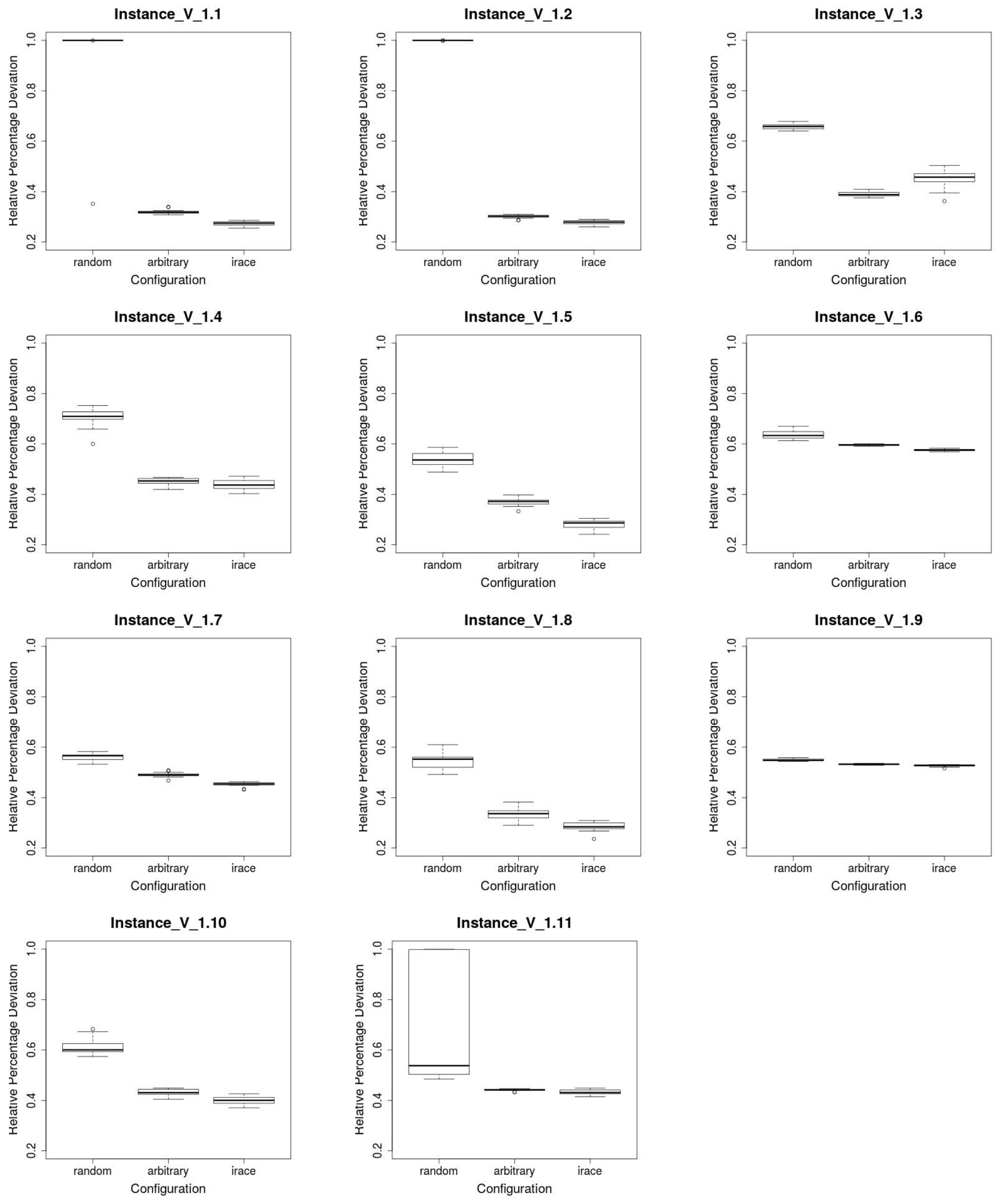


Figure 5.2: Boxplots summarizing the distribution of the final results for random, arbitrary and irace configuration, on every instance.

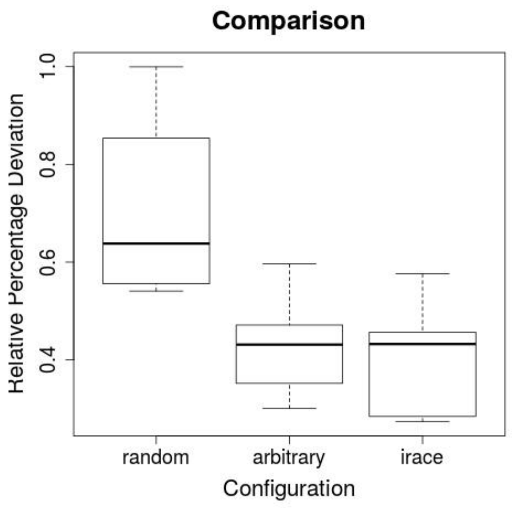


Figure 5.3: Overall performance boxplot for random, arbitrary and irace configuration

Figure 5.1 shows the behavior of the three algorithm configurations. There is one plot for every instance: it shows the run time on the x-axis and the RPD on the y-axis.
Figure 5.2 shows the distribution of the best objective function values found by the different configurations. There is one boxplot for every instance: it shows the different configurations on the x-axis and the RPD on the y-axis. A single box contains the values found by a specific configuration for the different runs (15 values in total).
Figure 5.3 shows the overall performance of the three configurations. The boxplot shows the different configurations on the x-axis and the RPD on the y-axis. A single box contains the average values found by a configuration on every instance (11 values in total).
It can be noticed that the irace configuration outperforms the random and the arbitrary ones on every instance but the third one, both in the short and the long run. For instances 5, 7 and 8 the irace configuration performs particularly better. They are all instances with a short horizon: it is possible that for a shorter horizon it is easier to achieve an improvement. Also notice that the random configuration is not able to find feasible solutions for instances 1 and 2.

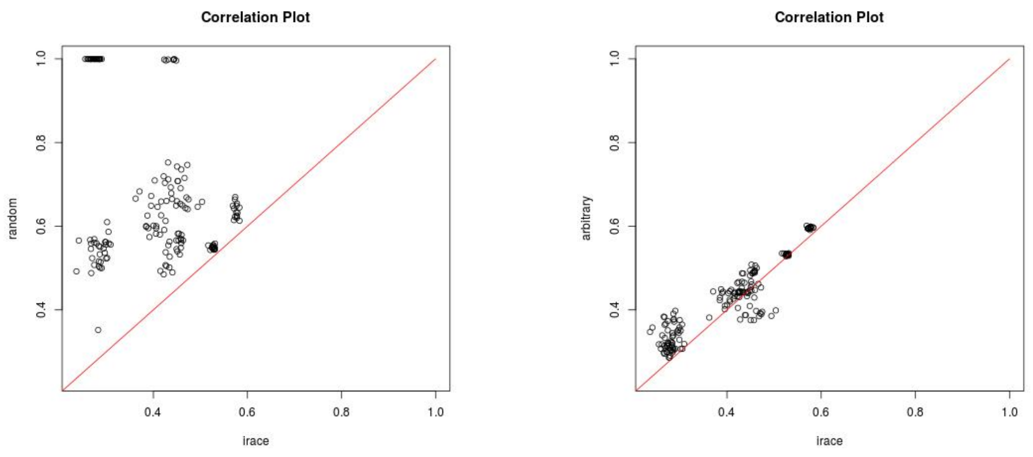


Figure 5.4: Correlation plots for the three configurations tested

Figure 5.4 show the distribution of the objective function values found for all runs. The irace configuration is compared to the random configuration and the arbitrary configuration: it shows

the RPD of the irace configuration on the x-axis and the RPD of the other configuration on the y-axis. Cluster of points represent the different runs on a specific instance. If a point lies on the red line, it means that the two different configurations found a solution of similar quality for a run on a specific instance. Points above the red line represents the runs for which irace performed better and whose values are closer to the best ones known.
When comparing the irace configuration with the random one, all points lie above the red line, meaning that the irace configuration performs significantly better than the random one. When comparing the irace configuration with the arbitrary one, instead, some points lie on the read line or below, but still the majority of them lie above, meaning that the irace configuration performes better that the arbitrary one but not on all instances and not as significantly as the first comparison.

5.3.2 Statistical hypothesis test

Statistical hypothesis allows us to make inferences about data that has been collected during the experiments: it tells if the pattern observed is real or just due to chance.
Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples. Differently from the Student test, it does not assume that the population has a normal distribution. It can be used to state, with a certain confidence, that the irace configuration performs better than random and arbitrary configurations.

A Wilcoxon test with confidence interval 95% has been performed between random configuration and irace configuration:
Null hypothesis: there is no statistically significant difference between the objective function values of the solutions found by random configuration and irace configuration.
Alternative hypothesis: there is statistically significant difference between the objective function values of the solutions found by random configuration and irace configuration.

$$\textit{p-value} = 1.16e - 28$$

Since *p-value* < 0.05, the null hypothesis can be strongly refused.

Applying the same test to the arbitrary and the irace configuration gives a p-value of $4.68e - 14$ which means that the difference between the two configurations is statistically significant.
So it is possible to state that the irace configuration performs significantly better that the random and the arbitrary one.

5.4 Sensitivity analysis

Parameters can significantly affect the performance of an algorithm. The **one-at-a-time** method will be used for the sensitivity analysis: by varying the irace configuration with respect to a parameter at time, the algorithm's sensitivity to changes in this parameter will be investigated. The expected result is that the best parameter value observed during this analysis should match or be close to the parameter values used by the **irace** configuration.
The configuration that has been chosen for the sensitivity analysis is the irace configuration because we are interested in the sensitivity of the optimal configuration to changes in its parameters and around the optimal values.

The first parameter investigated was *deliveredQuantityFactor*. It controls the quantity of product delivered to a customer. When equal to zero, no product is delivered at all. When equal to one, the maximum amount of product is delivered.

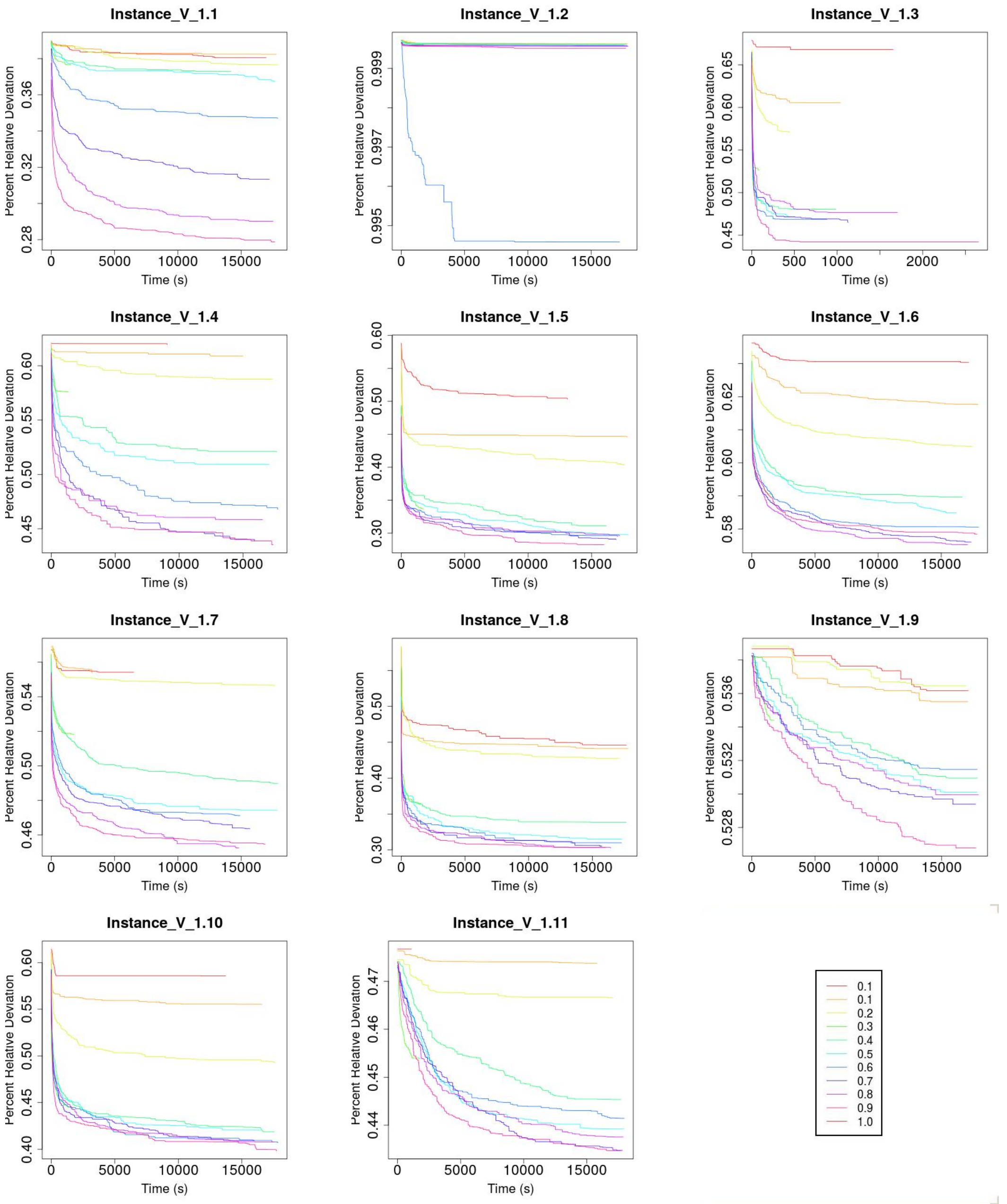


Figure 5.5: Performance plots for the irace configuration using different values for *deliveredQuantityFactor*, on every instance

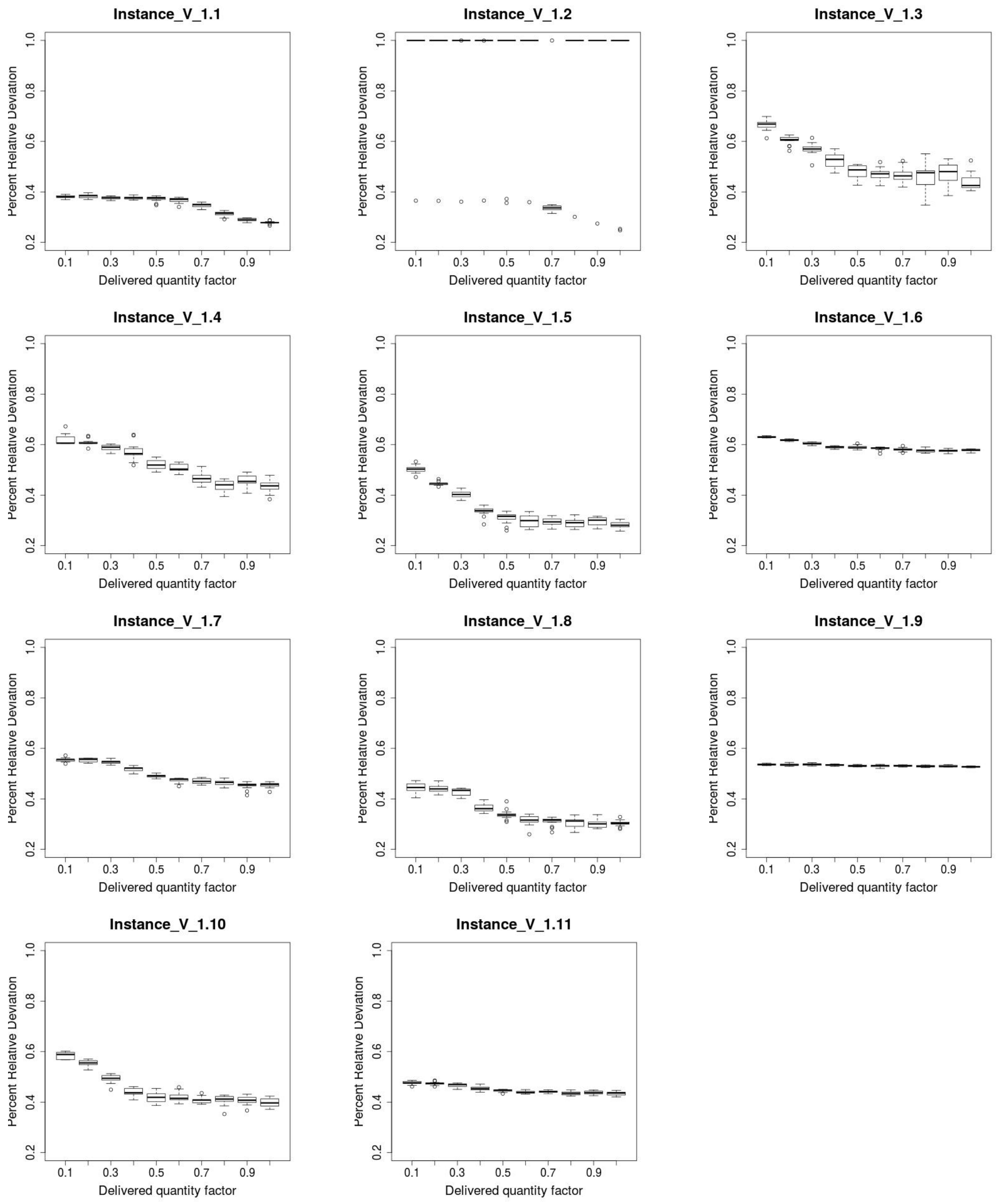


Figure 5.6: Boxplots summarizing the distribution of the final results for the irace configuration, using different values for *deliveredQuantityFactor*, on every instance

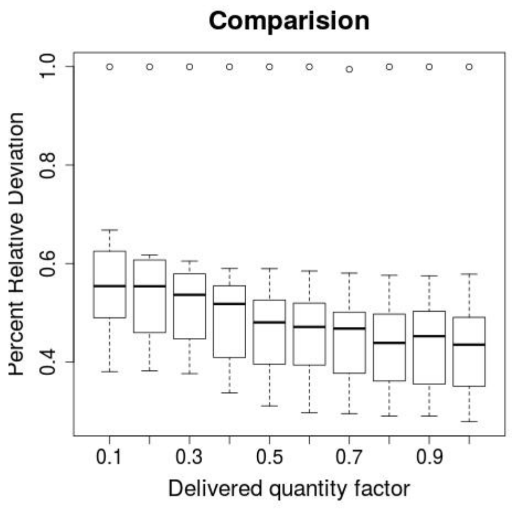
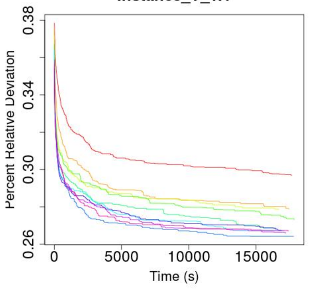
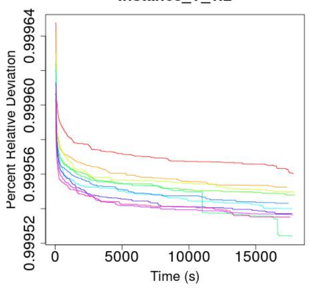


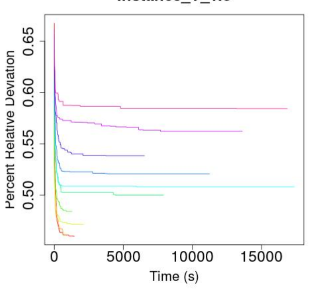
Figure 5.7: Overall performance boxlot for the irace configuration using different values for *deliveredQuantityFactor*

Figure 5.5 shows the behavior of the irace configuration for different values of *deliveredQuantity-Factor*. There is one plot for every instance: it shows the run-time on the x-axis and the RPD on the y-axis.
Figure 5.6 shows the distribution of the objective function values found by the irace configuration for different values of *deliveredQuantityFactor*. There is one boxplot for every instance: it shows the *deliveredQuantityFactor* on the x-axis and the RPD on the y-axis. A single box contains the objective function values found using a specific value of *deliveredQuantityFactor* during the different runs (15 values in total).
Figure 5.7 shows the overall performance of the irace configuration using different values of *deliv-eredQuantityFactor*. The boxplot shows the *deliveredQuantityFactor* on the x-axis and the RPD on the y-axis. A single box contains the average objective function values found using a specific value of *deliveredQuantityFactor* on every instance (11 values in total).
It can be noticed that the algorithm performs better for high values of the parameter. In fact, Irace chose a values of 0.9932.
For instances 3, 4, 5, 7, 8 and 10 the algorithm seems more sensitive to the parameter because it shows larger changes in the performance. They are all instances with a short horizon: it is possible that for a shorter horizon this parameter has a stronger impact. For instance 9 there is no much change, a possible reason is that it is the biggest instance available.

The second parameter investigated was *refuelFactor*. It determines when a trailer has to go back to the source to refuel. When equal to zero the trailer goes back to source only when completely empty, when equal to one the trailer goes back to the source after each delivery.







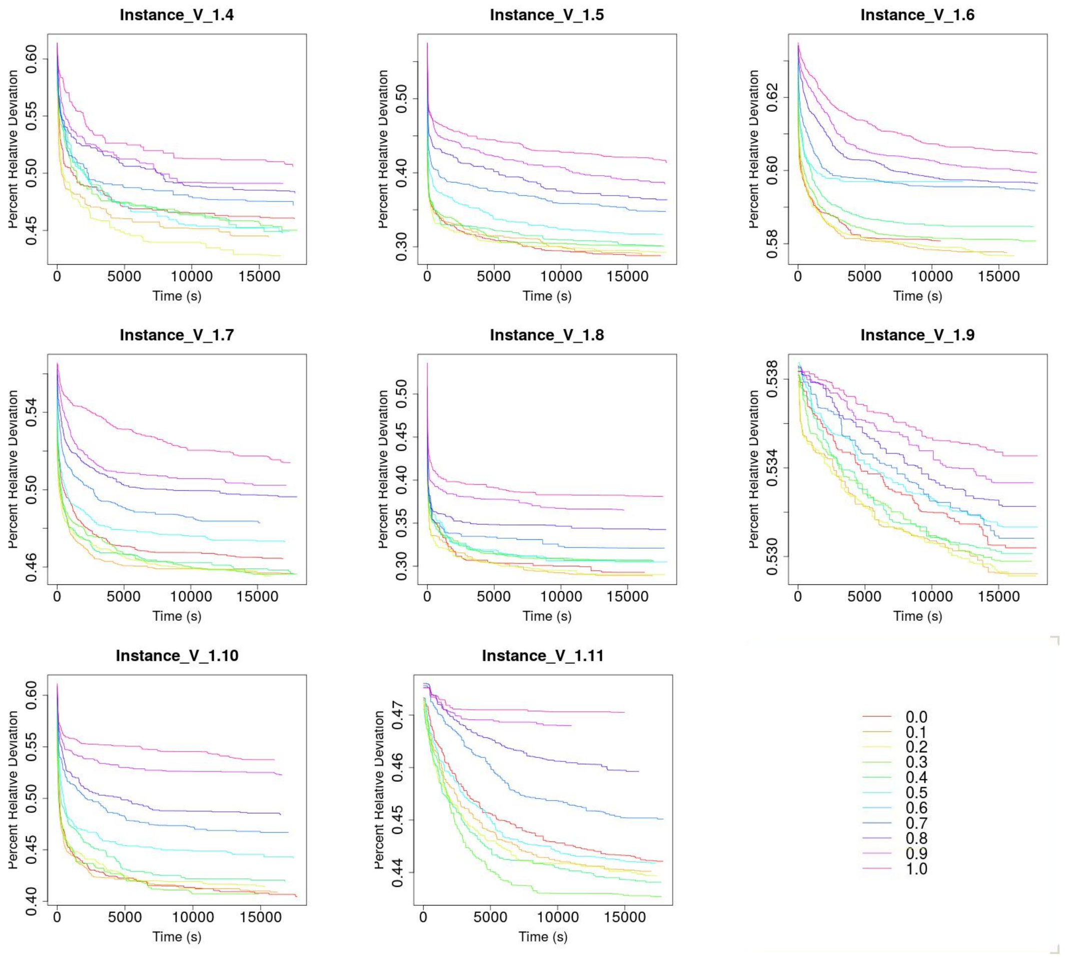
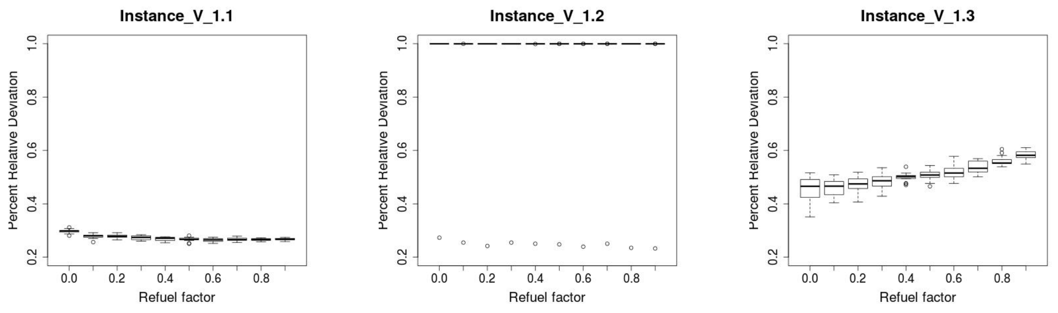


Figure 5.8: Performance plots for the irace configuration using different values for *refuelFactor*, on every instance



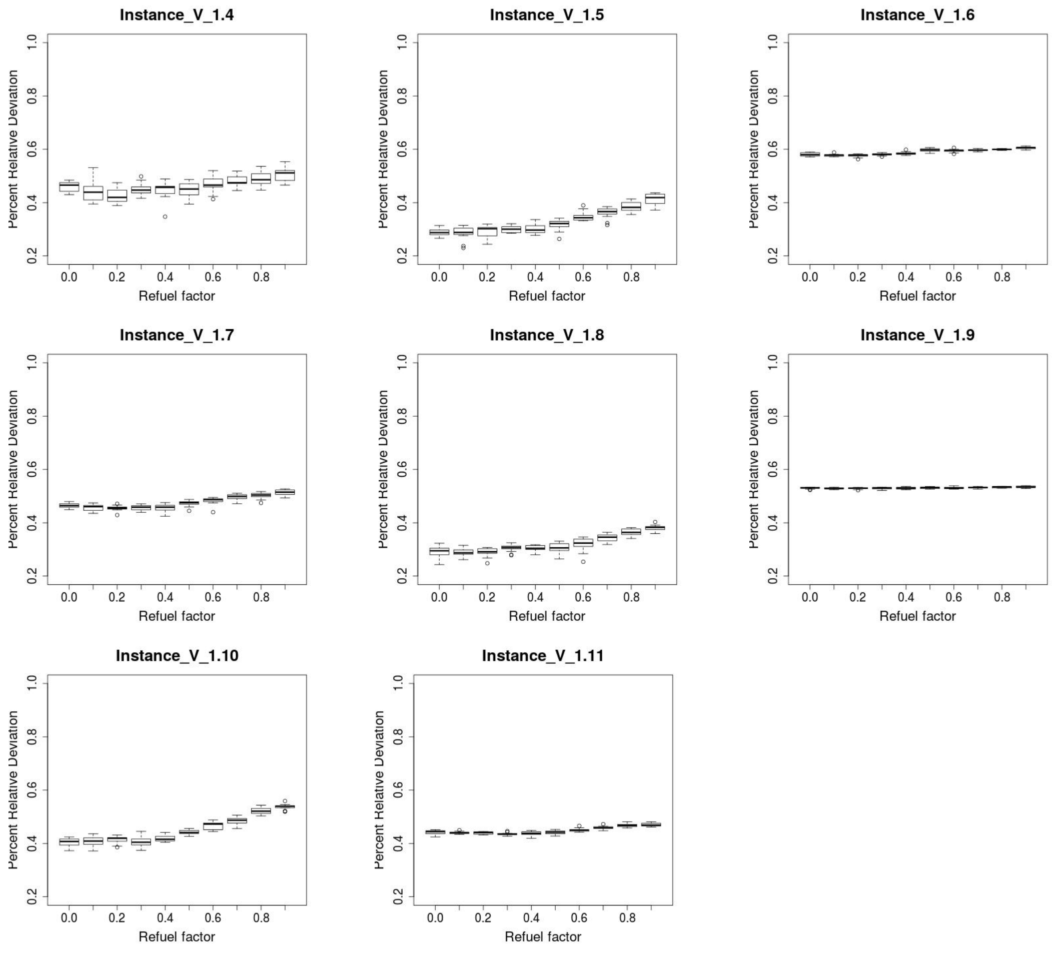


Figure 5.9: Boxplots summarizing the distribution of the final results for the irace configuration, using different values for *refuelFactor*, on every instance

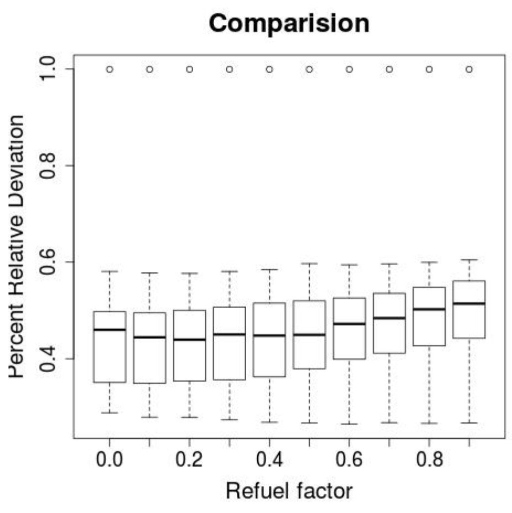
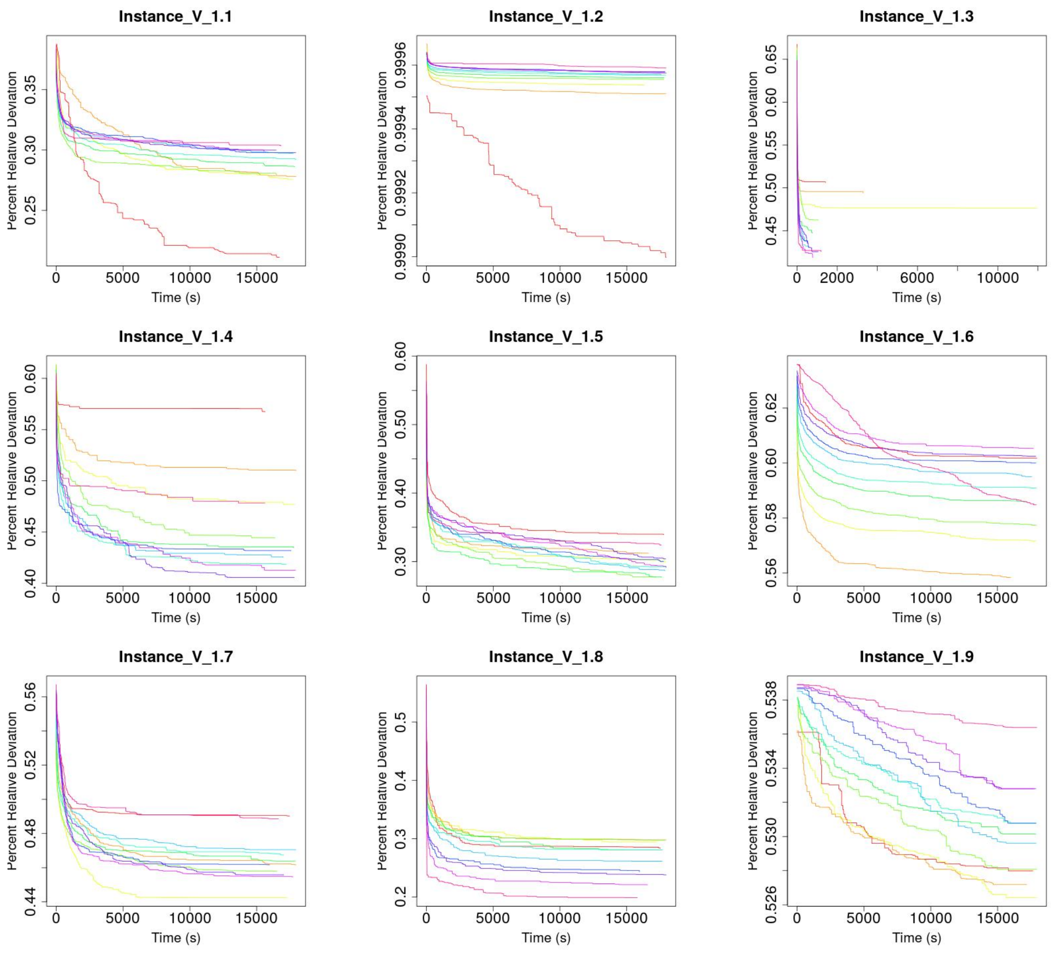


Figure 5.10: Overall performance boxplot for the irace configuration using different values for *refuelFactor*

Figure 5.8 shows the behavior of the irace configuration for different values of *refuelFactor*. There is one plot for every instance: it shows the run-time on the x-axis and the RPD on the y-axis.
Figure 5.9 shows the distribution of best objective function values found by the irace configuration for different values of *refuelFactor*. There is one boxplot for every instance: it shows the *refuelFactor* on the x-axis and the RPD on the y-axis. A single box contains the objective function values found using a specific value of *refuelFactor* during the different runs (15 values in total).
Figure 5.10 shows the overall performance of the irace configuration using different values of *refuelFactor*. The boxplot shows the *refuelFactor* on the x-axis and the RPD on the y-axis. A single box contains the average objective function values found using a specific value of *refuelFactor* on every instance (11 values in total).
It can be noticed that the algorithm performs better for low values of the parameter, and a value of 0.2 or 0.3 seems optimal. Irace actually chose a values of 0.1231. This is one of the advantages of Irace with respect to the one-at-a-time sensitivity analysis that is not able to detect optimal values in between the range step. Again, the algorithm seems more sensitive to the parameter on instances with a short horizon and less sensitive for larger ones like instance 6, 9 and 11.

The third parameter investigated was *randomFactor*. It controls the degree of randomness introduced in the algorithm.



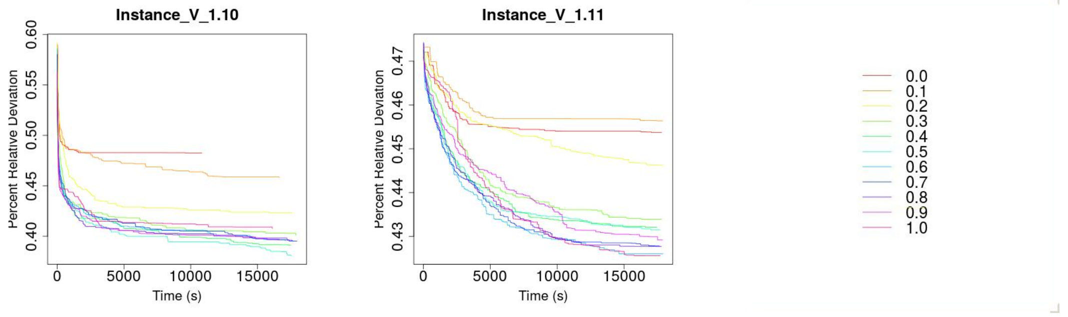
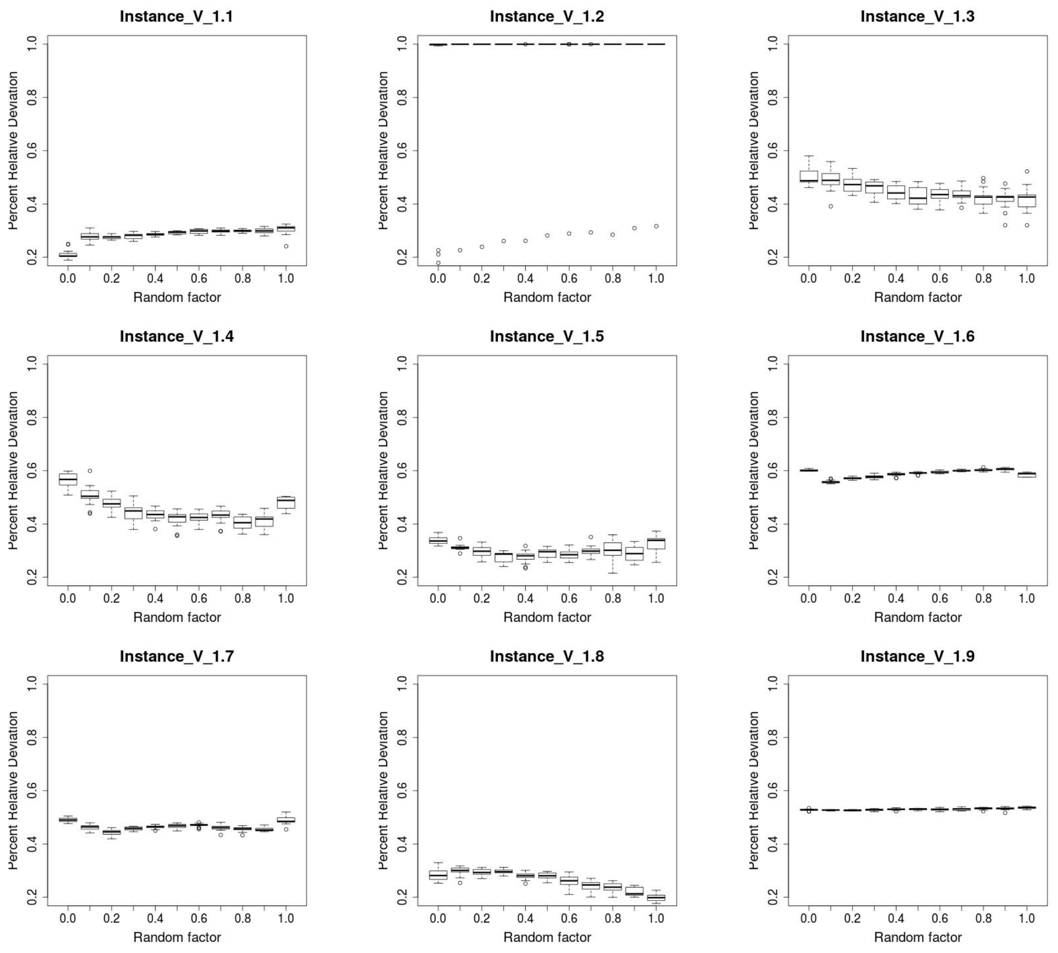


Figure 5.11: Performance plots for the irace configuration using different values for *random-Factor*, on every instance



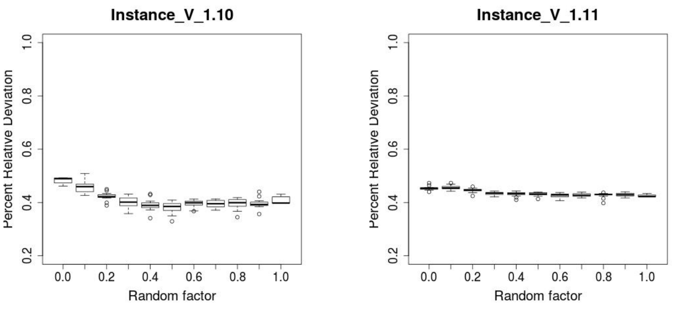


Figure 5.12: Boxplots summarizing the distribution of the final results for the irace configuration, using different values for *randomFactor*, on every instance

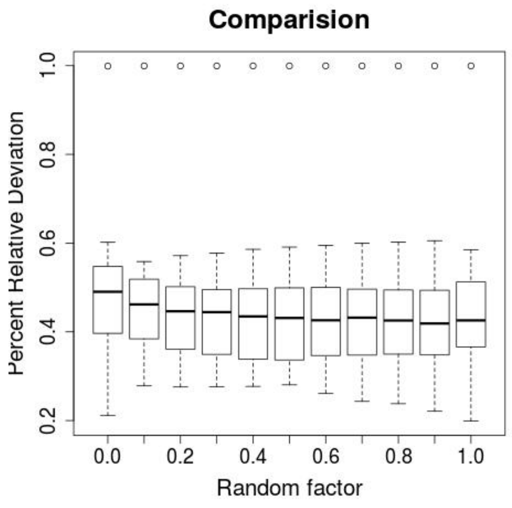


Figure 5.13: Overall performance boxplot for the irace configuration using different values for *randomFactor*

Figure 5.11 shows the behavior of the irace configuration for different values of *randomFactor*. There is one plot for every instance: it shows the *randomFactor* on the x-axis and the RPD on the y-axis. Figure 5.12 shows the distribution of the best objective function values found by the irace configuration for different values of *randomFactor*. There is one boxplot for every instance: it shows the *randomFactor* on the x-axis and the RPD on the y-axis. A single box contains the objective function values found using a specific value of *randomFactor* during the different runs (15 values in total).
Figure 5.13 shows the overall performance of the irace configuration using different values of *randomFactor*. The boxplot shows the *randomFactor* on the x-axis and the RPD on the y-axis. A single box contains the average objective function values found using a specific value of *randomFactor* on every instance (11 values in total).
For this parameter the behavior of the algorithm is more complex, it can be noticed that the algorithm performs better for low values on some instances and high values for some others. Irace chose a value of 0.2972.
Instances 6 and 9 still result the ones with fewer changes.

The last parameter investigated was actually a categorical parameter. It determines what pivoting rule to use: best improvement selects the neighboring solution that achieves a maximal improvement, while first improvement selects the first neighboring solution that achieves an improvement.

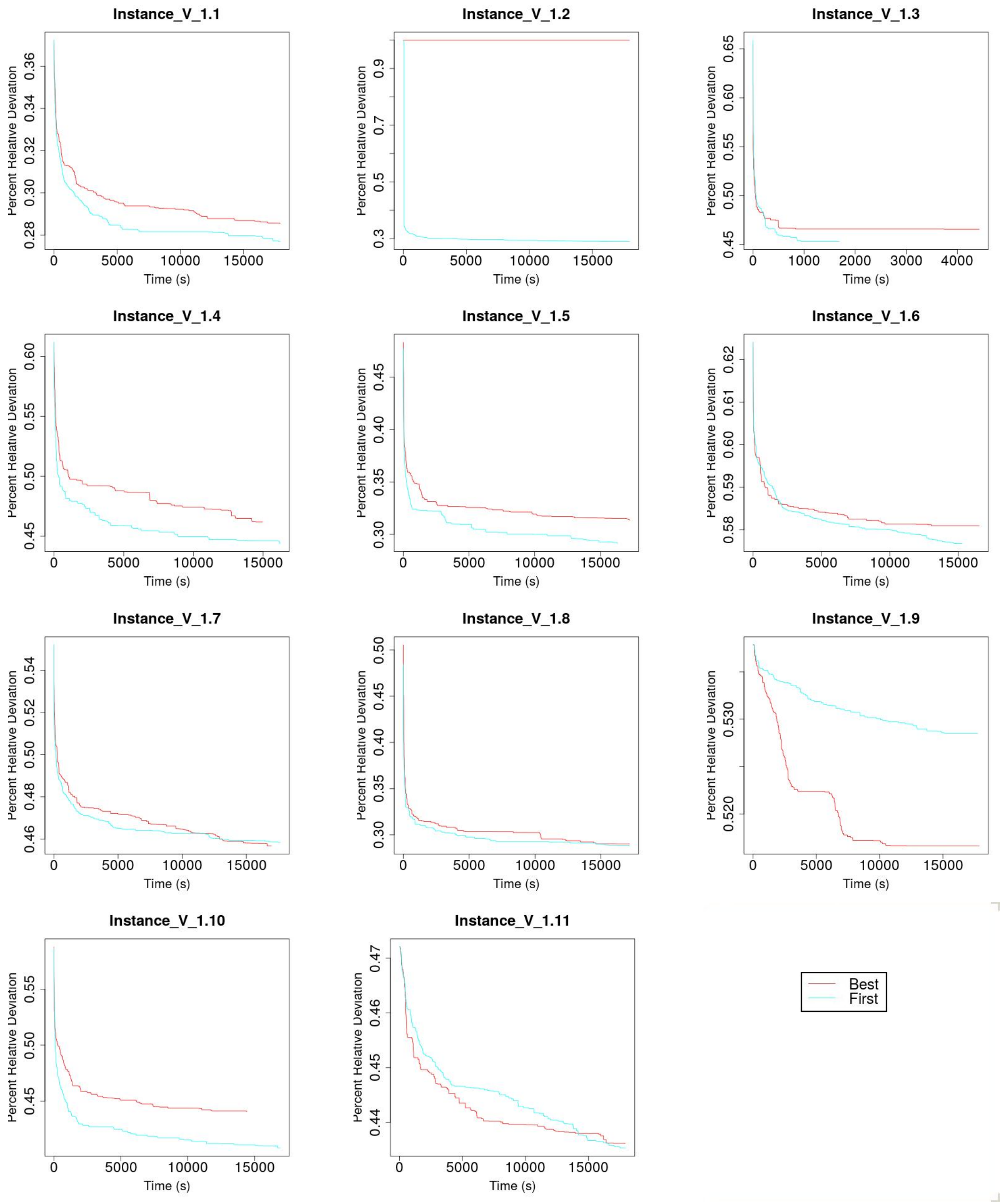


Figure 5.14: Performance plots for the irace configuration using different pivoting rules, on every instance

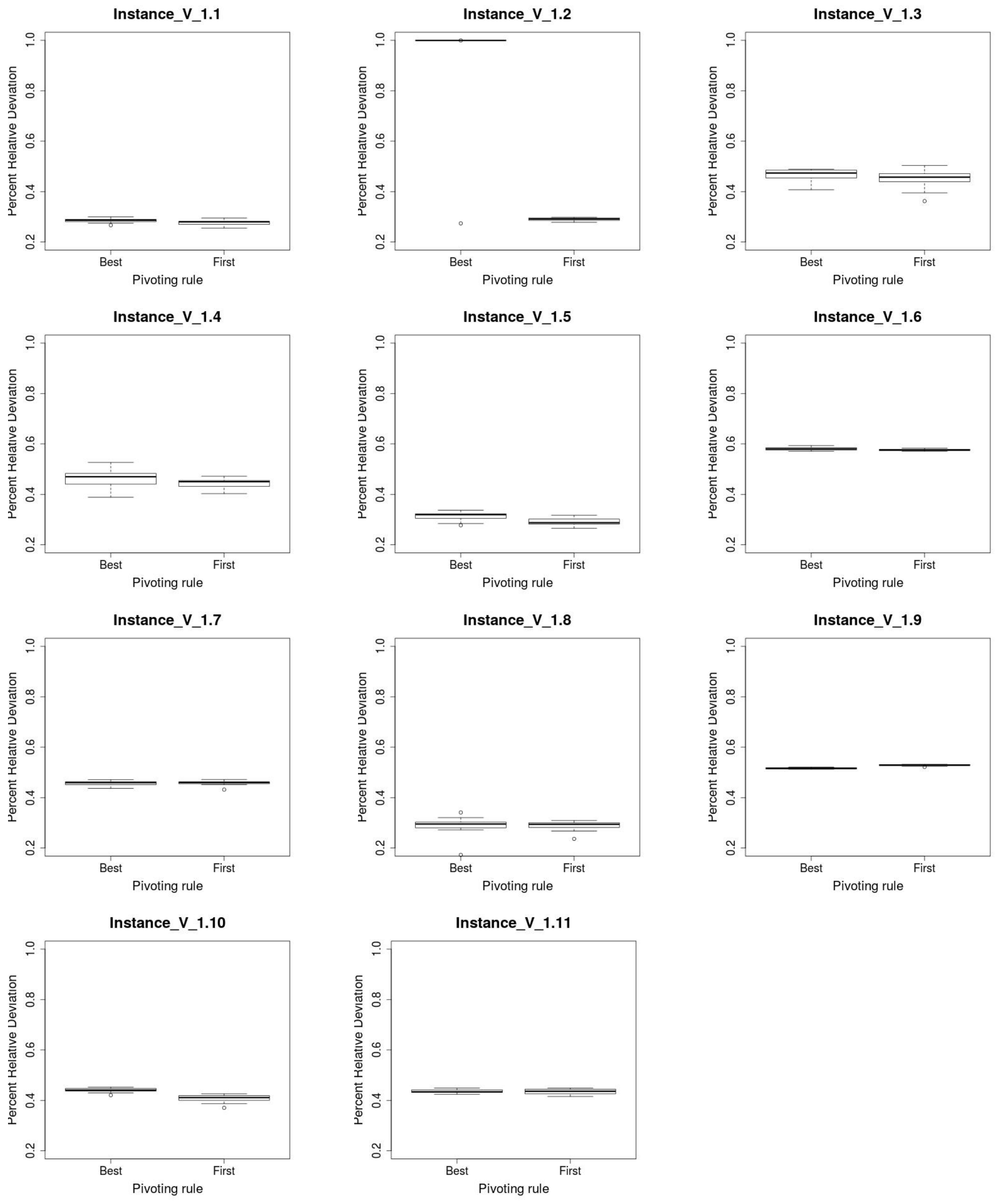


Figure 5.15: Boxplots summarizing the distribution of the final results for the irace configuration, using different pivoting rules, on every instance

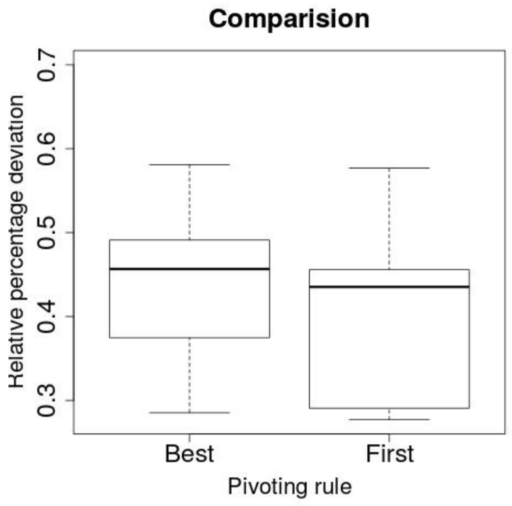


Figure 5.16: Overall performance boxplot for the irace configuration using different pivoting rules

Figure 5.14 shows the behavior of the irace configuration for different pivoting rules. There is one plot for every instance: it shows the pivoting rule on the x-axis and the RPD on the y-axis.
Figure 5.15 shows the distribution of the best objective function values found by the irace configuration for different pivoting rules. There is one boxplot for every instance: it shows the pivoting rule on the x-axis and the RPD on the y-axis. A single box contains the objective function values found using a specific pivoting rule during the different runs (15 values in total).
Figure 5.16 shows the overall performance of the irace configuration using different pivoting rules. The boxplot shows the pivoting rule on the x-axis and the RPD on the y-axis. A single box contains the average objective function values found using a specific pivoting rule on every instance (11 values in total).
For this parameter it can be noticed that the algorithm performs better using the first improvement as pivoting rule: the overall performance are not significantly different, since the boxplots overlap on most instances, but only the first improvement is able to find feasible solutions for the second instance. In fact, Irace chose the first improvement. Also, for instances 6, 7, and 11 best improvement seems to perform better on the short run, but first improvement outperforms it on the long run.

5.5 Considerations on the results

The Irace package was used to find the best configuration and the best set of parameters.
The algorithm obtained was compared to a random configuration and an ad-hoc configuration based on personal experience. A statistical analysis test was performed to state that, with statistical significance, the irace configuration had better performance than the random and the arbitrary one.
A sensitivity analysis was also performed to investigate the sensitivity of the irace configuration to changes in its parameters. The analysis shown that the right choice of parameters can affect significantly the performance of the algorithm and that the Irace package was able to find the best values for the set of parameters.

Chapter 6

Conclusions and future works

6.1 Conclusions

This work presented the automatic configuration and tuning of metaheuristics for the Air Liquide IRP, a problem proposed by the French company Air Liquide for the 2016 EURO-ROADEF Challenge. A review of the literature was useful to find out how similar IRPs were modelled and the different techniques used to solve them. The Air Liquide IRP, then, was modelled: it is a real world problem that is significatively more complex than a general IRP and the IRPs that have been reviewed during this work, since it includes more constraints related to vehicle routing and inventory management.
For this problem some metaheuristic components were defined, such as functions to generate initial solutions, neighborhood operators and perturbation methods. A delta evaluation function was also defined to reduce the complexity for computing the objective function value of a solution. Since the objective function proposed took into account only the quality of a solution and not its feasibility, a variant was proposed that penalizes unfeasible solutions.
The Irace package was used to select the best components developed, select the best performing configuration of the algorithm and the best set of parameters. The optimal configuration has been compared to a random configuration and a ad-hoc configuration, and statistical analysis has been used to prove that the optimal configuration outperforms them. A sensitivity analysis was performed to show the algorithm's behaviour with respect to some of its parameters.

6.2 Future works

The Irace Package proposes performs a static configuration, meaning that the best configuration of the algorithm and the best set of parameters are computed offline on a set of instances. Some of the papers reviewed performs it online, meaning that the value of parameters changes over time, but they only take into account one parameter a time. It may be effective to exploit the best of both methods: an online configuration that tunes more than one parameter at time. This way the best interaction between parameters can be found.
For this work ILS and VNS were used as SLS for the algorithm configuration. Other SLSs can be developed. According to the literature reviewed, LNS may perform particularly well with this problem.
The objective function has been defined in a way that a feasible solution has always a better objective function value that an unfeasible solution. It may be interesting to define the objective function in a way that the opposite can also holds: this way new regions of unfeasible solutions that were unexplored can be discovered and may lead to optimal solutions.

Bibliography

[1] Holger H. Hoos, Thomas Stützle, *Stochastic Local Search - Foundations and Applications*, Elsevier, 2004.

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

[3] Leandro C. Coelho, Jean-FranÃğois Cordeau, Gilbert Laporte, *Thirty Years of Inventory-Routing*, Transportation Science 48.1 (2013): 1-19.

[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*, Vol. 197. Berlin: Springer, 2009.

[6] Claudia Archetti, Luca Bertazzi, Alain Hertz, M. Grazia Speranza *A hybrid heuristic for an inventory routing problem.*, INFORMS Journal on Computing 24.1 (2012): 101-116.

[7] Paul Shaw, *Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problem*, International Conference on Principles and Practice of Constraint Programming. Springer Berlin Heidelberg, 1998.

[8] Vikas Goel, Kevin C. Furman, Jin-Hwa Song, Amr S. El-Bakry *Large Neighborhood Search for LNG Inventory Routing*, Journal of Heuristics 18.6 (2012): 821-848.

[9] M. Chrostiansem, K. Fagerholt *Maritime Inventory Routing Problems*, Encyclopedia of optimization. Springer US, 2008. 1947-1955.

[10] Stefan Ropke, David Pisinger, *An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows*, Transportation science 40.4 (2006): 455-472.

[11] Deniz Aksen, Onur Kaya, F. Sibel Salman, Özge Tüncel, *An Adaptive Large Neighborhood Search Algorithm for a Selective and Periodic Inventory Routing Problem*, European Journal of Operational Research 239.2 (2014): 413-426.

[12] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown Thomas Stützle, *ParamILS: an automatic algorithm configuration framework*, Journal of Artificial Intelligence Research 36.1 (2009): 267-306.

List of Figures

List of Tables

Abbreviations

CVRP Capacitated Vehicle Routing Problem. 4

IRP Inventory Routing Problem. 1

LNGIRP Liquified Natural Gas Inventory Routing Problem. 5
LNS Large Neighborhood Search. 5

MIRP Maritime Inventory Routing Problem. 5
ML maximum-level policy. 3

OU order-up-to-level policy. 3

PDPTW Pickup and Delivery Problem with Time Windows. 5

RPD Relative Percentage Deviation. 42

SLS Stochastic Local Search. 17
SPIRP Selective and Periodic Inventory Routing Problem. 5

VMI Vendor Management Inventory. 6
VRP Vehicle Routing Problem. 1