Università degli Studi di Cagliari

Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Matematica e Informatica
Ciclo XXIX

Ph.D. Thesis

A metaheuristic approach for the Vehicle Routing Problem with Backhauls

S.S.D. INF/01

Candidate
Gianfranco Fadda

Supervisor
Prof. Paola Zuddas

PhD Coordinator
Prof. Giuseppe Rodriguez

Final examination academic year 2015/2016
September 26, 2017

Abstract

Despite the vivid research activity in the sector of the exact methods, nowadays many Optimization Problems are classified as *Np-Hard* and need to be solved by heuristic methods, even in the case of instances of limited size. In this thesis a Vehicle Routing Problem with Backhauls is investigated. A Greedy Randomized Adaptive Search Procedure metaheuristic is proposed for this problem. Several versions of the metaheuristic are tested on symmetric and asymmetric instances. Although the metaheuristic does not outperform the best known solutions, a large number of high-quality routes are determined in several solutions for each instance. Therefore the metaheuristic is a promising approach to generate feasible paths for set-partitioning-based formulations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, a Vehicle Routing Problem with Backhauls (VRPB) is investigated. This is a challenging Combinatorial Optimization problem, which differs from the well-known Capacitated Vehicle Routing Problem because two types of customers must be served by a given fleet of vehicles: some customers *(or Linehaul customers)* need to receive goods from a depot, other customers *(or Backhaul customers)* need ship goods to the same depot. Each customer must be visited only once. Moreover, in each route all Linehauls must be visited before all Backhauls, in order to minimize loads reshuffling in the common case of rear-door vehicles. The objective is to minimize the total routing costs.

VRP problems are *NP*-hard and there is a little hope to find optimal solution for many instances of real interest. In such cases, heuristic and metaheuristic methods are generally utilized to find high-quality solutions, even if their optimality cannot be proved. The effectiveness of these methods depends on their ability to adapt to the specific problem at hand, take advantage of its basic structure, and avoid the entrapment in a local minimum. General speaking, a relevant effort in this field is to tune in the correct way some key parameters.

Therefore, it is of interest to adopt metaheuristics where few parameters need to be set and tuned. This appealing feature can be found in GRASP (Greedy Random Adaptive Search Procedure) metaheuristics. In addition, they allow reusing some existing and efficient local search procedures and one can focus on the implementation of efficient data structures to assure quick iterations. In fact, fast iterations are a key requirement to calculate an high number of feasible solutions. The best overall solution is kept as the final outcome.

More precisely, the GRASP is a multi-start metaheuristic which consists of a constructive procedure, based on a greedy randomized algorithm combined with a local search. In this thesis, a GRASP is proposed for the VRPB. In the constructive phase, one determines which customer is the last Linehaul and the first Backhaul in each route. Next, two open routes are created from these nodes to the depot. Finally, the two open routes are merged, in order to obtain a feasible routes for the VRPB. This procedure is repeated until all nodes are included in a route and, thus, a feasible solution is obtained for the VRPB. Next, this solution is improved by a local search phase. Different sequences of neighborhoods are implemented and tested in this phase. When a local minimum is found, the construction phase is run again, a new solution of the VRPB is built and improved in the local search phase, and so on until the maximum running time is not reached.

This thesis is organized as follows. Chapter 2 describes the several variants of the Vehicle Routing Problem and focuses on the description of the VRPB. Alternative mathematical models for this problem are reported.

In chapter 3 we briefly recall the basic concepts of integer programming and on general purpose algorithms to obtain optimal solutions. The exact methods for VRPB are reviewed.

In chapter 4, the main (meta)heuristic approaches for Combinatorial Optimization Problems are presented. Special attention is devoted to heuristics and metaheuristics for the VRPB.

In chapter 5, a Greedy Random Adaptive Search Procedure metaheuristic for VRPB is presented. The algorithm is analyzed in detail using several pseudocodes, starting from an high-level point of view and describing the main procedures.

In chapter 6 the experimentation is presented. The algorithm is tested using different benchmarks: the symmetric instances proposed by Goetschalckx and Jacobs-Blecha in [GJB89] and the asymmetric instances presented in [FTV94]. Different local search techniques are used and all results are presented.

Conclusions and future research direction are shown in chapter 7

Chapter 2

The Vehicle Routing Problem with Backhauls

The Combinatorial Optimization studies the optimization problems in which the feasible set is defined in terms of combinatorial structures, including graphs that play an important role. The key feature of these problems is to handle only discrete feasible sets, unlike linear optimization in which the feasible set is continuous.

2.1 The Vehicle Routing Problem - a short description

The Vehicle Routing Problem, which will be indicated from now with the acronym VRP, is a typical Combinatorial Optimization Problem about distribution networks and consists in distribution of material goods between a depot or a set of depots and a set of customers.
This problem was introduced by Dantzig and Ramser in [DR59]; the term VRP includes an entire class of problems that has for object the study of techniques for the route planning of a fleet of vehicles, which have a distribution service of material goods, services or information between a set of stores and a set of customers.
Core is the path planning (route) on which customers are willing to reach and serve, with the objective of minimizing the routing costs and assignment of vehicles relative paths.
This type of problem is the most important between routing problems, which constitute a subset of the logistics problems; these relate to the problem of defining a set of paths covered by a set of vehicles carrying materials, people or information and that start and end in the same depot using a suitable road network; the resolution of these problems involves the construction of a graph model.
The most realistic routing problems include the appearance of scheduling in which one must also schedule the service timetable; in this case it is considered in addition to the component typical of the geographical routing problem *pure* also a time

component. The VRP has many practical implications in the reality in logistical and distribution contexts detail, such as the school bus service, the collection of waste, the cleaning of roads by vehicles; these problems affect not only companies in the transportation industry, but every company has to face a goods actual transport (e.g. handle internal mail service of a big company).
The VRP can be explained by going to describe in detail the characteristics of the vehicles, customer and road network that define the operating environment. The road network used for transport is normally represented by a graph oriented or less (depending on whether the arches are set to the direction of travel or less) whose arcs represent sections of road passable and whose vertex correspond to the important points network, that is, at intersections and at points where they are located customers and depots. It is a weighted graph, or to each arc is specified the cost of transit (length connection) but some models can represent the travel time (it depends the type of vehicle that runs through this link or what time frame during which the link is crossed). Each customer is characterized by:

- a node of the road graph in which it is located;
- a quantity of goods, even of different types, which must be delivered and/or collected (the Customers can request: delivery of goods, removal of goods, both services);
- intervals of time (time windows) for the service, as customers have a precise time during which they can receive the requested service (the opening hours of a exercise are an example);
- times for load and unload goods by the customer;
- any subsets of vehicles that can be used for deliver (for example, in certain parts of the city may only be suitable for some types of vehicles);
- a question that, if not entirely satisfied:
  - you define levels of priority (precedence constraints defined between customers);
  - or if it is not backed in whole or in part the service is expected a penalty (in terms of time or costs).

The routes have origin and destination in one or more depots located in the vertex of the graph. Each depot possesses a number and certain types of vehicles may also vary the quantity of goods that the warehouse is able to treat. In some cases, a pre-assigning some customers to stores and vehicles depart and returning to the same depot, for which each store acts independently from the other and so the problem can be decomposed into several problems relating each to a single depot.

It is frequently the situation in which this decomposition can not take place because the vehicle does not returns to the same starting depot for the failure of customers pre-assignment special depot, to cope with this, if necessary, to additional constraints, for example the constraint of capacity. Another dimension for the classification of VRP problems is given by the vehicle's characteristics:

- the fleet of vehicles can be fixed or variable;
- the starting depot, where the vehicles come back or not at the end of the trail;
- capacity of the vehicle, which can be defined by the weight, volume, number of units of packaging of goods, with possible division into compartments of the goods;
- some vehicles may not be suitable for loading certain types of goods (such as the need for cold storage for perishable goods);
- loading and unloading methods and availability on board of facilities for handling of goods (movable platforms);
- impossible for the vehicle to transit in some road sections;
- the cost associated with the use of the vehicle, the cost related to the time used or distance traveled.

The problem is also characterized by drivers who are used for the driving of vehicles which are restricted by different types of trade union regulation depending on whether they are employees of the carrier or self-employed workers (eg working hours, number and length of breaks). Usually these constraints are associated with the vehicle.

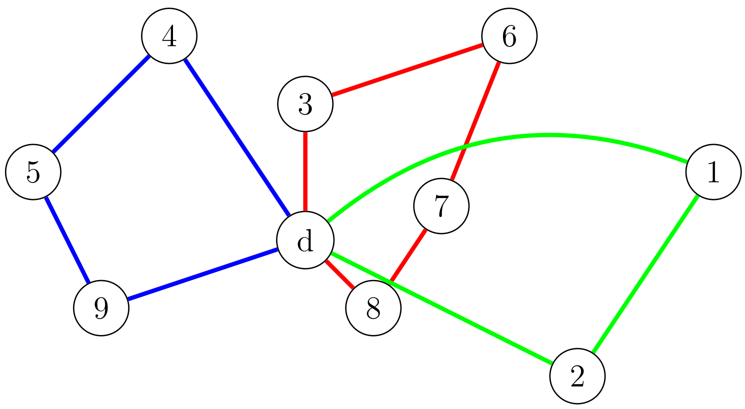


Figure 2.1: A VRP graph

The main objectives, even conflicting, of the vehicle routing problems, are:

- minimize the number of vehicles used to serve all customers;
- minimize the total distance traveled by the fleet;
- minimize total transport cost which depends on the total distance traveled, from total time and the fixed costs associated with the vehicle;
- minimize the penalties associated with the service led to complete only part of the clients;
- balancing paths regarding the travel time and / or paid by the vehicle;
- minimize an objective function which corresponds to a combination of previous objectives.

2.2 Complexity

The VRP is not a purely geographical problem as the customer demand can be binding; in fact most of the time you can find an optimal solution only if the number of customers to visit is relatively small.
Specifically, these problems belong to a class of *NP*-hard problems, that is, the execution of algorithms that solve in an exact way these problems requires a time of exponential calculation in the problem size. To better understand the concept of *NP*-hard following is a brief exposition of the various complexity classes. The various problems can be divided into classes according to the time required algorithm, defined by the number of operations required to solve this problem. The main categories are:

- *P* (Polynomial) problems, for which there are solution algorithms of complexity polynomial, are decision problems that can be solved with a car sequential deterministic in a time that is polynomial with respect to the size of the the input data.
- *NP* (Nondeterministic Polynomial) problems, this class includes the problems of Decision whose positive solutions can be verified in polynomial time having the right information, or equivalently, whose solution can be found in polynomial time with a non-deterministic machine.
- *NP* complete problem, a problem is *NP*-complete if and only if it belongs to *NP* and every other problem in *NP* can be attributed to it in polynomial time; are the difficult problems in *NP* class in the sense that, if it were an algorithm able to fix *quickly* (in the sense of using polynomial time) any *NP*-complete problem, then you could use it to solve *faster* every problem in *NP*.

- *NP*-hard problems (hard not deterministic polynomial time), a problem fall into this class if every problem in *NP* is reducible to it in time polynomial (even if it does not belong to *NP*); these problems are at least complex as those laid down in the version of optimizing an *NP*-complete problem; to demonstrate that a calculation problem is equivalent to a problem known *NP*hard it is to demonstrate that it is virtually impossible to find an efficient way to solve it.

2.3 The Vehicle Routing Problem with Backhauls (VRPB)

The VRP with Backhaul (VRPB) is an extension of the Capacitated VRP (CVRP). In this problem, the set of customers is divided into two subsets:

- the first subset, L, contains n Linehaul customers. Each customer has a specific demand of goods; Those are customers who need to receive a quantity of products;
- the second subset, B, it contains m Backhaul customers. From this type of customer a certain amount of product must be withdrawn.

It is a problem that typically well suited to reality as large islands, in which is possible to identify a restricted number of points of access/exit of goods, usually coincident with the ports, and a high number of Importers clients and other Exporters to be served.
In this case, the route traveled is initially all points of delivery of goods taken from a depot (Linehaul customers), and after all withdrawal points of goods to be transported to the depot (Backhaul customers).
The above described sequence is motivated by two main aspects: the first one is that in many practical application Linehaul customers have a higher priority than Backhaul customers. In addition, vehicles are often rear-loaded. Due to this fact, in case of a mixed service, an on-board load arrangement (supposing that is possible) is required during the route, and an extended time could be requested, making the process very inefficient.
The VRPB can be formulated using a directed graph $G = (N, A)$. N represent the set of vertex, wheres A represent the set of arcs. The set of customers is $N \backslash \{n_0\}$, where n_0 represent the depot node. In this problem, $N \backslash \{n_0\}$ is split in two subsets, L and B.
Vertex are numbered so that $L = \{1, ..., n\}$ and $B = \{n + 1, ..., n + m\}$.
In this case it is established a precedence constraint between the customers in the L and customers in B, if a path serving customers of both types: all of L customers must be visited before each of those in B, this to avoid having to reorganize the loads on the vehicle.
For each node i is associated with a non-negative demand (or offer) d_i for goods;

the depot is associated with a fictitious value $d_0 = 0$. The quantity of goods to be delivered and to be withdrawn is fixed and known in advance. If the cost matrix is asymmetric, the problem is called VRP with Asymmetrical Backhaul (AVRPB). There is also a variant that comprises the constraint time window (time windows) called VRPBTW.
The objective is find a route set that minimize the total routing cost, defined as the sum of the arcs belonging to the circuits. A set of constraint must be respected, solving a VRPB, such that:

- each route visit the depot;
- each node is visited by one and only one route;
- in each route, all Linehaul customers are visited prior to any Backhaul customers;
- the total requests of customers at L and those in B does not exceed, separately, the capacity C of the vehicle.
- Total distance traveled by the vehicle is minimized.

Typically routes with only Backhaul customers are not allowed.
Denoted by K_L and K_B the minimum number of vehicles required to serve all customers in all those in B and L, respectively, it shall be assumed for the admissibility of an instance that:
$K \geq max\{K_L, K_B\}$.
K_L and K_B can be obtained by solving the instance of Bin Packing Problem associated with the respective subsets of vertex.
The VRPB and the AVRPB generalize respectively SCVRP (Symmetric Capacitated VRP) and ACVRP (Asymmetric Capacitated VRP) when the set of Backhaul $B = \emptyset$, and therefore are NP-hard problems in the strict sense.

2.4 VRPB Mathematical Model

In the modeling phase, consider a depot n_0 (which generally coincides with the port), a set L of Linehaul Clients (importers), a set B of Backhaul Clients (exporters), and a set K of different trucks, each with capacity u_k. An integer demand $d_i \geq 0$ of load units is associated with each customer $i \in \{L \cup B\}$. When $i \in L$, d_i represents the number of load units used to service Linehaul customer. When $i \in B$, d_i represents the number of load units used to service Backhaul customers. In this problem setting, d_i may not exceed the value of the load capacity u_k of the truck k.

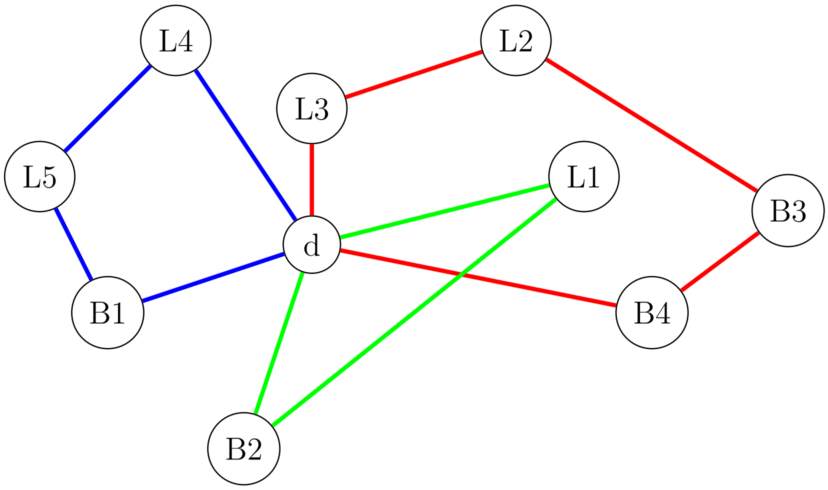


Figure 2.2: A VRPB graph

Detailing more, is possible to define an oriented graph $G(N, A)$ where:

- $N = n_0 \cup L \cup B$ (node n_0 is the depot)

We need to term two new node sets, in order to define the arcs of the graph:

- $B_0 = B \cup \{n_0\}$;
- $L_0 = L \cup \{n_0\}$

The arc set A can be partitioned into three disjoint subsets:

- $A_1 = \{(i, j) \in A : i \in L_0, j \in L\}$
- $A_2 = \{(i, j) \in B : i \in L_0, j \in B_0\}$
- $A_3 = \{(i, j) \in A : i \in L, j \in B_0\}$

Note that $A = A_1 \cup A_2 \cup A_3$ does not contain arcs that cannot belong to a feasible solution.
In this way, it is represented the fact that a truck k can serve a Linehaul customer after the depot or after another Linehaul (A_1), or a truck can serve a Backhaul customer or go back to the depot after a Backhaul customer (A_2), or a truck can serve a Backhaul customer after a Linehaul (A_3). All the other possibilities are not allowed; so, for example, it's not possible in a route to serve only Backhaul customers, or in a route to serve a Linehaul after a Backhaul customer.
For each node i, it is possible to split the completeness incidents arcs into 2 subsets:

- $\Delta_i^+ = \{j : (i, j) \in A\}$
- $\Delta_i^- = \{j : (j, i) \in A\}$

representing, of the given node i, the forward (Δ_i^+) and the backward (Δ_i^-) stars.
After define all possible arcs, during the modeling phase it is necessary to size the fleet of trucks; it is defined:
$\mathscr{F}$ = family of all *admissible* subsets of customers.
At this point we define:
$r(S)$: minimum number of vehicles to serve all customers in $S, S \in \mathscr{F}$
The following decision variable is defined:

- x_{ij}^k link selection variable: it is equal to 1 if arc $(i, j) \in A$ is traversed by truck $k \in K$, 0 otherwise;

Each arc (i, j) has a cost $c_{i,j}$ representing the cost related to the route between the node i and node j, independently from the type of vehicle used. These costs are related to e. g. toll costs. A heterogeneous fleet of vehicles are stationed at the depot and are used to supply the customers. In our problem we consider the cost of an arc exactly equal to its length, without considering fixed costs (e.g. capital amortization cost) and variable costs (fuel costs, service/maintenance costs, tyres wear costs).
In our problem, the carrier use only one type of vehicle, with the common feature of using a container for the carriage of goods; this means that we cannot consider the index k related to a truck but is possible to consider only one type of vehicle with only one capacity u of load.
Due to the above specifications, it is possible to term two variables, both linked to the problem:

- c_{ij}^k is the routing cost of truck $k \in K$ on arc $(i, j) \in A$;

- u_k is single truck k capacity.

Using the above considerations, it is possible to mathematically express the problem as a Linear Programming (LP) Problem, defining an objective function that minimize the total cost of the delivery of goods between depot and customers. Starting from the fact that all trucks use a container to stock the goods to deliver/collect, is correct to use a two index model, without considering the k truck.

$$min \sum_{(i,j)\in \overline{A}} c_{ij} x_{ij}$$

subject to

$$\sum_{i \in \Delta_j^-} x_{ij} = 1, \quad \forall j \in N \setminus \{0\} \tag{2.1}$$

$$\sum_{j \in \Delta_j^+} x_{ij} = 1, \quad \forall i \in N \setminus \{0\} \tag{2.2}$$

$$\sum_{i \in \Delta_0^-} x_{i0} = K \tag{2.3}$$

$$\sum_{j \in \Delta_0^+} x_{0j} = K \tag{2.4}$$

$$\sum_{j \in S} \sum_{i \in \Delta_j^- \setminus S} x_{ij} \geq r(S), \qquad \forall S \in \mathscr{F} \tag{2.5}$$

$$\sum_{i \in S} \sum_{j \in \Delta_i^+ \setminus S} x_{ij} \geq r(S), \qquad \forall S \in \mathscr{F} \tag{2.6}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall (i, j) \in A, \tag{2.7}$$

The constraints (2.1), (2.3) impose a maximum degree, the indegree and outdegree, respectively, for the customers. In the case studied, it is required that the customer is served by only one truck. The constraints (2.2), (2.4) behave exactly like the previous ones, but requiring exactly k trucks transiting for the depot.
The so-called capacity constraints-Cut (CCCS) (2.5) and (2.6) require both connectivity routes and the limitations on the capacity of the truck. Note that, because of the degree of constraints (2.1) - (2.4), for each given subset $S \in \mathscr{F}$, the left member of (2.5) and (2.6) are equal (i.e., the number of arcs entering in S is equal to number of arcs that come out). Finally, it is possible to note that both families of constraints (2.5) and (2.6) have an increasing exponential with a cardinality that depends to the max (n, m). Due to this fact, the LP relaxation of the minimization problem (see chapter 3) defined by the objective function and the constraints (2.1) - (2.6), without considering the constraint (2.7) but using:
$0 \leq x_{i,j} \leq 1$, with $(i, j) \in A$
cannot be directly solved, even for moderate sized problems.

Chapter 3

Integer Linear Programming

An Integer Programming problem consists of maximizing or minimizing a real function of many variables, subject to inequality and equality constraints and integrality restrictions on some or all of the variables. If the function that must be maximized or minimized and inequality and equality constraints are linear, the problem is called Linear Integer Programming problem (ILP).
A great number of real problems can be represented and solved by integer and combinatorial optimization: facility location, transportation network design, distribution of goods, production scheduling and in general Vehicle Routing Problems, that are analyzed in detail in chapter 2.

3.1 Integer Linear Programming

We can write a general linear Mixed-Integer Programming (MIP) problem as follows:

$$max\{cx + hy : Ax + Gy \leq b, x \in \mathbb{Z}_+^n, y \in \mathbb{R}_+^p\}$$

where $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_p)$ are the variables, c is a n-vector, h a p-vector, A an $m \times n$ matrix, G an $m \times p$ matrix and b an m-vector. An instance of the problem is a set of data (c, h, A, G, b). Because of the presence of both integer and continuous (real) variables, this problem is called mixed. Moreover, it can be observed that minimizing a function is equivalent to maximizing the negative of the same function and that an equality constraint can be represented by two inequalities.
The set $S = \{x \in \mathbb{Z}_+^n, y \in \mathbb{R}_+^p, Ax + Gy \leq b\}$ is called the feasible region, and an $(x, y) \in S$ is called a feasible solution. An instance is said to be feasible if $S \neq \emptyset$.
The function

$$z = cx + hy$$

is called the objective function. A feasible point (x_0, y_0) for which the objective function has the maximum value, that is, $cx_0 + hy_0 \geq cx + hy \quad \forall (x, y) \in S$, is called an optimal solution. If (x_0, y_0) is an optimal solution, $cx_0 + hy_0$ is called the optimal value or weight of the solution.
A feasible instance of MIP may not have an optimal solution. It can be said that an instance is unbounded if there is an $(x, y) \in S$ such that $cx + hy \geq \omega$, for any $\omega \in \mathbb{R}^1$. In this case we use the notation $z = \infty$. If you solve an instance of MIP you can obtain an optimal solution or show that it is either unbounded or infeasible. When there are no continuous variables we have a special case of MIP called linear(pure) integer programming problem (ILP):

$$max\{cx : Ax \leq b, x \in \mathbb{Z}^n_+\}$$

On the other hand, when there are no integer variables we obtain a linear programming problem (LP)

$$max\{hy : Gy \leq b, y \in \mathbb{R}^p_+\}$$

We have an other important frequent case, when the integer variables are used to represent logical relationships.
Consequently they can only be equal to 0 or 1. Thus we obtain the 0-1 MIP (respectively 0 -1 IP) in which $x \in \mathbb{Z}^n_+$ is replaced by $x \in B^n$, g B^n is the set of n-dimensional binary vectors.
For example, we use of 0-1 variables to represent binary choice, if we have to choose between two possibilities, as an event that can or cannot occur. In the model of this problem it is introduced a binary variable x, that assume the value 1 if the event occurs, and 0 otherwise.
The study of theory and algorithms of linear programming is fundamental to understand integer programming. It is well known that solving an integer programming problem is much more difficult than a linear programming problem, since the theory and the computational aspects of integer programming are less developed than the ones of linear programming. For this reason the theory of linear programming represents a guide for developing results for integer programming. Moreover, linear programming algorithms are very often used as a subroutine in integer programming algorithms to obtain upper bounds on the value of the integer program. Let

$$z_{IP} = max\{cx : Ax \leq b, x \in \mathbb{Z}^n_+\}$$

note that $z_{LP} \geq z_{IP}$ since $\mathbb{Z}^n_+ \subset \mathbb{R}^n_+$. The upper bound z_{LP} can be used to prove optimality for IP; that is, if x_0 is a feasible solution to IP and $cx_0 = z_{LP}$, then x_0 is an optimal solution to IP. See [WN99].

3.2 Continuous Relaxation

Let $z_{IP} = max\{cx : Ax \leq b, x \in \mathbb{Z}^n_+\}$ be an integer problem. If we remove integrality restrictions we obtain the following problem: $z_{LP} = max\{cx : Ax \leq b, x \in \mathbb{R}^n_+\}$, called continuous relaxation of the original problem. Generally, the continuous relaxation problem is easier to solve and it has a lower execution time than the integer problem associated. Let S and S^* be feasible regions of z_{IP} and z_{LP}, respectively. Then $S = S^* \cap \mathbb{Z}^n$. Consequently:

- S may be the empty set, though S^* is different to empty set,
- if S^* is bounded, then S is finite.

It follows that the optimal solution could be found calculating the value that the objective function f assumes in every point $(x, y) \in S$ and choosing the maximum value of f. Obviously, this method is allowed only if the cardinality of S is very small. You might remove integrality constraints and approximate the optimal solution of the continuous relaxation. However, this approach is useless for tow reasons:

- the approximate solution may be unfeasible
- the approximate solution may be feasible, but very far from the optimal solution.(when variables assume very small optimum values, for example if we have binary variables)

Both cases are shown in the figure 3.1.

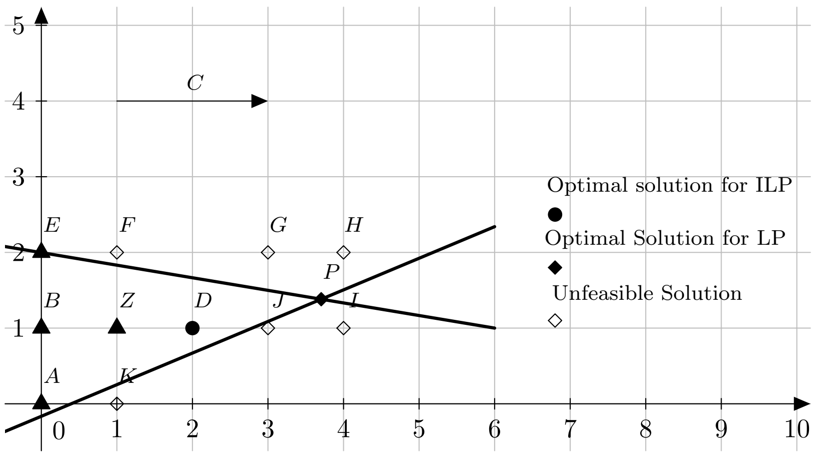


Figure 3.1: Continuous relaxation

Usually, we need an alternative approach using the continuous relaxation, that can be solved with the simplex method. In the paragraph 3.3 we analyze two of these methods: branch and bound and cutting-plane.

3.3 Exact Methods for ILP

Linear programming problems and in particular integer programming problems are very difficult to solve. In fact, no efficient general algorithm is known for their solution.
Algorithms for integer programming problems can be divided in three main sets:

- Exact algorithms, as cutting-planes, branch-and-bound, and dynamic programming, that guarantee to find an optimal solution; however may have an exponential number of iterations.

- Heuristic algorithms that provide a suboptimal solution,but without a guarantee on its quality. Although the running time is not guaranteed to be polynomial, empirical evidence shows that some of these algorithms find a good solution in a short time.

- Approximation algorithms that assure in polynomial time a suboptimal solution and a bound on the degree of sub-optimality.

In this section we analyze two exact method: branch-and-bound and cutting plane.

3.3.1 Branch and bound

Branch-and-bound was developed by Land and Doig and by Dakin. In this method it is very important to have an upper bound for the maximum value of ILP, easy to compute and not far from the optimum value. Gomory's cutting plane method is one method of obtaining an upper bound.
We give a general branch-and-bound algorithm for solving IP. In the description of the algorithm, $\mathscr{L}$ is a collection of integer programs $\{IP_i\}$, each of which is of the form $z_ip = maxcx : x \in S_i$ where $S_i \subseteq S$. Associated with each problem in $\mathscr{L}$ is an upper bound $\overline{z}_i \geq z_{iIP}$.
General Branch-and-Bound Algorithm

1. (Initialization): $\mathscr{L} = \{IP\}, S_0 = S, \overline{z}_0 = \infty$, and $\underline{z}_{IP} = -\infty$.

2. (Termination test): If $\mathscr{L} = \emptyset$, then the solution x_0 that yielded $\underline{z}_{IP} = cx_o$ is optimal.

3. (Problem selection and relaxation): Select and delete a problem IP_i from $\mathscr{L}$. Solve its relaxation RP_i. Let z_{iR} be the optimal value of the relaxation and let x_{iR} be an optimal solution if one exists.

4. (Pruning):

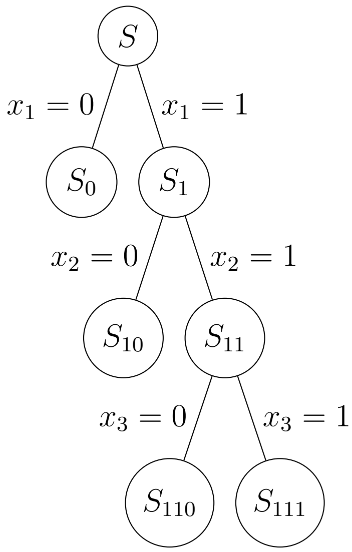


Figure 3.2: Example of branch-and-bound method

(a) If $z_{iR} \leq \underline{z}_{IP}$, go to Step 2. (Note if the relaxation is solved by a dual algorithm, then the step is applicable as soon as the dual value reaches or falls below $\underline{z}_{IP}$)

(b) If $x_{iR} \notin S_i$, go to Step 5.

(c) If $x_{iR} \in S_i$ and $cx_{iR} > \underline{z}_{IP}$, let $\underline{z}_{IP} = cx_{iR}$. Delete from $\mathscr{L}$ all problems with $\overline{z} \leq \underline{z}_{IP}$. If $cx_{iR} = ziR$, go to Step 2; otherwise go to Step 5.

5. (Division): Let $\{S_{ij}\}_{j=1}^{k}$ be a division of S_i. Add problems $\{IP_{ij}\}_{j=1}^{k}$ to L, where $\overline{z}_{ij} = z_{iR}$ for j= 1, ... ,k. Go to Step 2.

3.3.2 Cutting-plane method

Let z_{IP} be a linear integer programming problem, x^* th optimal solution (optimal value z^*), z_{LP} the continuous relaxation of z_{IP}, x_0 the optimal solution of z_{LP} (optimal value z_0). An hyperplane $ax \geq a_0$ is called cutting plane if:

- x_0 is unfeasible ($ax_0 < a_0$)
- is feasible for all optimal integer solution of the original problem ($ax \geq a_0, \forall x$ feasible and integer)

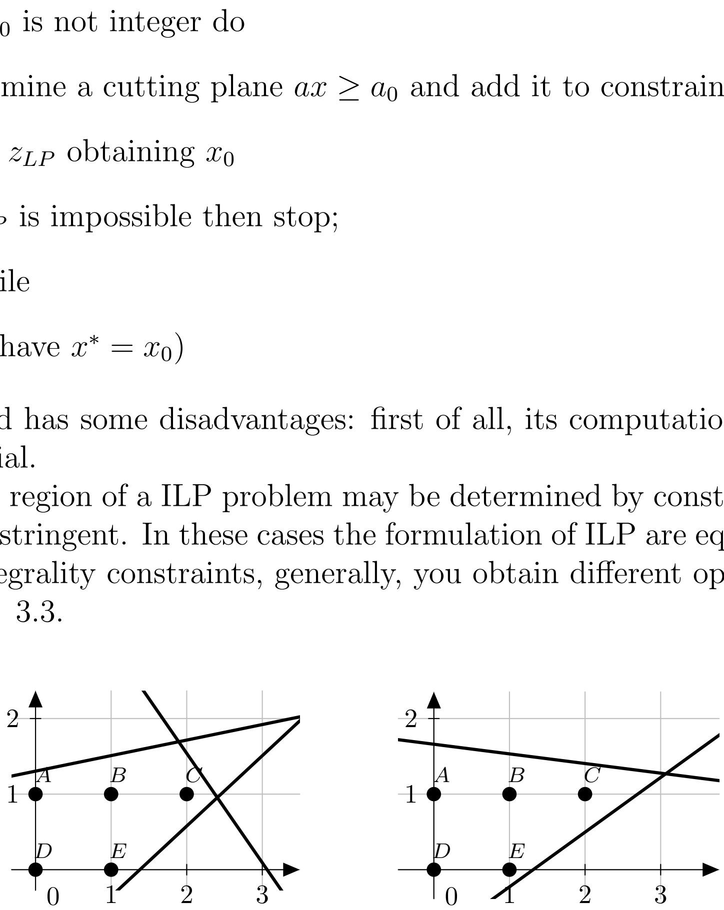
Cutting planes algorithm:

1. begin

2. solve z_{LP} obtaining x_0
3. if z_{LP} is unbounded or impossible then stop;
4. while x_0 is not integer do
5. determine a cutting plane $ax \geq a_0$ and add it to constraints of P
6. solve z_{LP} obtaining x_0
7. if z_{LP} is impossible then stop;
8. end while
9. end (you have $x^* = x_0$)

This method has some disadvantages: first of all, its computational complexity is not polynomial.

The feasible region of a ILP problem may be determined by constraints that can be more or less stringent. In these cases the formulation of ILP are equivalent, but if you remove integrality constraints, generally, you obtain different optimal solution, as shown in fig. 3.3.



(a) Feasible region (b) Feasible region

Figure 3.3: Example of different feasible region of the same ILP

To understand which is the ideal formulation we need the following definition:
Given a set $S \subseteq \mathbb{R}^n$, a point $x \in \mathbb{R}^n$ is a convex combination of points of S if there exists a finite set of points $\{x_i\}_i^t = l \in S$ and a $\lambda \in \mathbb{R}_+^t$ with $\sum_{i=l}^t \lambda_i = 1$ and $x = \sum_{i=l}^t \lambda_i x_i$.
The convex hull of S, denoted by conv(S), is the set of all points that are convex combinations of points in S. If $S \subseteq \mathbb{Z}^n$ conv(S) represents a polytope P' with every corner is an integer point. Given S it is possible to find A', d', subject to $P' = \{x \in \mathbb{R}^n : A'x \geq d', x \geq 0\} =$ conv(S) and it means that $min\{c^T x : x \in S\}=$ $\min\{c^T x : A'x \geq d, x \geq 0\}$
In this case you can solve the ILP through simplex method.
Unfortunately, to determine conv(S) is very difficult, because in general, the system $A'x \geq d'$ has a large number of constraints

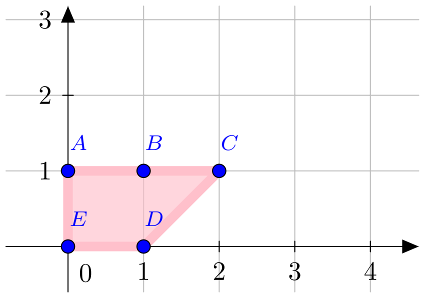


Figure 3.4: Convex hull

3.3.3 Exact Algorithms for VRP with Backhauls

Several exact algorithms are proposed in the literature in order to solve with an optimal solution a VRPB problems. In this brief review, we will focus on two peculiarities, relating to the input instances: the first one is the size ($n + m$, where $n =$ number of Linehaul and $m =$ number of Backhaul) of the instances solved, and the second one in the computational time used to achieve the results. The goal is to better analyze the limits of the exact methods, that are linked to the size of the problem.

In [TV97] an ILP model, valid both for AVRPB and VRPB Problems, is presented. The branch-and-bound lowest-first algorithm for this model, based on the Lagrangian relaxation, is an exact procedure that has been applied to three different subsets of instances taken form the literature; the first one is extract form the symmetric instances by [GJB89]. Instances whose $(n + m)$ size range from 25 to 68, were solved within an imposed time limit of 6000 CPU seconds. The second subset of instances solved is that proposed by [TV96]. In this case, VRP symmetric instances with a number of customers $n + m$ between 21 to 100, were solved. The first step was, starting from VRP Instances, to generate VRPB instances using different percentages (50%, 66%, 80%) of Linehaul/Backhaul customers. In the same way, a third set of VRPB Instances, starting from those proposed by [FTV94]was generated. In this case too, asymmetric VRPB instances are derived by introducing a different percentage (50%, 66%, 80%) of Linehaul/Backhaul customers. The number of customers $(n+m)$ range from 33 to 70 nodes, and were solved in an imposed time limit of 6000 seconds.

Another VRPB mathematical model is proposed by [MGB99]. An Exact method, that use duality in order to reduce the number of variables of a given integer program P, generate a dual problem D that is solvable by an Integer Programming Solver (in this case CPLEX). Like the exact method aforementioned, a subset of symmetric instances by [GJB89] and another subset of those proposed by [TV96] were used, within a size bounded to 113 nodes, with an imposed time limit of 25000 seconds.

Several variants of VRPB were investigated in the literature. A Pickup and Delivery

Problem with Backhauling and Time Windows is considered in [CS03], and a mixed integer linear program has been formulated. This formulation takes advantage of the conditions of the problem to eliminate those arcs that are known to be infeasible. The instances have been constructed from live data and by random generation. In their work, authors asserts that this technique would take a prohibitive amount of running time if applied to instances of realistic size. Alternative lower bounds were found by relaxing integrality or time window constraints for those instances having up to 100 customers to serve.
More recently, in the work of [OB16] another VRPB variant was investigated. A model of VRPMBTW, a combination of Vehicle Routing Problem with Time Windows (VRPTW) and Vehicle Routing Problem with Mixed Backhauls (VRPBM) is proposed. In their conclusions, the authors do not suggest the use of exact methods for instances of relative big sizes. In their case-study, a maximum number of 28 customers is reported, that was solved in up to 6 seconds.

Chapter 4

Heuristics and Metaheuristics

4.1 Approximate solutions

Find the optimal solution of the modest-sized NP-hard optimization problems also may be too burdensome. Furthermore, given that the parameters of the model considered may be suffering from approximation errors due to modeling, this effort may be of no importance.
In practical cases may be accepted *good* solutions which, hopefully, are not far away by the optimum. A heuristic algorithm is an algorithm that solves an optimization problem, generally using common sense rules, and provides a feasible solution but not necessarily good.
A further problem, in addition to the heuristic design, is its evaluation. When possible (not always so), should be given an overstatement for the error made by accepting a heuristic approach related to the studied problem.
Given a problem P, define S the set of all feasible solution. A function c evaluate the cost of a generic solution $x \in S$; called

$$z_{opt} = min(c(x), x \in S)$$

the value of the optimum solution and z_A the value provided by the heuristic algorithm, are defined:

- Absolute error: $A_E = z_{opt} - z_A$
- Relative error: $R_E = \frac{z_{opt}-z_A}{z_{opt}}$

in [SW00] is defined a $\rho - approximation$ algorithm as a procedure for an optimization problem, that generates solutions with a guarantee on its quality. In an approximation scheme, the user can specify any level of accuracy of the approximation. As might be expected, the run time increases as more accurate accuracy levels are

requested. A value $\epsilon > 0$ defines the accuracy of the approximation. An approximation scheme is a family of algorithms $\{A_\epsilon\}$ such that $\{A_\epsilon\}$ is a $(1+\epsilon)-$*approximation* algorithm. A polynomial-time approximation scheme (PTAS) is an approximation scheme with running time that is polynomial in the input size for fixed ϵ. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme with running time that is polynomial in the input size and $1/\epsilon$.
When they exist, should be applied approximation algorithms, for which it is calculated, by definition, a maximum limit of error made, with respect to the heuristic algorithms based on common sense, but for which it can not provide a maximum error limit.
For some problems approximate algorithms are not known, for others the best approximation algorithm has large values of ϵ (greater than 0.5). Moreover heuristic algorithms with average performance are acceptable, at times are preferred to approximate algorithms (at least at first) because they are easier to implement and generally faster.

4.2 Heuristic algorithms

As we stated in the previous section, many of Combinatorial Optimization problems are *difficult*, and it is often necessary to develop heuristic algorithms. Normally the heuristic algorithms have a low computational complexity, but in some cases, for large problems and complex structure, you may need to develop sophisticated heuristic algorithms, often with an high complexity. Furthermore, it is possible, in general, that a heuristic algorithm fails and is not able to determine any feasible solution of the problem, without being able to demonstrate that they do not exist.
Design effective heuristic algorithms requires careful analysis of the problem to be solved once to identify the *structure*, i.e the specific useful characteristics, and a good knowledge of the main algorithmic techniques available. In fact, even if every problem has its own specific characteristics, there are a number of general techniques that can be applied in different ways, to many problems, producing classes of well-defined optimization algorithms. In this chapter we will focus on two of the main algorithmic techniques useful for the realization of heuristic algorithms for Combinatorial Optimization problems: the greedy algorithms and those of local search. These algorithmic techniques surely do not exhaust the spectrum of possible heuristics, as far as provide a good starting point for the analysis and characterization of many approaches.
In particular, it is worth noting here that the emphasis on the *structure* of the optimization problem is also common to the techniques used for the construction of greatest lower bound on the optimal value of the objective function, where the absolute minimum cannot be calculated. This often causes a problem of the same structure is used both to realize that heuristics to determine greatest lower bound.
Is possible as well to have a sort of *partnership* between heuristics and relaxations,

see section 3.2, as in cases of rounding techniques and heuristics Lagrangian. By contrast, the above rounding techniques and heuristics are often classified into two large families: the *constructive* heuristics and the *improvement* heuristic. Typically, the solutions provided by the constructive heuristics and those used by the improvement heuristics are feasible solutions. For certain problems can be extremely difficult to determine an initial feasible solution, then it can be appropriate to use the Lagrangian relaxation of some constraints, i.e., determining a first solution that meets at least some of the constraints and penalize the fact that other constraints are not respected. Then iteratively starting from the solution (infeasible) obtained try to determine that an increased respect for the constraints in the neighborhood of that date.
Goetschalckx and Jacobs-Blecha in [GJB89] propose a heuristic approach based on space-filling curves. Linehaul and Backhaul customers transform from points in the points along a line plan using the transformation curve that fill the space. These two series of points are used to determine viable routes. Then each Linehaul path is merged with the Backhaul path closer than the mapping that fill the space. Toth and Vigo in [TV99] suggest a cluster-first, route second algorithm for the VRPCB and its asymmetric problem (AVRPCB). A cluster is a group of clients that contains only Linehaul customers or Backhaul.
Thangiah, Potvin, and Sun in [TPS96] propose a two-step approach for VRPB with time windows. First, an initial solution is determined using a heuristic insertion. Then, in a second phase, a λ-interchange procedure and a $2 - opt$ procedure is applied to improve the initial solution.

4.3 Metaheuristic algorithms

A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions, from [OL96].
Metaheuristics are strategies that guide the search process, with the goal of an efficiently exploration of the search space in order to find (near-)optimal solutions. The techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes, and they may incorporate mechanisms to avoid getting trapped in confined areas of the search space.
Metaheuristics may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy. So a specialized heuristic can be pair with a control logic that work in an abstract level description. Several methods of solution to address the VRPCB metaheuristics based on can be found in the literature. A similar approach is proposed in [OW02], a work that suggest a reactive tabu search heuristic. This paper describes two route-construction heuristics that

generate initial solutions quickly. These heuristics are based on the saving-insertion and saving-assignment procedures, respectively. The initial solutions are then improved by a reactive tabu search meta-heuristic. The reactive concept is used in a new way to trigger the switch between different neighborhood structures for the intensification and diversification phases of the search.
Another use of the concept of Reactive GRASP is in the work of [PR00]. A new procedure of Reactive GRASP is presented, in which the basic parameter that defines the restrictiveness of the cardinality of the elements of a candidate list during the construction phase is self-adjusted according to the quality of the solutions previously found, without require calibration efforts.In this work, on most of the literature problems considered, the new Reactive GRASP heuristic matches the optimal solution found by an exact column-generation with branch-and-bound algorithm.
In his work Brandao [Bra06] presents a tabu search-based procedure, with a search algorithm that starting from pseudo-lower bounds and improve the solution.
Ropke and Pisinger in [RP06b] show an uniform approach with an heuristic algorithm that can be used for a wide class of VRPB problems by modeling them as rich pickup and delivery problem with time windows (Rich PDPTW). In the same year [TMSZ06] propose a memetic algorithm, while a neural network based approaches are presented in [GO06].
A heuristic approach based on a hybrid operation of reactive tabu search (RTS) and adaptive memory programming (AMP) is proposed to solve the vehicle routing problem with Backhauls (VRPB). This search process that resulted in early convergence when tested on most of the VRPB instances. this method is discussed by Wassan in [Was07]
A multi-ant colony system (MACS) is used to solve VRPB which is a combinatorial optimization problem by [GA09]; in this algorithm artificial ants are used to construct a solution by using pheromone information from previously generated solutions, and an Ant Colony System (ACS) algorithm uses a new construction rule as well as two multi-route local search scheme.
An iterated local search heuristic yielding high-quality solutions is proposed by Arraiz and Palhazi Cuervo in [APC11]. In their heuristic search it is not limited to the space of feasible solutions; Instead, solutions are temporarily considered that do not meet the capacity constraint. Zachariadis and Kiranoudis in [ZK12] propose an effective local search heuristic which explores rich neighborhoods composed of exchanges of variable-length customer sequences. To efficiently investigate a rich solution neighborhood, tentative local search moves are statically encoded by data structures stored in special priority queue structures (Fibonacci Heaps) offering fast minimum retrieval, insertion and deletion capabilities. To avoid cycling phenomena and induce diversification, authors introduce the concept of promises, which is a parameter-free mechanism based on the regional aspiration criterion used in Tabu Search implementations.
Vidal, Crainic, Gendreau, and Prins in [VCGP14] suggest a unified framework for a great variety of routing problems of multi-attribute vehicles, including the VRPCB.

A general-purpose local search based on partial route concatenations is introduced, and a Unified Hybrid Genetic Search (UHGS) with a unified Split algorithm is proposed.

4.3.1 Hill climbing, local minimum

The hill climbing search is a local search based on a search cycle of nodes with the highest values (best) in the vicinity of a particular reference node. The term indicates the hill climbing algorithm's ability to *climb* the nodes toward those with higher values. The hill climbing algorithm search space is limited to only nodes closest to the current one. When a neighbor node is better than the reference node (the current node), the latter is replaced with the new node. The hill climbing algorithm processing cycle ends when it reaches the node with the highest value (local optimum), that is, when no nodes neighbor has greater value than the reference value. In this search technique, the ability of the designer in tuning environmental variables is crucial. For example, the concept of neighborhood, or the use of techniques to avoid a relapse into a local minimum already analyzed (stagnation). In fact, when the algorithm finds a local maximum value, greater than all neighboring nodes, he stops. However, this does not exclude the distant presence of global maximum with higher value. Research hill climbing has the same disadvantages of the greedy search (greedy search). It is moving quickly towards the best node using a too short-sighted strategy. To avoid this problem, the greedy search is often accompanied by other associated techniques that avoid the fallout in local minimum, such as dynamically changing the construction of the neighborhood rules, even randomly, or by preventing the algorithm to process solutions already visited.

4.3.2 Variable Neighborhood Search

A relatively new technique and therefore not yet thoroughly explored is the so-called VNS (Variable Neighbourhood Search), introduced in [HM01] and discussed in [VMOR12]. Contrary to other metaheuristics based on local search methods, VNS does not follow a trajectory but explores neighborhood at a distance gradually increasing from the best current feasible solution, and moves from the latter with a new one if and only if there was evidence of an improvement of the objective function.

VNS Algorithm

Denote N_k a finite set of neighborhoods structures pre-selected ($K = 1, ..., k_{max}$), and with $N_k(x)$ the set of solutions in the kth neighborhood of x (the classical heuristic local search usually use a single structure around, ie $k_{max} = 1$).

The VNS based heuristic includes the following steps:
Initialization:
select the set of neighborhoods structures $N_k, k = 1, ..., K_{max}$, which will be used in research;
find an initial solution x;
choose a condition of exploration term.
Repeat these steps until you have met the time requirement:
fix $k = 1$
until $k \leq k_{max}$, repeat the following steps:

- diversification (shaking): randomly generates a point x neighborhood of x' belonging to the kth ($x' \in N_k(x)$);
- intensification (local research): apply some method of Local Search using x' as the initial solution; denoted with x'' the local optimum obtained
- possible shift: if the local optimum just found is better than incoming x, considers x'' as a new incoming optimum ($x = x''$) and continue the search with $N_1(k = 1)$; otherwise, fix $k = k + 1$

even if x'' is worse than x, since it is in the situation in which $k > k_{max}$, assign $x = x''$ and continue research

A graphical interpretation of this method, applied to a function of scalar

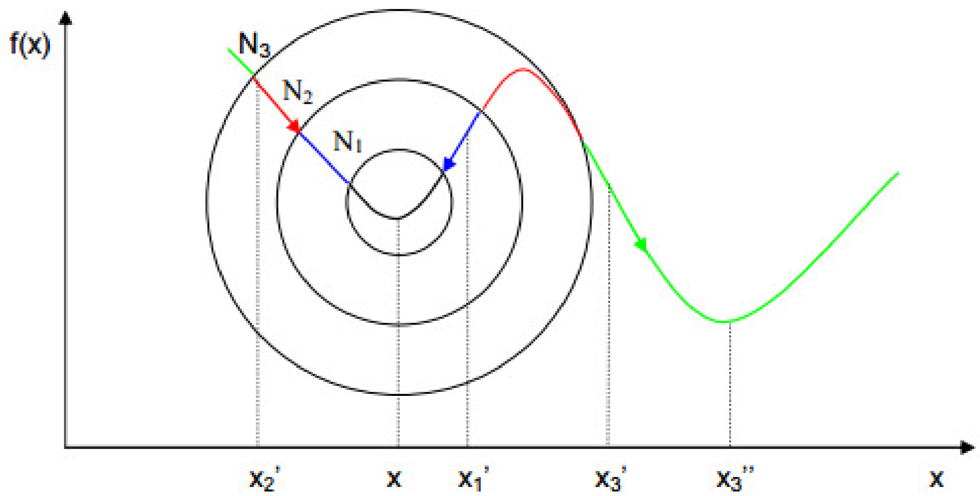


Figure 4.1: VNS method graphical interpretation

variable, can be the following:
Starting by the excellent local x: We take the N_1 around and choosing a random solution x'_1 in its interior, after which applies the iterative descent algorithm; what happens is that the value go back to x, ie $x''_1 = x$.
As second step, it leaves the first region and considers the around N_2, in which it chooses a random solution x'_2; also in this case, applying to x'_2 the descent algorithm, it's back to x, ie $x''_2 = x$.
It repeats again the reasoning: choosing a random solution x'_3 belonging from the around N_3, even more distant from x than the previous; this time, by applying the descent algorithm x'_3, is reached a local optimum $x'_3 = x$, and even better.
What is done at this point is to take x''_3 as a new solution incumbent (ie as the new x) and re-run the same steps seen before.
The execution term condition can be a time limit, a maximum number of iterations, or a maximum number of iterations between two improvements.

4.3.3 Large Neighborhood Search and Adaptive Large Neighborhood Search

Shaw, in [Sha98] presents the Large Neighborhood Search (LNS) metaheuristic. In this heuristic is used a destroy and a repair method with the aim of defining, given a solution, its neighborhood.
The *destroy* method destroy, using (in general) stochastic observation, different part of the starting solution in every invocation, while the *repair* method rebuilds the destroyed solution. The neighborhood of a solution x, called $N(x)$, is then defined as the set of solution that can be founded by first applying the destroy method and after the repair method. the idea is that more impactful is the destroy phase, larger is the possibility of building new neighborhoods. A very simple destroy method can randomly cut a percentage of nodes in a set of routes, and an equally simple repair method could rebuild the solution by inserting removed customers, for example using a greedy heuristic. if the destroy method destruct a large percentage of the initial solution $x, N(x)$ contains a large amount of solution that can be rebuilt by rebuilt method, and not necessarily with an improving approach.
Therefore, it is possible to consider of a change in the neighborhood dimension linked to the dimension of the destroyed part of the solution, or use acceptance criteria taken from Simulated Annealing to change the size of the neighborhood In general, the algorithm obtain a solution x' destroying and rebuilding a starting solution x, evaluate x' with a cost function; if x' is better than x, than x' become the actual solution and the algorithm proceeds until a stopping criteria is met.
In the Adaptive Large Neighborhood Search (ALNS) proposed in [RP06a] the LNS is extended allowing multiple and destroy methods in the same search, each of them characterized by its own weight, dynamically adjusted during the search.

4.3.4 Greedy Randomized Adaptive Search Procedure

A Greedy Randomized Adaptive Search Procedure (GRASP) is an iterative algorithm proposed by[LK73], in which each GRASP iteration consists of two phases, a construction phase, in which is produced a feasible solution, and a local search phase, in which a local optimum is sought in the neighborhood of the previously constructed solution. The best overall solution is maintained as a result. GRASP is an iterative metaheuristic used to solve combinatorial optimization problems.
In the construction phase, a viable solution is iteratively built, one at a time. The basic construction phase GRASP is similar to the semi-greedy heuristic proposed independently by Hart and Shogan in [HS87]. At each iteration of the construction phase, the choice of the next element to be added is determined by sorting all the candidate elements (that is, those that can be added to the solution) in a candidate list RCL, than a greedy function:

$$g : RCL \to \mathbb{R}$$

measures the benefit of selecting each item at a step k.
The GRASP metaheuristic is adaptive because the benefits associated with each element are updated at each k iteration of the construction phase to reflect changes caused by the preceding element choice. The probabilistic component of a GRASP is characterized by choosing in a random way one of the candidates on the list, but not necessarily taking the first one. The list of the best candidates is called the restricted candidate list (RCL). This choice allows different technical solutions to be achieved at each GRASP iteration, but does not necessarily compromise the power of the greedy adaptive component of the method.
As is the case for many deterministic methods, the solutions generated by a GRASP construction are not guaranteed to be locally optimal with respect to simple neighborhood definition. So, it is almost always helpful to apply a local search for groped to improve each constructed solution. A local search algorithm works iteratively replacing then current solution by a better solution in the current solution area. The algorithm stops when no better solution is in the neighborhood. The neighborhood structure N for a problem P define, starting from a solution s of the problem, a subset of solutions of $N(s)$. A solution s is called locally optimal if there is no better solution in $N(s)$. A suitable choice of a neighborhood structure, united to efficient of neighborhood search techniques, and a starting solution are typical features that a local search algorithm must have. A pseudocode of the GRASP procedure is shown:

Function ***GRASP****(Max_iterations)*
Begin
  Read_Input()
  i ← 0

while ($i < Max\_iterations$)
RouteSet ← ***constructionPhase****()*
if RouteSet is not feasible then
RouteSet ← ***repair****(RouteSet)*
endif
BestLocalRouteSet ← ***localSearch*** *(RouteSet)*
if ***cost***($BestLocalRouteSet$) < $BestCost$
BestRouteSet ← *BestLocalRouteSet*
BestCost ← ***cost*** *(BestLocalRouteSet)*
endif
i ← *i + 1*
endwhile
return RouteSet
end

More in detail, in figure 4.2 shows the construction phase of a GRASP procedure, which has the characteristic of that doesn't use any information on the history of research. The GRASP starts from an empty solution. The set of candidate elements is composed by all the elements in the available set. In order to make the selection of the next item to be included in the solution under construction, all candidates elements must be evaluated. To do this is used a greedy evaluation function, which evaluate the incremental increase in the cost function that is obtained with the inclusion of the element in the solution being create. The candidates, before the insertion, are valued and sorted in descending order of performance. The random insertion, choose a feasible node to insert, in according to the truck residual capacity.

A Restricted Candidate List (RCL) that includes the nodes with the best performance, that are the elements whose inclusion in the partial solution in construction appears to have smaller additional costs, is created at this point. This is the greedy aspect of the algorithm.

The element to be included in the partial solution is extracted random from the RCL. This is the probabilistic aspect of the metaheuristic.

Summarizing the construction phase of GRASP, is possible to follow three main phases:

1. evaluation of candidates
2. construction of the Restricted Candidate List
3. random selection of an item from this list

Once the selected node has been inserted into the partial solution, the list of candidates must be updated and the candidate elements re-evaluated. It is necessary to do this because the partially constructed solution influences the performance of the candidate elements. This is the adaptive aspect of the heuristic. These steps are repeated until there is a candidate element.

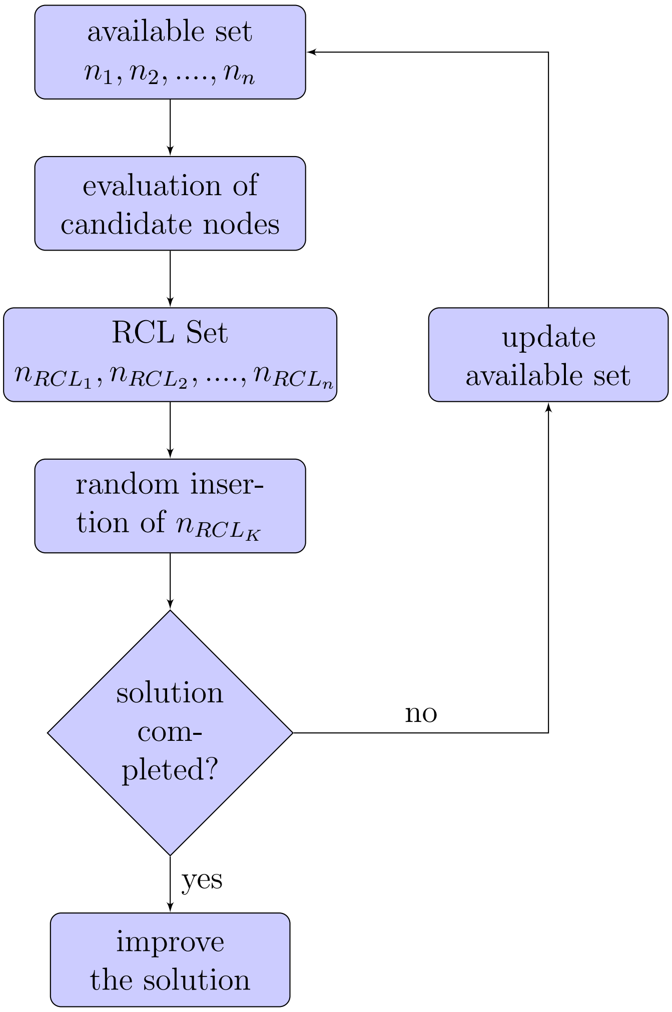


Figure 4.2: Construction of solution diagram

In the case in which the greedy randomized construction procedure is not able to produce a feasible solution, is possible to apply a repair procedure for obtaining eligibility. The solutions generated by a greedy randomized construction are not necessarily optimal. In general, it can be improved by a second phase, the phase of local search.

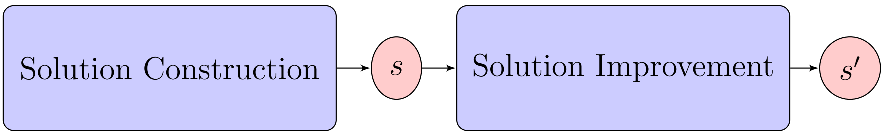


Figure 4.3: Solution improvement

4.3.5 The GRASP algorithm over VRP Problems. Literature overview

It is difficult to formally analyze the quality of the solution values found using the GRASP methodology. However, there is an intuitive justification that sees GRASP as repetitive sampling technique
Each GRASP iteration produces a test sample from an unknown distribution of all results obtained. The mean and variance of the distribution are functions of the restrictive nature of the list of candidates. For example, if the cardinality of the restricted lists is limited to one, then only one solution will be produced and the variance of the distribution will be zero.
Since an effective greedy function, the value of the average solution in this case should be good, but probably not optimal. If a limit is imposed cardinality is less restrictive, many different solutions will be produced which implies a larger variation. Since the greedy function is more impaired in this case, the average value of the solution should degrade. Intuitively, however, the order statistics and the fact that the samples are produced at random, find the best value should exceed the average value. In fact, often the best solutions are often optimal solutions.
A particularly interesting feature of GRASP is the ease with which it can be implemented. In just a few parameters must be set and adjusted, and then the development can focus on delivering efficient data structures to ensure rapid GRASP iterations. Finally, GRASP can be trivially implemented in parallel. Each processor can be initialized with its own copy of the proceedings, the instance data, and a sequence of independent random numbers.
The GRASP iterations are then performed in parallel with a single global variable needed to retain the best solution found on all processors. GRASP has been applied to the vehicle, aircraft, telecommunications, inventory and routing problems.
In the work of Johnson et al. [JPY88], is studied that local optimization procedures can require exponential time but, from an arbitrary starting point, empirically their

efficiency significantly improves, as the initial solution improves. The result is that often many GRASP solutions are generated in the same amount of time required for the local optimization procedure converge to a single random start. In addition, the best of these solutions GRASP is generally significantly better than the only solution obtained from a random starting point.

In [Hjo95]three metaheuristics to effectively search through the space of cyclic orders for VRP are developed. They are based on GRASP, tabu search, and genetic algorithms. In Tabu Search, different schemes are investigated to control the length of the taboo list, including a reactive tabu search method. To get good solutions when using the genetic algorithm, crossover specialist are developed, and is added a local search component. GRASP is used to construct a good first solution.

A GRASP is proposed by [KB95]] for solving VRPTW. The goal is to find the minimum number of vehicle in order to serve the customers. The greedy function of the construction phase takes into account both the cost of minimum global insertion and the penalty cost. Local search is applied to the best solution we found every five iterations of the first phase, and not to each feasible solution. The propose metaheuristic was tested on Solomons data sets in [Sol87] and in a real case study. The former consist of six data sets, each of which contains between 8 and 12 100-node problems over a grid 100*100, with customers placed in the area using different distributions (uniform, clustered, random). The case study was obtained from an Industry-based problem, and contain two instances with 417 customers and two instances with 219 customers, randomly distributed. All datasets are Euclidean (symmetric) distances.

In his work [ABY97] presents an outlet to rebuild paths in response to aircraft groundings and delays experienced during the day. The objective is to minimize the cost of air flights reallocation by taking into account the available resources and other system constraints. This paper develops a GRASP and provides empirical results for data associated with Continental Airlines 757 fleet. The GRASP and lower bounding procedure were applied to a 757 flight schedule obtained from Continental Airlines. The schedule consists of 42 flights serviced by 16 aircraft over a network of 13 airports spanning eight time zones and three continents. A total of 6068 problem instances were generated. For over 90% of the instances tested, the best GRASP solution was within 10% of the lower bound, and the GRASP and lower bounding procedure required less than 15 CPU seconds per instance.

[Atk98] proposed a Greedy Randomised Search Heuristic for Time-Constrained Vehicle Scheduling and the Incorporation of a Learning Strategy. The application described is for solve a vehicle-scheduling problem (VSP). The metaheuristic is tested using a case study, that involves the routing of vehicles over a large area in central London for delivering hot meals from kitchens to schools. In their paper, authors are concerned only with the testing of various search heuristics by applying them to problem VSP, in which 86 schools are to be supplied with meals from 44 kitchens using 16 vehicles.

A methodology is presented by [BHJD98]. The Inventory routing Problem (IRP), in which a central supplier must restock a subset of customers on an intermittent basis, was solved comparing a Clarke-Wright, a GRASP and a sweep-line algorithm. In this work, the proposed procedures were tested on data sets generated from field experience with a national liquid propane distributor. All of the heuristic proposed was suitable to solve the proposed problem, in terms of time responding. From this point of view, the results ranks the GRASP in a mid position.

A GRASP for a routing problem in the telecommunications sector is described by [RR99].

The incorporation of interactive tools for heuristic algorithms is investigated by [CB02]. Close is used in the construction of the airways and improving. The construction phase implemented in a heuristic clustering that builds paths by grouping the remaining customers according to the semi-defined vehicles by applying heuristics 3-opt to reduce the total distance traveled by each vehicle. The greedy function takes into account the routes with smaller cost of insertion and customers with the largest difference between the smallest and the second smallest of the input costs and fewer paths may cross. Since the step of local search, 3-opt is used.

The mixed postman problem, a generalization of the Chinese postman problem, is to find the shortest tour that passes through each edge of a given graph Joint (a chart containing both oriented and directed edges) at least once. In his paper[CMS02] proposes an understanding for the mixed postman problem. The Virtual Private Line Circuit Routing problem is formulated as a multicommodity entire ow problem with additional constraints and an objective function that minimizes propagation delays and congestion and/or network. The authors of [RR03] propose variants of a GRASP with the path relinking.

In the work of [dlP04], a new approach based in the GRASP metaheuristic, Simulated Annealing and Genetic Algorithms is introduced to solve the Undirected Rural Postman Problem (URPP).

The Rural Postman Problem (RPP) consists of determining a minimum cost tour of a specified arc set of a graph $G = (V, A)$ with the particularity that only a subset $T \subseteq A$ of arcs is required to be traversed at least once. The arcs can be directed, undirected or both. RPP is a NP-hard problem.

This approach was applied to the 26 instances described and exactly solved in [CCCM86]. This method was compared with the heuristics of [CCCM86], with the approach of [dCRS98] and [BRRC02]. The result values presented show that, according with the quality of solutions, this hybrid approach outperformed the other methods.

The authors of [LGWL04] propose a hybrid socket for the routing of the vehicles both with Time Window and limited number of vehicles. It is combined with more initialization, re-use solution, the mutation improvement, and with four heuristics: shortly before the left, near customer first, short waiting times first, and long route first.

An NP-hard production-distribution problem for one product over a multi-period horizon is investigated in the work of [BLP07]. A metaheuristic that simultaneously tackle production and routing decisions is developed: a GRASP and two improved versions using either a reactive mechanism or a path-relinking process. These algorithms were tested on 90 randomly generated instances with 50, 100 and 200 customers and 20 periods. About this problem, reaction and path-relinking give better results than the GRASP alone.

In their article, [FB89] presents a model that can be used by planners to both locate maintenance stations and to develop flight schedules that better meet the cyclical demand for maintenance. A first two-phase heuristic is described, from which GRASP is derived. A case study with data supplied by American Airlines for their Boeing 727 and Super 80 and DC-10 fleets was used to test the procedure.

The problem is a large-scale MIP. Because obtaining feasible solutions from its LP relaxation is difficult, the authors propose a GRASP.

The code (Fortran) itself is set up to handle 500 planes and 300 cities and was able to outperform the results obtained in the same period, even if executed on a non-performance computer.

A GRASP with a 2-exchange local search is used to solve the Intermodal Assignment Problem in the work of [FGV95]. The problem of optimally assigning highway trailers railcar hitch in intermodal transport is discussed. Using a set covering formulation, the problem is modeled as an integer linear program, whose linear programming relaxation produces a narrow lower limit. This formulation also provides the basis for the development of a branch and bound algorithm and a GRASP to resolve the problem. The greedy strategy of the construction phase of GRASP consists in selecting an assignment feasible at every step of the most difficult to use available railcar together with the most difficult to assign trailer. The algorithm was tested on 60 historical instances of the given problem. All those instances were already solved using an exact method (branch and bound), but the GRASP proposed outperform from the time consumption point of view. In 23 instances, the optimal value was found in the constructive phase of the GRASP, without using the local search phase.

In [Bar97], the author reports on the results of an effort to design and analyze the rail car unloading area of Procter & Gamble's principal laundry detergent plant. In this problem, the bottleneck lies in the packaging department. The combinatorial problem related allocation of rail cars for positions on the platform and unloading equipment for railroad cars is modeled as a mixed integer nonlinear program. Accounting for the operational and physical limitations of the system, GRASP was used to determine the maximum performance that could be achieved under normal conditions. At about solving the problem, they have proposed four alternatives and evaluated with the help of a GRASP.

Two heuristic based on simulated annealing and GRASP are presented by [Sos00] for the approximate search solutions for a simplified fleet assignment problem.

Both methods are based on exchanging sequence parts of flight legs assigned to an aircraft (rotation cycle) between two randomly chosen aircraft. In simulated annealing, the exchange is such that a solution is accepted according to a probability distribution, while in the grip only exchanges that leads to a better solution is possible and potentially best part of the job is retained and the rest are randomly reallocated. The construction phase does not make use of a list of candidates explicitly restricted, but a solution is constructed simply trying to make the time interval between two flights as small as possible.

Chapter 5

A GRASP for the VRPB

5.1 First version

In this GRASP metaheuristic, the number of routes of the final solution is required as an input data. This number is equal to the k number of available trucks, and the metaheuristic starts from this information to create the sequence of visits of clients. Denote by P a generic VRPB instance, with feasible solution s. The solution s is a set composed by k routes. It is possible to say that, $\forall r \in s$, a sequence Linehaul-backahaul (or Linehaul-depot) has to happen. Consequently, a couple of customer (i, j) with $i \in$ *Linehaul* set and $j \in$ *Backhaul* set (or the depot), must exist in every route $r \in s$, with $r = 1, , k$ and $r \in s$. The main idea is to find the k most promising pairs, and start from these pairs to construct two feasible open-routes, and finally merge them in a feasible route r. In the constructive phase of the metaheuristic, more than k couple are generated, but only k of these are selected, using the Saving $(S_{i,j})$ criteria.
At the greater value of saving $S_{i,j}$, related couple (i, j) is associated as the most promising one and so on, until k couples have been selected.
The *greedy* open route construction is not only guided to the value of the demand (or offer) of the node to insert, but a weight ω_u is associated to all candidate node u that can be inserted in the open route that it is creating.
The rule is that during the construction of a route, starting from a current node t, an insertion of a node u is more promising instead of a node v that has the same demand, if the truck is partially loaded and node u is closer to the depot in respect to the node v. A linear low is used: less is the free space in the truck, more weight is assigned to the distance between the node and the depot:

$$w_{t|u} = \alpha_t \frac{1}{D_{tu}} + (1 - \alpha_t) \frac{1}{\beta D_{tu} + \gamma D_{u0}}$$

where:

$$\alpha_t = \frac{C_{Rt}}{C_M}$$

and C_{Rt} is the residual load capacity of the truck k after serving the node t, C_M is the capacity of the truck, and so it is possible to declare that $\alpha_t \in (0, 1]$, while $\beta = 1$ and $\gamma = 1$ are tuning parameters.
The distance between two nodes i, j is expressed by the value $D_{i,j}$.
In designing a meta-heuristic, two conflicting criteria must be taken into account: diversification and intensification.

- In intensification, the promising regions are explored in a deeper way hoping to find better solutions.

- In diversification, non-explored solutions in the search space must be visited to be sure that all regions of the search space are evenly explored and that the search is not confined to only a reduced number of regions.

Several research in the neighborhood are implemented in the local phase of the proposed GRASP. Strategy of best-improvement or first-improvement or combo moves are implemented.
In the case of a best-improvement strategy, all elements of the neighborhood are evaluated and the current solution is replaced from the best one from the neighborhood (if it finds an improving one). In the case of a first-improvement strategy, the current solution moves to the first neighbor whose cost function value is less than the one of the current solution.
Combo moves starts the local search with one of the previous strategy and makes use of a change of strategy to introduce a diversification in case of local minimum.

5.1.1 Restricted Candidate List

Manifold studies have been conducted to analyze the behavior of a GRASP algorithm in relation to the space of solutions during the exploration of the neighborhood. In particular, in the work of [ARR02] the probability distributions of solution time to a sub-optimal target value is analyzed in five GRASPs that have appeared in the literature and for which source code is available. In this study, a common behavior was found despite several algorithm implementations.
In the work of [RR05], an analysis related to RCL list between greedy and random factors is made. In this work is focused that the GRASP metaheuristic mix greedy and random construction for the restricted candidate list (RCL) creation. This is done basically starting from two different approach: in the first case, using greediness to build the list of candidate and randomness to select an element from them, or, in a second case, by using randomness to build the list and greediness for selection. The candidate elements $e \in C$, where C is the set of available nodes, are sorted according to a greedy function value $v(e)$.
In a cardinality-based RCL, the latter is made up of the first p top-ranked elements.

In a value-based construction, the RCL consists of the elements in the set

$$\{e \in C : v' \leq v(e) \leq v' + \alpha * (v'' - v')\}$$

where
$v'' = max\{v(e) : e \in C\}, v' = min\{v(e) : e \in C\}$ and α is a parameter $\in [0, 1]$.
Since the best value for α is often difficult to determine, it is often assigned a random value for each GRASP iteration.
The case $\alpha = 0$ corresponds to a pure greedy algorithm, while $\alpha = 1$ is equivalent to a random construction.
In the metaheuristic proposed, a cardinality based approach have been used to propose three variants, related to the construction of the Restricted Candidate List (RCL):

- In the first variant, all eligible nodes $e \in C$ are inserted in the RCL. In this case, the feature of $diversification$ in the search space of a methaeuristic is preferred.
- $Intensification$ is followed in the second variant, that composes the RCL list using only the first p most promising nodes from the neighborhood. In our tests, $p = 5$.
- An intermediate variant, the third, is more articulated: after evaluating $v(e), \forall e \in C$, it assigns each node $e \in C$ a value of probability directly linked to the value of $v(e)$. At this point, it inserts in the RCL list a sequence of nodes, starting form the first one (that is the most probably) and repeats the insertion until all nodes that lyes in the first slot percentage of 70% of probability, are inserted.

Another parameter that is possible to use, in order to tune the metaheuristic, is the number of iteration of the GRASP procedure. In the metaheuristic proposed, this number is linked to the number of node of the instance, multiplied by a constant The total computation time increases linearly with the number of iterations. The quality of the current solution could only improve with the last iteration: this means that with a large number of iterations would expect to find a best solution for the price of a greater computation time. Each newly generated solution is compared to the best solution found up to that time and is stored as a best solution if it exceeds in quality.

5.1.2 Main

Function ***GRASP***
Begin
  initialize *(LinehaulSet, BackhaulSet, k, C, max_iter, BestCost, BestRouteSet)*
  i $\leftarrow$ *0*

```
  while (i < Max_iter)
    RouteSet ← constructionPhase(LinehaulSet, BackhaulSet, k, C)
    BestLocalRouteSet ←  localSearch (RouteSet)
    if cost(BestLocalRouteSet) < BestCost
      BestRouteSet ← BestLocalRouteSet
      BestCost ← cost (BestLocalRouteSet)
    endif
    i ← i + 1
  endwhile
  return RouteSet
end
```

The GRASP function is the heart of the algorithm. The first call is the initialize function, to which are passed as empty the following parameters:

- *LinehaulSet*, that will be initialize with the Linehaul customers
- *BackhaulSet*, that will be initialize with the Backhaul customers
- k, that represents the number of routes that the final solution must have
- C, that is the maximum load of a single truck
- max_iter, is the maximum number of iteration of the algorithm
- *BestCost* and *BestRouteSet*, are the cost and the composition of the best solution that was found by the algorithm, respectively.

A counter i is initialized to 1 and used in the main while loop, from which it will come out when the algorithm will reach the number of iterations determined, stored in the variable max_iter. After the initialization phase, inside the while loop is called the function *constructionPhase*, in which are passed as parameters the set of Linehaul (*LinehaulSet*) customers, the set of Backhaul customers (*BackhaulSet*) and k, that represents the number of routes that will compose the final solution (set by the user), and the maximum load capacity of the truck C. The *constructionPhase* returns a collection of routes indicated by *RouteSet*, that is a feasible solution of the initial problem. The procedure *localSearch* then take as a parameter the collection of routes just returned, *RouteSet*. The procedure returns the best *RouteSet* after the application of local search, indicated with *BestLocalRouteSet*. At this point it is estimated the overall cost of the routes *BestLocalRouteSet*: If the actual value of the objective function related at the solution found is lower than the actual *BestCost*, (the lowest cost found by the algorithm up to the actual iteration), then it is assigned to *BestRouteSet* the collection routes just found (*BestLocalRouteSet*) and at *BestCost* the cost of *BestLocalRouteSet*. It increases the value of the counter i and repeats the steps described above for max_iter iterations. The main

GRASP procedure returns the *BestRouteSet*, that is the *best* set of routes found and improved by local search, meaning *best* set of routes in term of cost.

5.1.3 Initialize

Function ***initialize*** *(LinehaulSet, BackhaulSet, k, C, max_iter, BestCost, BestRouteSet)*
Begin
LinehaulSet ← Load the set of Linehaul Customer
BackhaulSet ← Load the set of Backhaul Customer
k ← Load the number of routes That form the final solution
C ← Load the capacity of the truck
max_iter ← Load the number of restart of the GRASP Algorithm
BestCost ← Set to infinity the actual value of objective function
BestRouteSet ← Set to Empty the set of k best route
end

The initialize function initializes the set of Linehaul customers indicated with *LinehaulSet*, the set of Backhaul customers indicated with *BackhaulSet*, the k number of routes that the number of routes that will compose the final solution. C represent the load capacity of a truck (which in reality is read from the instance), the number of iterations *max_iter*, the lowest cost for the routes found by the heuristic after *max_iter* iterations, initially set equal to infinity, that is indicated with *BestCost* and finally the best set of routes indicated with *BestRouteSet*.

5.1.4 Construction phase

Function ***constructionPhase*** *(LinehaulSet, BackhaulSet, k, C)*
Begin
PairSet ← Set to empty the k pairs of Linehaul/Backhaul client
RouteSet ← Set to empty the set of k route
PairSet ← ***createPair*** *(LinehaulSet, BackhaulSet, k)*
RouteSet ← ***createRoutes*** *(LinehaulSet, BackhaulSet, PairSet, C)*
return RouteSet
end

The constructionPhase procedure receives as parameters the two sets of customers: the Linehaul set (*LinehaulSet*), and the Backhaul set (*BackhaulSet*), the k number required to create k routes and the capacity of the truck, C. Within the procedure are initialized to empty the $| k |$ set of pairs that will be created, called *PairSet*, and the set of $| k |$ routes that will be returned, indicated by *RouteSet*. Then come two procedure calls. The *createPair* procedure that input the set of

Linehaul *LinehaulSet* customers, the set of Backhaul *BackhaulSet* customers and k number of routes that you want to create and return a *PairSet*, which is a set of couples (Linehaul, Backhaul). The procedure *createRoutes*, which is passed the set of Linehaul *LinehaulSet*, the set of Backhaul *BackhaulSet*, the pairs of *PairSet* returned by the previous procedure and the capacity of the truck C, and returns a set of routes indicated by *RouteSet*. The main procedure *constructionPhase* returns *RouteSet*, i.e. a set of eligible routes.

5.1.5 Creation of couples

As anticipated, in this GRASP metaheuristic it is possible to select the number of trucks that is required to use, and thus the number of different routes that we want to have in our final solution. Obviously, there will be a minimum number of necessary trucks to cover all customers, determinable by solving a Bin Packing problem.

Function ***createPair*** *(LinehaulSet, BackhaulSet, k)*
Mode: a parameter that control the selection of the most promising neighbor
Begin
    SelectedSet ← ***smaller*** *(LinehaulSet, BackhaulSet; if equal useBackhaulSet)*
    RemainingSet ← ***notUsedBetween*** *(SelectedSet, LinehaulSet, BackhaulSet)*
    SelectedRCLList ← ***createRCLList*** *(SelectedSet, RemainingSet, mode)*
    SelectedPair ← ***createAllPair*** *(SelectedSet, SelectedRCLList)*
    SavingList ← ***computeSavings*** *(SelectedPair)*
    OrderedSavingList ← *Order SavingList in a not-growing way*
    PairSet ← *SelectedPair ordered by SavingList*
        if k >| SelectedSet |
            residualPair ← ***createResidualPair*** *(k - | SelectedSet |, RemainingSet*
            PairSet ← ***merge*** *(PairSet, residualPair)*
        endif
    return the first k values of PairSet;
end

The basic idea is to create all of customer pairs (a Linehaul with a Backhaul) possible, according to the number of the instance customers taken into consideration. For example, referring to an instance of 33 nodes asymmetric VRPB, the possible configurations are showed in table 5.1.
The *createPair* procedure receives as parameters the set of Linehaul customers *LinehaulSet*, the set of Backhaul customers *BackhaulSet*, and k number of routes that is required. In general, the two sets don't have the same cardinality. Due to this fact, in order to be able to compose a *complete* set of customer pairs (Linehaul, Backhaul), the smallest of the two sets between *LinehaulSet* and *BackhaulSet* is chosen as *selectedSet*. *RemainingSet*, for exclusion, is equal to the other set. If the two sets have the same cardinality, the *BackhaulSet* is chosen as the *SelectedSet*.

Table 5.1: Distribution of the nodes

BH Rate	*BH number*	*LH number*
50%	16	17
66%	22	11
80%	26	7

To give a practical example, solving an instance with a number of Backhaul equal to 50% of the total of 33 customers, 16 pairs (Linehaul, Backhaul) will be created, in an instance with Backhaul to 66% of customers, 11 pairs will be created and in the third type of instance with Backhaul equal to the 80% of customers, 7 customer pairs will be created. As already described before, each element of the *SelectedSet* will be the first component of a eligible couple.
Different cases are evaluated: if the *SelectedSet* is composed by Linehaul customer, only a Backhaul customer can be considered as eligible, to perform a couple. If in *SelectedSet* is stored the set of bachauls, then a predecessor node is searched in the set of Linehaul customer, and used to create the couple.

Once identified the *SelectedSet* and *RemainingSet*, an RCL list is created of avaliable nodes for each customer (or node) of the *SelectedSet*.

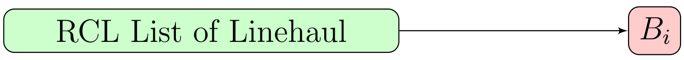


Figure 5.1: Single RCL creation, for a node of $SelectedSet = BackhaulSet$)

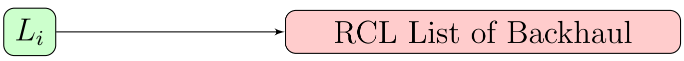


Figure 5.2: Single RCL creation, for a node of $SelectedSet = LinehaulSet$

The $createRCLList$ procedure receives as parameters the *selectedSet* and *remainingSet* and returns the RCL list for each element of *SelectedSet*, indicated $SelectedRCLList$. In this way, starting from a set of nodes, the output is a set of RCL, each one related to corresponding node of the *SelectedSet*. Initially all customers of *RemainingSet* are placed in the list RCL. It is necessary to establish an evaluation criterion useful to classify and sort the customer in the RCL list. At this step, the Euclidean distance between the current customer (or node) of *SelectedSet* and the customer (or node) of the RCL list is used to classify nodes into RCL list. During this phase the customers within the RCL list are ranked by an increasing distances criterion, so in the first position, there will be the closest node to the node of the *SelectedSet* concerned. In this phase, every RCL list contains the same elements. What changes is the order, because as mentioned above, it depends on the distance from the current node.

Function ***createRCLList*** *(SelectedSet, RemainingSet, mode)*
Begin
    i ← 0
    for each node i in SelectedSet
        SelectedRCLList (i) ← insert all nodes of RemainingSet in according to the value of mode
    end foreach
    return SelectedRCLList
end

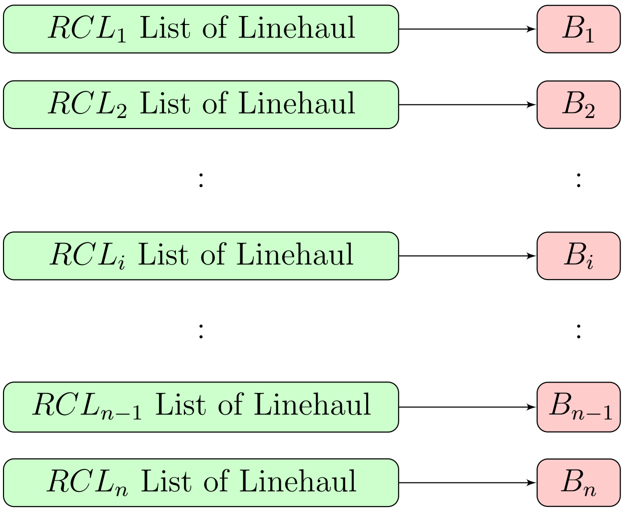


Figure 5.3: RCL lists creation, $SelectedSet = BackhaulSet$

Function *createRCLList* ends with a collection of RCL lists, one for each element of the *SelectedSet*. The next step is nothing more than the creation of pairs.

Function ***createAllPair*** *(SelectedSet, SelectedRCLList)*
Begin
    i ← 0
    for each node in SelectedSet
        i ← i + 1
        if mode← fixNumber
        BestMatch ← ***rouletteWheelSelection*** *(SelectedRCLList (i))*
        SelectedPair (i) ←(node, BestMatch)
        endif
    endfor

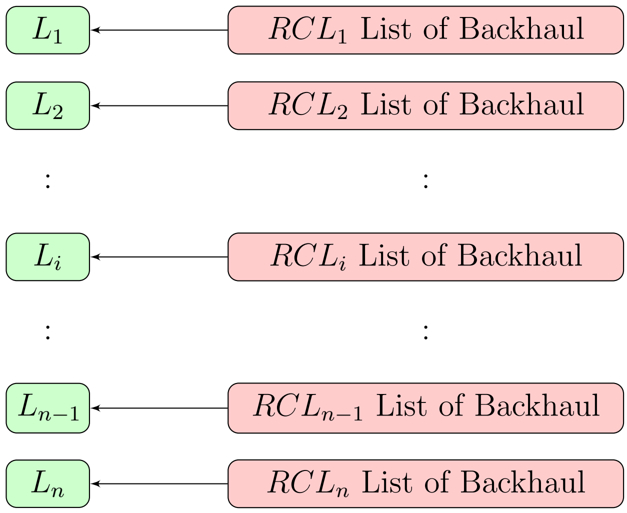


Figure 5.4: RCL lists creation, $SelectedSet = LinehaulSet$)

return $SelectedPair$
end

The $createAllPair$ procedure receives the set of customers contained in $SelectedSet$ and the collection of related RCL list, for each element of $SelectedSet$. This collection is indicated as $SelectedRCLList$. A counter i is initialized to 0 and is used to move through the elements of the $SelectedSet$, with the aim of finding, for each of them, a node with which to pair, taking it from the corresponding RCL list. The procedure returns the set of pairs that have been created, indicated $SelectedPair$. The node that must be associated with the current item of $SelectedSet$ is chosen randomly from the corresponding RCL list, using the roulette wheel selection mechanism, and is indicated as $BestMatch$. For node extraction three variants have been used:

First variant

In the first variant, a value of probability is assigned to each node of the RCL list, that is inversely proportional to its distance of the current node. In this way, the closest customer and thus closer to the element of $SelectedSet$, will have a greater chance. In the RCL List, distances having previously been ordered in an increasing way, and consequently probabilities are already ordered in a not-growing order. The random aspect of the heuristic is related to the extraction of a random element from the RCL list. A random number $p \in [0, 1]$ is thus extracted, and a sum of the probabilities of the elements of the RCL List is performed until the sum is not at least equal to the random number p. As soon as the sum becomes greater than or equal to p, the index of the actual element of the RCL List is saved, and it corresponds to the value returned by $rouletteWheelSelection$ procedure.

Second variant

In the second variant there is an additional step compared to the first. After obtaining the corresponding probabilities for each customer of the RCL list, a sub-list is extracted from the list of all customers, using the first elements whose sum of the probability is 0.7. In other words in this variant it is not considered the entire probability range from 0 to 1, but only the 70% of the range. We need that the sum of probabilities is equal to 1, and a new set of values is computed for customers included in the 70% and re-compute the probability inversely proportional to the distance for each customer. Once allocated to each customer his probability, a random number $p \in [0, 1]$ is generated, and, like the first variant, the probabilities are added until the sum is at least equal to the random number p. As soon as the sum becomes greater than or equal to p, it saves the corresponding index of the RCL list, and return them.

Third variant

In the third variant, as well as in the second, there is an additional step compared to the first. After obtaining the corresponding probabilities for each customer of the RCL list, a sub-list of the 5 most promising nodes is extract. A new RCL list with 5 equiprobable customers is generated. An extraction of a integer random number p between 1 and 5 which becomes the index of the RCL list that we want to extract. It saves the index to return the corresponding element.

A particular attention is required in all variants, because the $createAllPail$ have to manage two aspects that are dynamic during the iterations: the first is that if a node already have been used during the iteration, it is not available for the next sub-RCL list, and the second problem is that in the last steps of the sequence, due to the fact that many nodes were used, final RCL Lists can have a short length, until it could happen that only one element can be available for the last one. One of these variants has been used during all iterations. As anticipated, the element of RCL list, referred to as $BestMatch$, whose index corresponds to the extracted random number, will be paired with the corresponding *selectedSet* element. And, as mentioned above, it should be noted that once awarded the $BestMatch$, it will be deleted from the lists of the elements of the RCL *SelectedSet* that still be used.

The phase of creation of pairs (Linehaul, Backhaul) is now completed. For the next deployment step is necessary to introduce the concept of saving.

Savings and choice of couples

The concept of saving was introduced by a constructive heuristic, derived from savings algorithm (Clarke and Wright, [CW64]). The basic idea of saving is to save on the cost of fusing routes. To better explain, a practical example is analyzed. In this example, the depot is indicated with d, while i and j are generic customers.

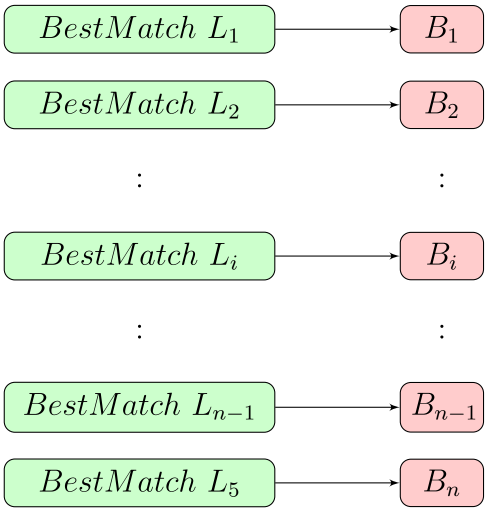


Figure 5.5: Couples extraction, $SelectedSet = BackhaulSet$)

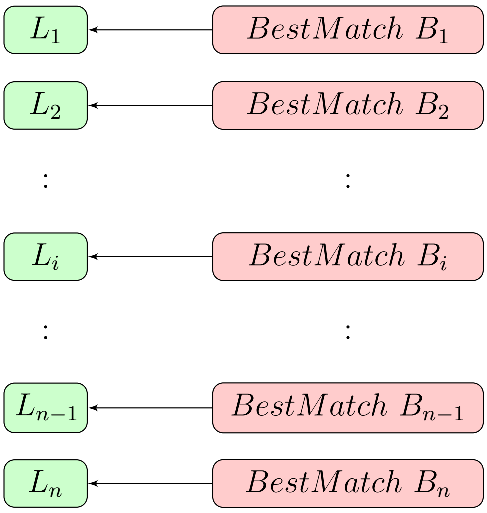


Figure 5.6: Couples extraction, $SelectedSet = LinehaulSet$)

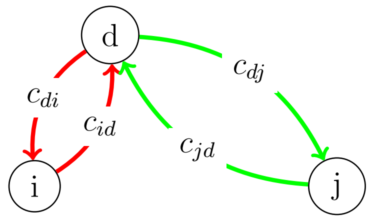


Figure 5.7: Case a): route costs without saving

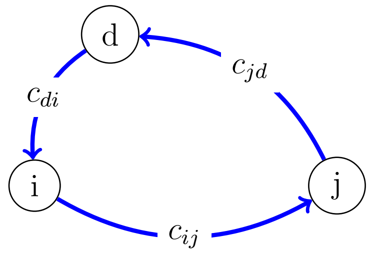


Figure 5.8: Case b): route costs with saving

The goal is to visit customers i, j starting from the depot. In figure 5.7 customers are visited on separate routes, each one starts from the depot and arrives to the customer. So, after serving customer i, the truck returns to the depot and after leaves to customer j, and finally returns to the depot. There would be an alternative, that is to visit both clients on the same route, as shown in figure 5.8. The cost to go from the depot to a customer or to a customer to another customer, can be known, in the problem, or it can be calculated. The question is whether there is a convenience in visiting the j customer immediately after the customer i without going back to the depot.
Indicating with c_{ij} the cost to go from the customer i to the customer j, we have:
The total cost turns out to be:
Case a: $= c_{di} + c_{id} + c_{dj} + c_{jd}$
Case b: $= c_{di} + c_{ij} + c_{jd}$

To find the saving S_{ij} visiting the j customer immediately after the customer i without returning the depot, it is calculated:
S_{ij} = Case a - Case b
$= (c_{di} + c_{id} + c_{dj} + c_{jd})$ - $(c_{di} + c_{ij} + c_{jd})$
$= c_{id} + c_{dj} - c_{ij}$
A great value of S_{ij} indicates that there is an high convenience from the cost point of view to visit the i and j customers on the same route. The concept of saving, as brief explained, is used in the implementation of the metaheuristic.
The *createAllPair* function creates a pair for each element of the selected set,

that is the less numerous set between Backhaul and Linehaul customers. In general, the number of required routes k (that will compose the final solution) is smaller than the cardinality of the *SelectedSet*. So, it is necessary to select a set of couple, from the all ones. For each pair created earlier it is evaluated the savings if the truck consecutively visits customers of the couple without going back to the depot. The computed savings are then used to rank couple in a decreasing way, because as has been said, a great value savings indicates a certain convenience in performing that route, and everything is stored in $PairSet$. At this point the first k pairs of PairSet are selected, which will give rise to exactly k routes.

Function ***createResidualPair*** *(k - | SelectedSet |, RemainingSet)*
Begin
  RemainingSet ← *Set to empty the pair k of client Linehaul / Backhaul*
  ResidualPair ← ***createResidual*** *(Depot, k - | SelectedSet |, RemainingSet)*
  RouteSet ← ***createRoutes*** *(LinehaulSet, BackhaulSet, PairSet, C)*
  return ResidualPair
end

There may be a special case, because k may be greater than the cardinality of *SelectedSet*. In this case you are not able to select exactly k routes. The solution proposed is to create the $k - |SelectedSet|$ remaining pairs, indicated by $residualPair$, using customers from *RemainingSet*. The *createResidualPair* procedure receives as parameters a number indicating how many are the missing couples, that is $k - |SelectedSet|$ and receive the *RemainingSet*. First, it order the customers of *RemainingSet* by increasing distances to the depot. Then, through *createResidual* procedure are created $k - |SelectedSet|$ couples, taking the depot and the first $k - |SelectedSet|$ customers of *RemainingSet*. The $k - |SelectedSet|$ have finally returned with remaining couples, $residualPair$. At this point it is possible to merge the couples, adding the pairs of $residualPair$ to those of $PairSet$. Now it is concluded the construction process of couples, with the return from the procedure $createPair$ the set of pairs selected from $PairSet$.

Assuming that the final solution must contain 3 routes, $k = 3$, i.e. three customer pairs have thus been generated (Linehaul, Backhaul). Now the set of Linehaul (*LinehaulSet*) and the set of Backhaul (*BackhaulSet*) will contain the starting elements, the elements that have been assigned to the pairs. Generalizing the two cases it can be said to have 3 customer couples to start, see figure 5.11

5.1.6 Creating routes

The *createRoutes* procedure receives as parameters the set of Linehaul $LinehaulSet$, the set of Backhaul $BackhaulSet$, the set of selected pairs $PairSet$ and truck capacity C.

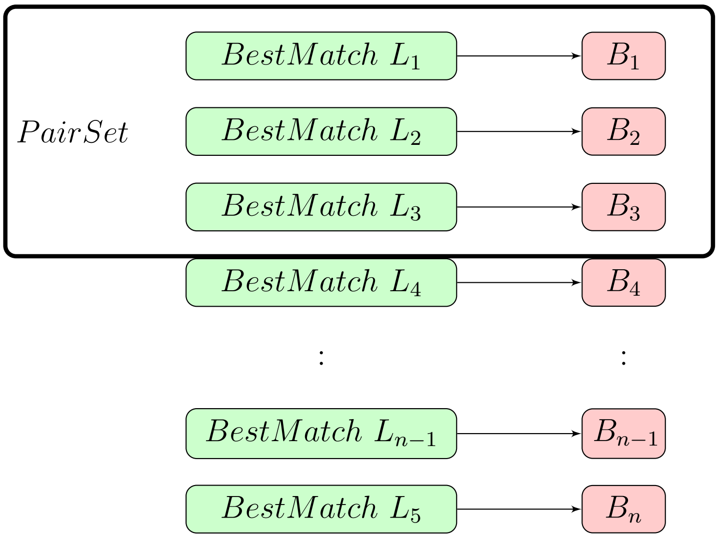


Figure 5.9: $PairSet$ selected for $k = 3$ with $SelectedSet = BackhaulSet$

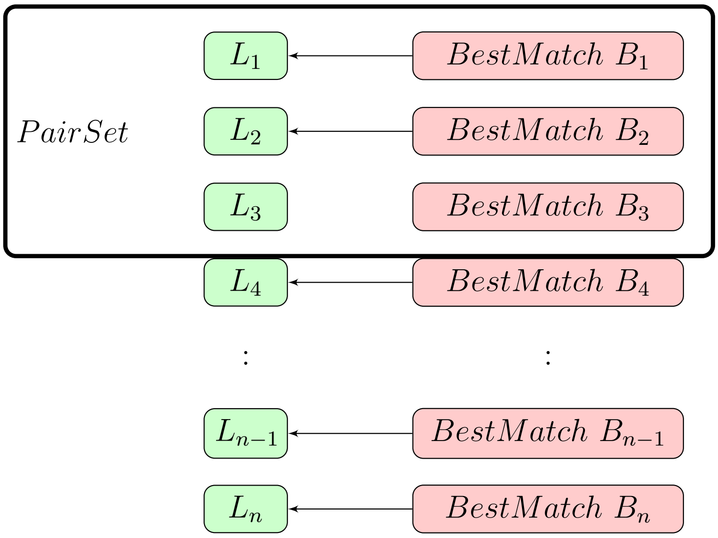


Figure 5.10: $PairSet$ selected for $k = 3$ with $SelectedSet = LinehaulSet$

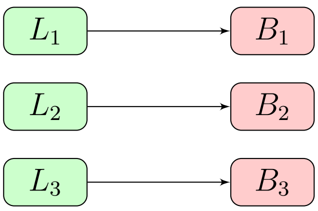


Figure 5.11: $PairSet$ from which start to create routes

Function ***createRoutes*** *(LinehaulSet, BackhaulSet, PairSet, C)*
Begin
  ResidualLinehaulSet ← *LinehaulSet*
  ResidualBackhaulSet ← *BackhaulSet*
  RouteSet ← *Set to Empty*
  for each pair in PairSet
    firstBHnode ← *Backhaul node from pair*
    BHRoute ← ***constructBHRoute*** *(firstBHnode, ResidualBackhaulSet, C)*
    lastLHnode ← *Linehaul from node pair*
    LHRoute ← ***constructLHRoute*** *(lastLHnode, ResidualLinehaulSet, C)*
    update (ResidualBackhaulSet, ResidualLinehaulSet)
    singleRoute ← ***merge*** *(LHRoute, BHRoute)*
    RouteSet ← *RouteSet + singleRoute*
  end foreach
  for each ResidualElement in ResidualLinehaulSet or in ResidualBackhaulSet
    RouteSet ← *RouteSet +* ***createDirectRoute*** *(ResidualElement, depot)*
  endforeach
  return RouteSet
end

Linehaul and Backhaul customers set that are not part of the $pairSet$ are inserted into two sets called $ResidualLinehaulSet$ and $ResidualBackhaulSet$, respectively. It is also initialized to the empty set of routes indicated with $RouteSet$. For each pair of $PairSet$, the Backhaul node and the Linehaul node are treated separately. It begins with the extraction the Backhaul of the first pair node, in this case B_1, and a semi-route of Backhaul customers from the same B_1 they are processed separately, through $constructBHRoute$ procedure. The $constructBHRoute$ procedure receives as parameters the first Backhaul node, $firstBHnode$, which in this case is the Backhaul node of the first pair, the $ResidualBackhaulSet$ and the capacity of the truck, C.

Function ***constructBHRoute*** *(firstBHnode, ResidualBackhaulSet, C)*
Begin
  currentNode ← *firstBHnode*
  BHRoute ← *firstBHnode*
  residualCapacity ← *C - currentNodeDemand*
  while $(residualCapacity > 0)$
    RCLlist ← ***computeBHRCL****(currentNode, ResidualBackhaulSet, residual-Capacity, C)*
    newNode ← ***rouletteWheelSelection*** *(RCLlist)*
    BHRoute ← *BHRoute + newNode*
    residualCapacity ← *residualCapacity - newNodeDemand*
    currentNode ← *newNode*

update *(ResidualBackhaulSet)*
endwhile
return BHRoute
end

At first, it is assigned to the current node, denoted by $currentNode$, the Backhaul node of the first $firstBHnode$ couple. It is then inserted into the route of the Backhaul that is being built, as the first node, the $firstBHnode$. After every insertion, the remaining capacity is updated, $residualCapacity$, subtracting the capacity of the truck the offer (or demand) of current node $currentNodeDemand$. The process of construction of the Backhaul route is guided by a while loop that checks the residual load capacity on the truck, $residualCapacity$, which decreases as the nodes are added. As long as the remaining capacity is greater than zero, it creates a list RCL for the current node $currentNode$. Initially the current node is B_1, see figure 5.12

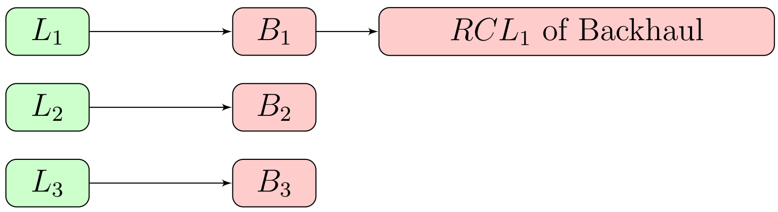


Figure 5.12: RCL construction for the Backhaul node of the first pair

The RCL list is created taking into account some details, described in the next paragraph. It can however anticipate that also in this case to each element of the RCL list is associated a variable probability. The element that will add to the route, indicated by $newNode$, is randomly chosen from the RCL list that corresponds to the current node, using the same *roulette wheel selection* mechanism. As before, three variants have been used.

First variant

In the first variant a random number $p \in [0, 1]$ is generated and the single probabilities associated at all elements of the RCL List are added until the sum is not at least equal to p. As soon as the sum becomes greater than or equal to p, the current RCL list index is saved and corresponding to the element to be returned by the $rouletteWhellSelection$ procedure.

Second variant

The second variant select from the list all the elements until the sum of the probability is 0.7. In this case are not considered available all nodes of the $remainingSet$, but his 70%. A sub-RCL list is recreated, but limited to 70% of the extracted elements. Like the first variant, a random number $p \in [0, 1]$ is generated, and the sum of the probabilities until the sum was at least equal to p is computed. As soon as the sum becomes greater than or equal to p, the corresponding index of the sub-RCL list is saved to be returned by the $rouletteWheelSelection$ procedure.

Third variant

The third variant select from the candidate list the first 5 elements, namely the 5 elements with the highest probability if the size of the RCL list is ≥ 5. If the size is lower than 5 elements, all elements are considerate. Extrapolated elements are made equally probable and extracted a random element, saving the index corresponding to be returned by $rouletteWheelSelection$ procedure.
Like the couple generation, only one of the three strategies is used. The three strategies share the fact that only the nodes that have a compatible request with the remaining load capacity are used to compose the RCL lists. Once a customer is extracted, the $newNode$, it can be added to the route of the Backhaul, $BHRoute$. See figure 5.14.

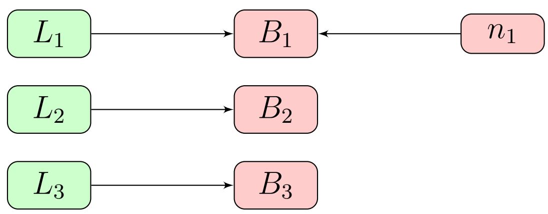


Figure 5.13: Insertion of the first node in the route of the Backhaul of the first pair

Now one updates the remaining capacity of the truck, $residualCapacity$, which has decreased because of the offer (or demand) of n_1 node. The $newNode$ becomes the current node, n_1. It is updated $ResidualBackhaulSet$, eliminating the node just inserted into the route. A new RCL is created for the new inserted node n_1.

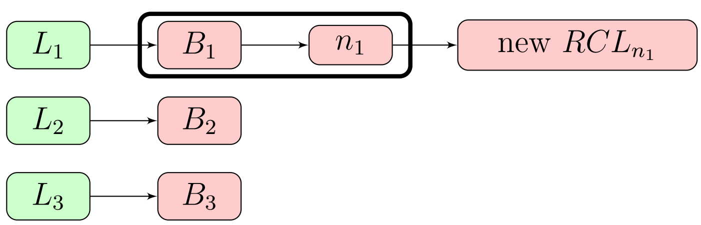


Figure 5.14: RCL Construction for Backhaul node n_1

The algorithm repeats these steps, until it is no longer possible to add nodes to the route for capacity reasons.

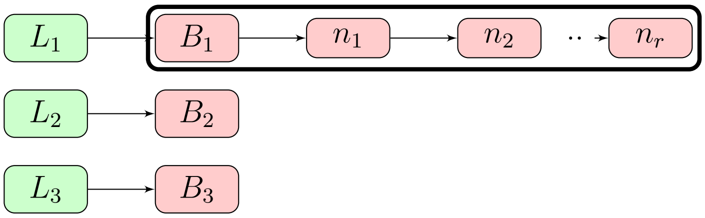


Figure 5.15: Example of BHRoute

The *constructBHRoute* function returns the Backhaul route just created, called *BHRoute*. This is an open-route that is going to be linked to another open-route made using Linehaul nodes. *FunctionconstructLHRoute* starts from the other node of the couple that is processed. See figure 5.15.

```
Function constructLHRoute (lastLHnode, ResidualLinehaulSet, C)
Begin
    currentNode ← lastLHnode
    LHRoute ← lastLHnode
    residualCapacity ← C - currentNodeDemand
    while (residualCapacity > 0)
        RCLlist ← computeLHRCL(currentNode, ResidualLinehaulSet, residual-
Capacity, C)
        newNode ← rouletteWheelSelection (RCLlist)
        LHRoute ← LHRoute + newNode
        residualCapacity ← residualCapacity - newNodeDemand
        currentNode ← newNode
        update (ResidualBackhaulSet)
    endwhile
    return LHRoute
end
```

At first it is assigned to the current node, denoted by *currentNode*, the Linehaul node of the first *lastLHnode* couple. Then it is inserted into the route of Linehaul that is being created, as the last node, just the *lastLHnode*. It then updates the remaining capacity of the truck, *residualCapacity*, subtracting the demand *currentNodeDemand* of the current node. The process of construction of the Linehaul route is guided by a while loop that checks the residual capacity of the truck, *residualCapacity*, which decreases as the nodes are added. As long as the remaining capacity is greater than zero, it creates a list RCL for the current node *currentNode*. Initially the current node is L_1.

As for the case of the Backhaul, an RCL list is created taking into account some details, and associating to each element of the RCL list a probability.
The element which will add to the route, indicated by *newNode*, is chosen ran-

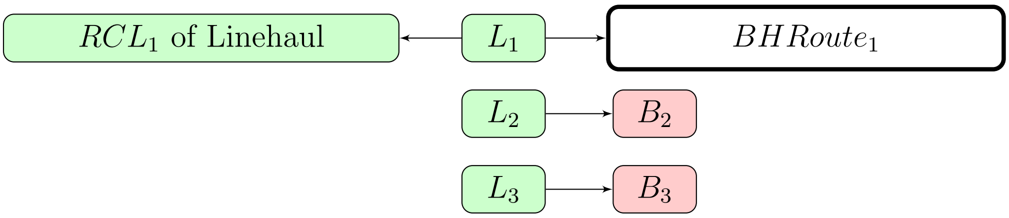


Figure 5.16: RCL Construction for Linehaul node of the first pair

domly from the RCL list corresponding to the current node, using the same *roulettewheelselection* mechanism. As before, three variants are proposed, in a similar way of the case of Backhaul.
Only one of the three strategies is used, in particular the same strategy used for the creation of the first couples and for Backhaul route. Once extracted, the *newNode* can be added to the open route of Linehaul *LHRoute*. See figure 5.17

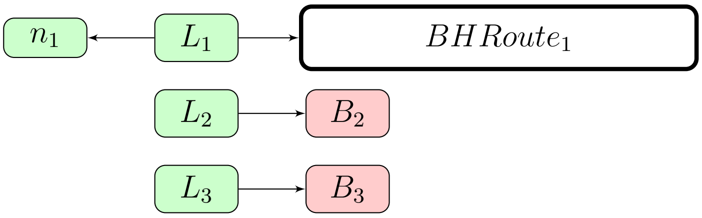


Figure 5.17: Insertion of the first node in the Linehaul route

At this point one updates the residual capacity of the truck, *residualCapacity*, which is decreased because of the node n_1 request. The newNode becomes the current node, n_1. It is updated ResidualLinehaulSet, eliminating the node just inserted into the route.
It then to calculates the RCL for the new inserted node n_1.

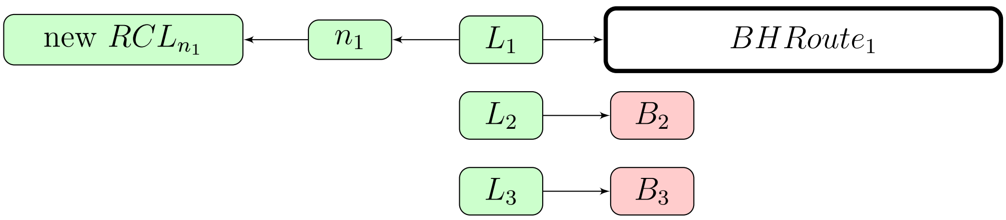


Figure 5.18: RCL Construction for Linehaul node n_1 of the first pair

The algorithm goes on until it is no longer possible to add nodes to the route for capacity reasons.
The *constructLHRoute* procedure returns the Linehaul route just created, called *LHRoute*. See figure 5.19
After obtaining 2 open-routes, the *createRoutes* procedure performs the merge of the Linehaul customers *LHRoute*1 and the route of Backhaul customers *BHRoute*1

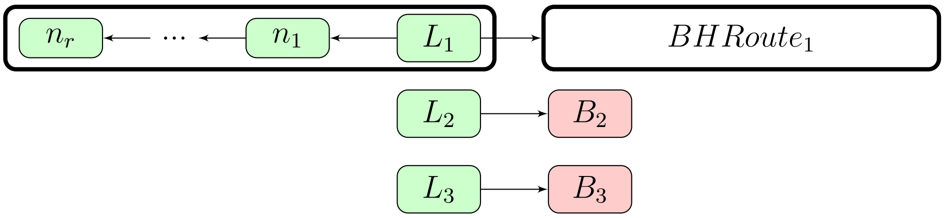


Figure 5.19: Example of LHRoute

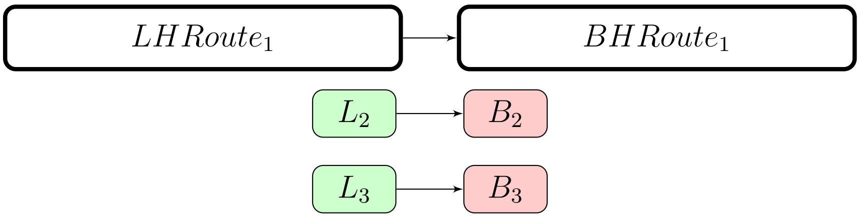


Figure 5.20: *LHRoute* and *BHRoute* from first pair

route, creating the *singleRoute*1, the first one of the set of routes routeSet. The *singleRoute*1 is added to *RouteSet*. See figure 5.22

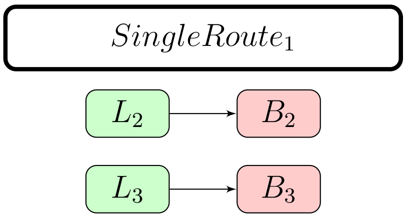


Figure 5.21: *LHRoute* and *BHRoute* merge

The process is repeated for all pairs of *PairSet*, adding all the routes found to the set of routes *routeSet*.

Some customers may remain in *ResidualLinehaulSet* or *ResidualBackhaulSet*, because it failed to enter in the previously created routes. In this case, *direct* routes are created: depot $\rightarrow$ *ResidualElement* $\rightarrow$ depot, for each customer that is in a *residual set*. Each direct route created is then added to *routeSet*, and the local search phase will manage the overabundant number of routes. The aim of function *createRoutes* is to return a collection of feasible routes, *routeSet*, at the end of the construction phase, that form a feasible solution useful to *constructionPhase* function.

Calculation of the RCL when creating routes

.

Function ***computeBHRCL*** *(currentNode, ResidualBackhaulSet, residualCapacity, C)*

$SingleRoute_1$

$SingleRoute_2$

$\vdots$

$SingleRoute_k$

Figure 5.22: RouteSet of k initial pair

Begin
    weightSum $\leftarrow$ *0*
    $\beta \leftarrow 1$
    $\gamma \leftarrow 1$
    foreach Node in ResidualBackhaulSet with demand $<= residualCapacity$
        $\alpha i \leftarrow$ *residualCapacity / C*
        $\omega_{ij} \leftarrow \alpha_i * (1/distance(i,j) + (1-\alpha_i) * (1/(\beta * distance(i,j) + \gamma * distance(j,0)))$
        $weightSum \leftarrow weightSum + \omega_{ij}$
        Insert Node ω_{ij} *in RCLList sorted by* ω_{ij}
    end foreach
    foreach Node in RCLList
        $p_{ij} \leftarrow \omega_{ij}$ */ weightSum*
        insert p_{ij} *into RCLList*
    end foreach
    return RCLList
end

Function ***computeLHRCL*** *(currentNode, ResidualLinehaulSet, residualCapacity, C)*
Begin
    weightSum $\leftarrow$ *0*
    $\beta \leftarrow 1$
    $\gamma \leftarrow 1$
    foreach Node in ResidualLinehaulSet with demand $<= residualCapacity$
        $\alpha i \leftarrow$ *residualCapacity / C*
        $\omega_{ij} \leftarrow \alpha_i * (1/distance(i,j) + (1-\alpha_i) * (1/(\beta * distance(i,j) + \gamma * distance(j,0)))$
        $weightSum \leftarrow weightSum + \omega_{ij}$
        Insert Node ω_{ij} *in RCLList sorted by* ω_{ij}
    end foreach
    foreach Node in RCLList
        $p_{ij} \leftarrow \omega_{ij}$ */ weightSum*
        insert p_{ij} *into RCLList*

end foreach
return RCLList
end

Functions $computeBHRCL$ and $computeLHRCL$ performs the same task. In the input parameters, both receive the current node $currentNode$ and, related to the truck, the remaining capacity $residualCapacity$ and the maximum load capacity C. They differ in the parameter of the whole residue, which in the first case is the $ResidualBackhaulSet$ while the second case is the $ResidualLinehaulSet$. As output, they return the RCL List associated with the current node.
Only nodes in $ResidualBackhaulSet$ ($ResidualLinehaulSet$) whose offer (demand) is $\leq residualCapacity$ are considered. This is because it is unnecessary to consider nodes that do not meet the eligibility conditions. It is first assigned to a parameter, α_t, that depends on the current node, a value corresponding to the ratio between the residual capacity and the capacity of the truck.
Then, using an *amount-sensitive* approach, the $\omega_{t|u}$ parameter is calculated, which is the sum of two quantities:

- the first quantity is the ratio between α_t and the distance D_{tu} (between the current node t and the node-to-reach u)
- the second quantity is the ratio between 1 - α_t and the sum of D_{tu} and D_{u0} distances (D_{u0} is the distance between the node u and the depot)

Clearly, the first part of the sum increase for the u nodes closest to the current node. The second part of the sum instead takes into account the distance between the u and the *depot* node. This was done to promote the nodes closest to the depot when the remaining capacity is saturating.
At this point, one computes the sum of all the $\omega_{t|u}$ parameters found, and is stored in the variable $weightSum$. Finally each node is added at the RCL list, choosing as sorting values decreasing of $\omega_{t|u}$. The probability $p_{t|u}$, associate to a node u and related to a node t, is the ratio between the $\omega_{t|u}$ value associated and the value of the variable $weightSum$. In this way, nodes with a high $\omega_{t|u}$ value will have a higher probability.
The procedure can be summarized adopting the following five points:

1. compute the residual capacity of the truck, at the current node C_{Ri} and for each node j which satisfies: $d_j \leq C_{Ri}$

2. Determine the weight associated to the node u reachable from the node t (see par. 5.1) with the formula:

$$w_{t|u} = \alpha_t \frac{1}{D_{tu}} + (1 - \alpha_t) \frac{1}{\beta D_{tu} + \gamma D_{u0}}$$

where:

$$\alpha_t = \frac{C_{Rt}}{C_M}$$

C_{Rt} is the residual load capacity of the truck k after serving the node t, C_M is the capacity of the truck, and so it is possible to declare that $\alpha_t \in (0, 1]$, while $\beta = 1$ and $\gamma = 1$ are tuning parameters. The distance between two nodes i, j is expressed by the value $D_{i,j}$.

3. order nodes on a non-increasing way, using the $\omega_{t|u}$ weight criterium
4. calculate the sum of weights $\omega_{t|u}$ associated with each node of the RCL list, $weightSum$
5. associate to each node u of the RCL associated to the node t a probability $p_{t|u} = \omega_{t|u} / weightSum$

5.1.7 Local search

The local search proposed explores the search space of feasible routes and evaluates at the same time three types of moves in three neighborhoods:

- in the *Node Relocate* a node is moved from its current route and inserted into another route according the cheapest insertion criterium;
- in the *Node Exchange* a pair of nodes is swapped between two different routes;
- in *2-opt* removes two edges from a route and reconnects the two paths.

Both first improving and best improving moves in these neighborhoods are implemented. The local search was implemented by a Static Move Description (SMD), introduced in [ZK10], in order to reduce the complexity required for examining the neighborhoods. Combo moves, a mix of Node Relocate and Node Exchange, are used to avoid the entrapment in local minimum. Furthermore, first-improvement and best-improvement approaches are used in order for diversification/intensification the local search phase in the search space.

5.2 Second version

5.2.1 A GRASP for the VRPB with pre-processing

In this section, a pre-processing approach is presented on the GRASP metaheuristic, in order to improve the performances. In the first version, we propose three variant, regarding the construction of the RCL. In any case, however, it was necessary to

create a RCL from a subsets of node. The disadvantage is, despite we use efficient data structure, that the dynamic computing of list of weighed probabilities is deemed compute-intensive. The possible compromise is to use more memory space, and compute statically all possible (and *feasible*) RCL list, before the GRASP iteration, so that the metaheuristic only seek at data structures reducing the cpu time.
It is possible to see, in the experimental test in 6.2, that the running time, in the same set of benchmark, decreases by 83%, with a moderate lost of accuracy (+1.66%). The decrease in accuracy can be explained by the fact that in this version we compute statically all feasible RCL lists, using a greedy function that depends on the Euclidean distance from the nodes, and the *amount-sensitive* approach is less efficient in this static version than that used in (see section 5.1.6). Local search phase is very similar to that used in the first version. See section 5.1.7

5.2.2 Main

Function ***Enhanced_GRASP***
Begin
  initialize *(LinehaulSet, BackhaulSet, k, C, max_iter, BestCost, v, RCL_C, RCL_L, RCL_B, RCL_LD, RCL_BD)*
  $i \leftarrow 1$
  while $(i < max\_iter)$
    RouteSet ← ***constructionPhase*** *(LinehaulSet, BackhaulSet, k, C, v, RCL_C, RCL_L, RCL_B, RCL_LD, RCL_BD)*
    BestLocalRouteSet ← ***localSearch*** *(RouteSet)*
    if $\textbf{cost}(BestLocalRouteSet) < BestCost$
      BestRouteSet ← *BestLocalRouteSet*
      BestCost ← ***cost*** *(BestLocalRouteSet)*
    endif
    i ← i + 1
  endwhile
  return BestRouteSet
end

This section describes the main function, called *Enhanced_GRASP*. The idea of a sequence of iterations, made up from successive constructions of a greedy randomized solution, and subsequent iterative improvements of it by a local search is preserved. We can observe that the flow is similar to that explained in the main function of the first version 5.1.2
In this version, the main difference is that the *initialize* function compute, in pre processing, all feasible RCL, in according to 2.4.

5.2.3 Initialize

*Function **initialize**(LinehaulSet, BackhaulSet, k, C, max_iter, BestCost)*
Begin
LinehaulSet ← Load the set of Linehaul Customer
BackhaulSet ← Load the set of Backhaul Customer
k ← Load the number of routes that form the final solution
C ← Load the capacity of the truck
max_iter ← Load the number of restart of the GRASP Algorithm
BestCost ← Set to infinity the actual value of objective function
RCL_C ← Load RCL for couple: calculate in preprocessing
RCL_L ← Load RCL for Linehaul: calculate in preprocessing
RCL_B ← Load RCL for Backhaul: calculate in preprocessing
RCL_LD ← Load RCL for Linehaul that contains also informations by depot distance: calculate in preprocessing
RCL_BD ← Load RCL for Backhaul that contais also informations by depot distance: calculate in preprocessing
end

5.2.4 Feasible RCL Lists computing

*Function **createRCL_C** (LinehaulSet, BackhaulSet, k)*
Begin
mode ← 0
*SelectedSet ← **smaller** (LinehaulSet , BackhaulSet; if equal use BackhaulSet)*
*RemainingSet ← **notUsedBetween** (SelectedSet, LinehaulSet, BackhaulSet)*
if $|$ *BackhaulSet* $|< k$
Add the depot to the SelectedSet
endif
*RCL_C ← **createRCLList** (SelectedSet, RemainingSet, mode)*
return RCL_C
end

*Function **createRCL_L** (LinehaulSet, RemainingSet, mode)*
Begin
mode ← 1
SelectedSet ← LinehaulSet
RemainingSet ← null
*RCL_L← **createRCLList** (SelectedSet, RemainingSet, mode)*
return RCL_L
end

```
Function createRCL_B(BackhaulSet, RemainingSet, mode)
Begin
    mode ← 1
    SelectedSet ← BackhaulSet
    RemainingSet ← null
    RCL_B← createRCLList (SelectedSet, RemainingSet, mode)
    return  RCL_B
end

Function createRCL_LD (LinehaulSet, RemainingSet, mode)
Begin
    mode ← 2
    SelectedSet ← LinehaulSet
    RemainingSet ← null
    RCL_LD← createRCLList (SelectedSet, RemainingSet, mode)
    return RCL_LD
end

Function createRCL_BD(BackhaulSet, RemainingSet, mode)
Begin
    mode ← 2
    SelectedSet ← BackhaulSet
    RemainingSet ← null
    RCL_BD← createRCLList (SelectedSet, RemainingSet, mode)
    return  RCL_BD
end

Function createRCLList(SelectedSet, RemainingSet, mode)
Begin
    foreach node in SelectedSet
    if mode ← 0
        compute t nodes distance from node i and RemainingSet
        SelectedRCLList (i) ← set of t nodes
    if mode ← 1
        compute t nodes distance from node i and SelectedSet
        SelectedRCLList (i) ← set of t nodes
    if mode ← 2
        compute t nodes distance of depot and nodes i in SelectedSet
        SelectedRCLList (i) ← set of t nodes
    end foreach
    SelectedRCLList ← sort_by_decreasing_distance (SelectedRCLList)
    return SelectedRCLList
```

end

During a pre-processing phase, all feasible RCL List are computed; in the mathematical model in 2.4 we define A as the set of *admitted* arcs: $A = A_1 \cup A_2 \cup A_3$ where:

- $A_1 = \{(i,j) : i \in L_0,\ j \in L\}$

- $A_2 = \{(i,j) : i \in B,\ j \in B_0\}$

- $A_3 = \{(i,j) : i \in L,\ j \in B_0\}$

Using this notation, to better explain the creation of the RCL lists, we associate the single functions to the right subset of A.

- arcs in A_1 are made by the join of *createRCL_L* and *createRCL_LD*

- arcs in A_2 are made by the join of *createRCL_B* and *createRCL_BD*

- arcs in A_3 are made by *createRCL_C* (Couples)

Finally, the multi-purpose *createRCLList* function is called by previous functions, and the control variable *mode* is used to lead the right RCL list construction. Another control variable, used in *createRCLList*, is t. As in the first version, three variants are proposed, regarding the size of the RCL List:

- $t = 1$, all available nodes are used to create the RCL List. For each node i a probability p_i is computed with a greedy function related to the distance, and associated to i. After, a ranking of all nodes is used to create the RCL List (tightly *greedy*).

- $t = 2$, the same behavior of $t = 1$. A sum of p_i of ranked list is made, and only the nodes that are in the sum equal or less of 0,7 are used. In other words, the first 70% of nodes are used, not in term of node number, but in term of their probability.

- $t = 3$, only 5 nodes (if available), starting from the most probable (generated like case $t = 1$), are used to create the RCL List. In this case, after the selection, all nodes are set to be equally probable (tightly *random*).

5.2.5 Construction phase

Function ***constructionPhase*** *(LinehaulSet, BackhaulSet, k, C, RCL_C, RCL_L, RCL_B, RCL_LD, RCL_BD)*
Begin
  PairSet ← *Set to Empty the k pair of client Linehaul/Backhaul*
  RouteSet ← *Set to Empty the set of k route of the final solution*
  PairSet ← ***CreatePair*** *(k, RCL_C)*
  RouteSet ← ***CreateRoutes*** *(PairSet, C, RCL_L, RCL_B, RCL_LD, RCL_BD)*
  return RouteSet
end

The *constructionPhase* is very similar to that described in 5.1.4. The main difference is that all pre-computed RCL Lists are passed as parameters, and not dynamically computed during the iterations.

5.2.6 Creation of couples

Function ***createPair*** *(k, RCL_C)*
Begin
  SelectedPair ← *createAllPair (RCL_C)*
  If k >| SelectedPair |
    residualPair ← ***CreateResidualPair*** *(k - | SelectedPair |, RCL_C(depot))*
    SelectedPair ← ***merge*** *(SelectedPair, residualPair)*
  End if
  SavingList ← ***ComputeSavings*** *(SelectedPair) and order it in a not-growing way*
  PairSet ← *SelectedPair ordered in SavingList order*
  return the first k values of PairSet
end

Function ***CreateResidualPair*** *(k -| SelectedPair |, RCL_C(depot))*
Begin
  residualPair ← ***createResidual*** *(k -| SelectedPair |, RCL_C(depot))*
    made residual pair with couple Linehaul-depot
  return residualPair
end

Function ***createAllPair (RCL_C)***
Begin
  i ← *0*

for each node in RCL_C
i ← i +1
BestMatch ← ***rouletteWellSelection*** *(RCL_C (i))*
SelectedPair (i) ← (node, BestMatch)
end for
return SelectedPair
end

In this section, the creation of couples is explained. As well as the latest functions explained, the structure is similar to the corresponding one in the first version. In this case too, the main difference is that all pre-computed RCL Lists are passed as parameters, and not dynamically computed during the iterations.

5.2.7 Creation of routes

Function ***CreateRoutes****(PairSet, C, RCL_L, RCL_B, RCL_LD, RCL_BD)*
Begin
foreach pair in PairSet
firstBHnode ← Backhaul node from pair
lastLHnode ← Linehaul node from pair
BHRoute ← ***constructBHRoute*** *(firstBHnode, RCL_B , RCL_BD ,C)*
LHRoute ← ***constructLHRoute*** *(lastLHnode, RCL_L , RCL_LD, C)*
singleRoute ← ***merge*** *(BHRoute, LHRoute)*
RouteSet ← RouteSet + singleRoute
end foreach
RouteSet ← RouteSet + ***createDirectRoute*** *(ResidualElement, depot)*
return RouteSet
end

Function ***constructBHRoute*** *(firstBHnode, RCL_B , RCL_BD, C)*
Begin
currentNode ← firstBHnode
BHRoute ← firstBHnode
residualCapacity ← C - currentNodeDemand
while (residualCapacity is > 0)
*if residualCapacity < C*0,75*
newNode ← ***rouletteWheelSelection*** *(RCL_B)*
else newNode ← ***rouletteWheelSelection*** *(RCL_BD)*
BHRoute ← BHRoute + newNode
residualCapacity ← residualCapacity - newNodeDemand

```
        currentNode ← newNode
    endwhile
    return BHRoute
end


Function constructLHRoute(lastLHnode, RCL_L , RCL_LD, C)
Begin
    currentNode ← lastLHnode
    LHRoute ← lastLHnode
    residualCapacity ← C - currentNodeDemand
    while (residualCapacity is > 0)
        if residualCapacity < C*0,75
        newNode ← rouletteWheelSelection (RCLlist, RCL_L)
        else newNode ← rouletteWheelSelection (RCLlist, RCL_LD)
        LHRoute ← LHRoute + newNode
        residualCapacity ← residualCapacity - newNodeDemand
        currentNode←newNode
    endwhile
    return LHRoute
end
```

Function *CreateRoutes*, starts from a set of couples, *PairSet*, and creates two open route, using *constructBHRoute* end *constructLHRoute* functions respectively. The *roulette wheel selection* mechanism is used to create the open routes, from this functions. Note that, when residual capacity is less then 25%, the *construct route)* functions switches from RCL Lists that are made only with a greedy function distance based, to the RCL lists made by the same set, but those one computed taking into account the depot too, with the *amount-sensitive* approach.

Chapter 6

Test and results

Benchmarking is a tool for evaluating performance. In heuristic field, an exact value useful for comparison term often is not available. In the literature different classes of benchmark instances are used to experimentally compare the performance of exact and heuristic algorithms proposed. The experimentation of the metaheuristic is carried out using two different sets of instances, symmetric and asymmetric. Furthermore, a tuning phase is necessary to set the metaheuristic ready to perform. For example, it is necessary to establish the number of iterations to run the GRASP on the specific instance. It is necessary to set a suitable number of restart to ensure a sufficiently high number of different solutions on which to apply the local search.

6.1 Experimentation on asymmetric instances

The class that contains 24 instances of AVRPB, obtained from ACVRP instances described by Fischetti et Al in [FTV94], is used for the experimentation of asymmetric instances. For each of ACVRP instance three instances of AVRPB have been created, each corresponding to a percentage of Linehaul respectively 50%, 60% and 80% of customers. The set of customers is partitioned defining, in the list of vertex (generic customer), a Backhaul as the first vertex in every two in the instances with 50% of Linehaul, the first every three customers in the case of 66% and and the first every five customers, respectively, in case of a percentage of Linehaul of 80%.
The customer demand, the capacity of the truck, and the cost matrix are equal to those of the original ACVRP instances. The number of available trucks is determined by $K = max(K_L, K_B)$.
Asymmetric instances have been solved by a number or GRASP iteration equal to 10 multiplied by the number of customers instance (the *size* of the instance).
We propose three variants, in the *costructionphase* (see chapter5). For each variant, the three different percentage between Linehaul-Backhaul are solved, using

different sequences of local search phases, and the Gap indicated is calculated as the average Gap between the lowest objective function value and the BKV.
Results are presented in detail from table 6.1 to table 6.18. For a better performance comparison of the proposed metaheuristic, summary results are presented in table 6.19, related to the objective function Gap, and in table 6.20, related to the average time consumption (expressed in seconds). Both are compared with results obtained by [TV99].
We solve 80% of asymmetric instances, while a value N/A means that the value is not available for the specific instance.

6.1.1 Table 6.1 - 6.18

The following notation is adopted:

- size is referred to the cardinality of the instance
- The first column indicates the instance that has been resolved and the percentage of Linehaul (50%, 66% and 80%)
- BKV is the best known solution, obtained by [TV99]
- sequences of local search phases: {BR: Best Relocate, BE: Best Exchange, FR: First Relocate, FE: First Exchange}
- BrBe: Best Exchange applied to the output of Best Relocate
- BeBr: Best Relocate applied to the output of the Best Exchange
- FrFe: First Exchange applied to the output of First Relocate
- FeFr: First Relocate applied to the output of the First Exchange
- Time (m) indicates the time in minutes to perform the procedure
- Best Gap indicates a percentage, what we deviate from the BKV, computed for the best o. f. value

6.1.2 Table 6.19 - 6.20

In table 6.19, for each variant proposed, it have been reported the average results of the different type of sequences of local search phases implemented, and for every variant and every sequence, the average result is reported in the line *avg*.
The notation is the same of tables 6.1 - 6.18.
The best result is obtained in the 3rd Var., using the BrBe (Best Relocate - Best

Table 6.1: 1^{st} Var., Linehaul = 50 %, restarts = 10 ∗ size: o. f. values

50%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1841	2095	2217	2061	2063	2104	2198	2269	2755	11.95
a036-02f	2112	2219	2470	2216	2278	2271	2384	2371	2813	4.92
a039-02f	2162	2283	2384	2283	2290	2279	2350	2279	2661	5.41
a045-02f	2363	2428	2724	2404	2531	2425	2451	2548	2951	1.74
a048-02f	2352	2641	2935	2583	2586	2583	2655	2744	3531	9.82
a056-02f	2459	2656	2886	2632	2603	2654	2770	2730	3581	5.86
a065-02f	2788	3134	3479	3134	3178	3075	3138	3297	3866	10.29
a071-02f	3012	3384	3785	3219	3393	3243	3555	3443	4535	6.87

Table 6.2: 1^{st} Var., Linehaul = 50 %, restarts = 10 ∗ size: time consumption

50%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	3	4	4	5	2	1	0	0
a036-02f	4	4	6	7	4	4	2	0
a039-02f	7	6	9	11	6	7	3	1
a045-02f	16	13	20	25	13	15	7	2
a048-02f	22	23	29	39	18	15	7	1
a056-02f	46	45	57	69	37	33	17	5
a065-02f	95	84	111	131	75	86	45	22
a071-02f	130	130	144	166	100	83	52	19

Table 6.3: 1^{st} Var., Linehaul = 66 %, restarts = 10 ∗ size: o. f. values

66%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1900	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a036-02f	2190	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a039-02f	2165	2303	2409	2258	2318	2258	2303	2512	2437	4.30
a045-02f	2234	2719	2745	2595	2522	2669	2527	2809	2775	12.89
a048-02f	2458	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	2302	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a065-02f	2678	3436	3469	3266	3206	3287	3240	3410	3699	19.72
a071-02f	2831	3193	3737	3168	3209	3205	3312	3484	4025	11.90

Table 6.4: 1st Var., Linehaul = 66 %, restarts = 10 ∗ size: time consumption

66%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a036-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a039-02f	5	5	9	9	7	6	3	2
a045-02f	11	12	18	19	15	13	7	4
a048-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a065-02f	76	85	111	109	108	86	63	31
a071-02f	139	130	134	137	131	118	53	33

Table 6.5: 1st Var., Linehaul = 80 %, restarts = 10 ∗ size: o. f. values

80%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1704	1793	1852	1791	1767	1783	1889	1785	2133	3.70
a036-02f	2002	2309	2417	2227	2186	2314	2303	2359	2486	9.19
a039-03f	1982	2184	2406	2184	2227	2177	2231	2180	2502	9.84
a045-03f	2184	2385	2726	2363	2413	2352	2400	2383	2685	7.69
a048-02f	2355	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	2328	2858	2929	2727	2625	2762	2677	3091	2938	12.76
a065-03f	2689	2831	3251	2820	2859	2775	2888	2806	3313	3.20
a071-02f	2707	3269	3517	3148	3217	3096	3229	3112	3633	14.37

Table 6.6: 1st Var., Linehaul = 80 %, restarts = 10 ∗ size: time consumption

80%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	5	4	6	7	4	5	2	1
a036-02f	4	4	8	7	7	6	3	2
a039-03f	10	6	12	13	5	7	3	3
a045-03f	23	14	27	30	11	14	6	5
a048-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	41	44	65	61	65	52	38	23
a065-03f	125	87	137	153	57	72	42	37
a071-02f	151	133	134	403	92	95	67	55

Table 6.7: 2^{nd} Var., Linehaul = 50 %, restarts = 10 ∗ size: o. f. values

50%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1841	2149	2204	2113	2088	2005	2193	2066	2592	8.91
a036-02f	2112	2207	2491	2203	2268	2216	2370	2359	2663	4.31
a039-02f	2162	2285	2492	2267	2321	2351	2384	2427	2450	4.86
a045-02f	2363	2432	2799	2432	2571	2460	2497	2580	3090	2.92
a048-02f	2352	2691	2847	2600	2463	2558	2667	2621	3518	4.72
a056-02f	2459	2609	3006	2609	2621	2686	2760	2790	3508	6.10
a065-02f	2788	3147	3451	3098	3106	3144	3197	3281	3908	11.12
a071-02f	3012	3264	3734	3261	3304	3296	3510	3579	4198	8.27

Table 6.8: 2^{nd} Var., Linehaul = 50 %, restarts = 10 ∗ size: time consumption

50%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	6	8	9	13	5	3	1	0
a036-02f	8	10	12	18	8	9	3	1
a039-02f	13	11	16	26	11	14	6	3
a045-02f	27	24	35	46	23	27	11	4
a048-02f	38	41	51	69	32	26	12	3
a056-02f	78	75	99	133	65	60	27	7
a065-02f	163	151	210	237	140	164	77	38
a071-02f	205	218	104	141	183	153	84	28

Table 6.9: 2^{nd} Var., Linehaul = 66 %, restarts = 10 ∗ size: o. f. values

66%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr0	FR	FE	Best Gap %
a034-02f	1900	2035	2194	2005	2040	2053	2038	2087	2397	5.53
a036-02f	2190	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a039-02f	2165	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a045-02f	2234	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a048-02f	2458	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	2302	2413	2710	2383	2343	2341	2348	2435	3024	1.69
a065-02f	2678	3430	3339	3228	3161	3191	3267	3587	3465	18.04
a071-02f	2831	3301	3492	3207	3249	3168	3422	3447	4182	11.90

Table 6.10: 2nd Var., Linehaul = 66 %, restarts = 10 ∗ size: time consumption

66%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	7	9	10	16	6	8	3	1
a036-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a039-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a045-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a048-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	84	73	105	140	72	87	32	19
a065-02f	124	136	189	163	164	161	98	52
a071-02f	93	187	109	126	214	179	79	52

Table 6.11: 2nd Var., Linehaul = 80 %, restarts = 10 ∗ size: o. f. values

80%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1704	1846	1997	1792	1809	1738	1790	1901	2086	2.00
a036-02f	2002	2403	2352	2361	2262	2325	2272	2457	2402	12.99
a039-03f	1982	2175	2414	2164	2162	2208	2202	2244	2544	9.08
a045-03f	2184	2429	2737	2395	2437	2279	2401	2281	2719	4.35
a048-02f	2355	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	2328	2857	2844	2725	2715	2796	2752	3077	3054	16.62
a065-03f	2689	2732	3349	2732	2828	2757	2858	2805	3284	1.60
a071-02f	2707	3221	3747	3143	3172	3130	3235	3296	3735	15.63

Table 6.12: 2nd Var., Linehaul = 80 %, restarts = 10 ∗ size: time consumption

80%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	9	9	12	19	7	9	3	2
a036-02f	7	10	13	16	12	10	5	5
a039-03f	18	11	22	26	10	14	6	5
a045-03f	40	27	47	56	21	26	12	12
a048-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	67	74	113	114	99	91	59	37
a065-03f	213	139	204	107	103	131	84	59
a071-02f	95	88	107	139	83	108	131	126

Table 6.13: 3^{rd} Var., Linehaul = 50 %, restarts = 10 $*$ size: o. f. values

50%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1841	1977	2281	1977	2110	2084	2189	2270	2728	7.39
a036-02f	2112	2264	2388	2259	2253	2275	2361	2330	2669	6.68
a039-02f	2162	2350	2523	2339	2311	2315	2296	2358	2660	6.89
a045-02f	2363	2545	2832	2494	2465	2476	2502	2490	3014	4.32
a048-02f	2352	2658	2694	2608	2528	2495	2699	2636	3366	6.08
a056-02f	2459	2645	2923	2645	2700	2623	2734	2814	3410	6.67
a065-02f	2788	3090	3368	3088	3103	3122	3129	3225	3686	10.76
a071-02f	3012	3247	3613	3227	3396	3290	3435	3483	4105	7.14

Table 6.14: 3^{rd} Var., Linehaul = 50 %, restarts = 10 $*$ size: time consumption

50%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	2	2	3	3	2	1	0	0
a036-02f	3	3	4	5	3	3	1	0
a039-02f	5	4	6	8	4	5	2	1
a045-02f	10	8	13	17	8	10	4	1
a048-02f	13	14	17	24	11	9	4	1
a056-02f	28	29	37	46	25	22	11	3
a065-02f	49	48	59	74	44	50	27	14
a071-02f	68	67	84	107	57	49	29	10

Table 6.15: 3^{rd} Var., Linehaul = 66 %, restarts = 10 $*$ size: o. f. values

66%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1900	2053	2226	2026	1987	1966	2058	2033	2307	3.47
a036-02f	2190	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a039-02f	2165	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a045-02f	2234	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a048-02f	2458	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	2302	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a065-02f	2678	3329	3423	3129	3115	3149	2978	3437	3550	11.20
a071-02f	2831	3037	3690	3037	3267	3083	3202	3282	4018	7.28

Table 6.16: 3rd Var., Linehaul = 66 %, restarts = 10 ∗ size: time consumption

66%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	2	2	3	5	2	3	1	0
a036-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a039-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a045-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a048-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a065-02f	35	42	54	60	53	49	33	23
a071-02f	76	66	85	110	78	74	30	21

Table 6.17: 3rd Var., Linehaul = 80 %, restarts = 10 ∗ size: o. f. values

80%	BKV	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE	Best Gap %
a034-02f	1704	1769	2002	1761	1831	1776	1761	1811	2052	3.35
a036-02f	2002	2424	2332	2285	2171	2355	2284	2471	2504	8.44
a039-03f	1982	2156	2473	2156	2178	2159	2177	2205	2479	8.78
a045-03f	2184	2366	2691	2366	2397	2338	2373	2338	2851	8.33
a048-02f	2355	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	2328	2744	2894	2510	2500	2580	2751	2963	2926	7.39
a065-03f	2689	2787	3216	2773	2834	2723	2745	2800	3292	1.26
a071-02f	2707	3075	3624	3075	3178	3183	3182	3224	3598	13.59

Table 6.18: 3rd Var., Linehaul = 80 %, restarts = 10 ∗ size: time consumption

80%	BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
a034-02f	3	2	4	5	2	3	1	0
a036-02f	2	3	4	4	4	4	1	1
a039-03f	6	4	8	9	4	5	2	1
a045-03f	13	8	16	19	7	9	4	3
a048-02f	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
a056-02f	22	26	32	31	36	33	16	15
a065-03f	60	40	69	85	31	38	24	19
a071-02f	74	68	89	133	68	87	41	49

Table 6.19: Average Gap for all GRASP variants and all sequences of local search phases in respect to BKV for asymmetric instances

GRASP variant	% of Linehaul	o. f. Gap (%) BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
1st Var.	50 %	9,03	19.44	7.54	9.42	8.20	12.60	13.63	39.47
	66 %	17.29	23.92	13.58	13.26	14.93	14.37	23.04	29.27
	80 %	12.68	21.75	10.43	10.52	10.57	12.86	13.40	26.02
avg		13.00	21.70	10.52	11.07	11.23	13.28	16.69	31.59
2nd Var.	50 %	8.95	20.30	7.86	8.67	8.36	13.01	13.35	35.34
	66 %	14.15	20.31	10.72	10.49	10.20	13.03	17.83	33.66
	80 %	13.23	23.85	10.96	11.26	10.31	11.94	15.74	26.65
avg		12.11	21.49	9.84	10.14	9.62	12.66	15.64	31.88
3rd Var.	50 %	8.77	18.46	8.05	9.23	8.34	11.87	13.24	34.43
	66 %	13.21	25.11	10.25	12.10	9.99	10.87	17.09	31.97
	80 %	11.02	22.82	8.45	9.39	9.64	10.53	14.07	26.02
avg		11.00	22.13	8.92	10.24	9.32	11.09	14.80	30.81

Exchange) sequence of local search phase. In this case, an average Gap of 8.92% is achieved.
From the time consumption point of view, the critical phase is the local search. The most promising strategy is the First Exchange, that in the 3rd Var. obtain an average time consumption of 8.32 minutes (499.2 s), but it is one of the worst in accuracy. The combo strategy of Best Relocate follow to a Best Exchange (BrBe) is the most accurate, from the objective function value point of view, but it obtain, in the 3rd Var., an average time of 30.83 minutes (1849.8 s), which is one of the worst, best only of BeBr.

6.2 Experimentation on symmetric instances

In order to test symmetric problems, the class of problems that consists of 62 Euclidean VRPB randomly instances generated, proposed by Goetshalckx and Jacobs-Blecha in [GJB89] is used. The customer coordinates are uniformly distributed in the interval [0, 24000] to the x values and the interval [0, 32000] for the y values. The depot is located centrally at coordinates (12000, 16000).
The cost of the link $c_{i,j} \in A$ (see the mathematical model, paragraph 2.4) is defined as the Euclidean distance between i and j customers.

Table 6.20: Time comparison - GRASP variants for asymmetric instances

GRASP variant	% of Linehaul	time consumption for each instance (min) BR	BE	BrBe	BeBr	FrFe	FeFr	FR	FE
1st var.	50	40.38	38.63	47.50	56.63	31.88	30.50	16.63	6.25
	66	62.75	50.13	73.13	70.25	47.00	55.25	40.00	33.00
	80	48.75	39.50	53.38	90.00	32.88	35.00	21.63	16.88
avg		50.63	42.75	58.00	72.29	37.25	40.25	26.08	18.71
2nd var.	50	67.25	67.25	67.00	85.38	58.38	57.00	27.63	10.50
	66	60.00	75.50	80.17	89.67	83.83	81.67	38.33	22.67
	80	62.75	50.13	73.13	70.25	47.00	55.25	40.00	33.00
avg		63.33	64.29	73.43	81.76	63.07	64.64	35.32	22.06
3rd var.	50	22.25	21.88	27.88	35.50	19.25	18.63	9.75	3.75
	66	27.17	25.50	34.00	41.83	29.67	30.33	14.17	9.33
	80	24.75	20.63	30.63	39.50	20.88	24.75	12.00	11.88
avg		24.72	22.67	30.83	38.94	23.26	24.57	11.97	8.32

In table 6.21 are reported the size of the solved instances, that correspond to the sum of n Linehaul and m Backhaul customers.
Customers requests are generated from a normal distribution with an arithmetic mean $\mu = 500$ and standard deviation $\sigma = 200$. Fourteen values for the total number of vertex, $n + m$ (whose total vary between 25 to 150), with a percentage of Linehaul equal to 50%, 66%, and 80%. For each value of $n + m$, the capacity C of the vehicle is defined so that about a number of vehicles $k \in [3, 12], k \in \mathbb{N}$, are used to serve all the requests.
Time consumption, in symmetric instances, is indicated in milliseconds (ms).
The number of restarts of the GRASP metaheuristic ($max\_iteration$) is linked to the size of the instance, and it vary with the law:

$$max\_iteration = 15 * (n + m)$$

Results on symmetrical instances are presented in detail from table 6.22 to table 6.49. Summary results are presented in table 6.50, concerning to the objective function Gap, and in table 6.51 concerning to the average time consumption (expressed in ms).
To better perform a comparison between the approach presented and the results from the literature, for each instance the Best Known Value (BKV) and its related Gap are reported in results. The reference BKV used are those obtained from [ZK12]. Both objective function value and the request CPU time are compared. Notice that the authors of [ZK12], regarding the termination condition used for

Table 6.21: Symmetric instances - names and size (nodes)

instance	size	instance	size	instance	size	instance	size
A1.txt	25	E1.txt	45	H1.txt	68	K1.txt	113
A2.txt	25	E2.txt	45	H2.txt	68	K2.txt	113
A3.txt	25	E3.txt	45	H3.txt	68	K3.txt	113
A4.txt	25	F1.txt	60	H4.txt	68	K4.txt	113
B1.txt	30	F2.txt	60	H5.txt	68	L1.txt	150
B2.txt	30	F3.txt	60	H6.txt	68	L2.txt	150
B3.txt	30	F4.txt	60	I1.txt	90	L3.txt	150
C1.txt	40	G1.txt	57	I2.txt	90	L4.txt	150
C2.txt	40	G2.txt	57	I3.txt	90	L5.txt	150
C3.txt	40	G3.txt	57	I4.txt	90	M1.txt	125
C4.txt	40	G4.txt	57	I5.txt	90	M2.txt	125
D1.txt	38	G5.txt	57	J1.txt	94	M3.txt	125
D2.txt	38	G6.txt	57	J2.txt	94	M4.txt	125
D3.txt	38			J3.txt	94	N1.txt	150
D4.txt	38			J4.txt	94	N2.txt	150
						N3.txt	150
						N4.txt	150

a single execution, sets to the completion of 120 CPU seconds for problems with $(n+m) \leq 50$, and 300 CPU seconds for instances involving up to 50 vertices.

6.2.1 Table 6.22 - 6.49

The following notation is adopted:

- instance is the solved instance
- BKV is the best known solution, obtained by [ZK12]
- first version, without preprocessing: the values of the objective function and the related Gap (%), and the time consumption. The local search strategy is free, choosing the best result find for every instance solved
- second version, with preprocessing: the values of the objective function and the related Gap (%), and the time consumption; in this case, we force two local search sequences: FeFr: First Relocate applied to the output of the First Exchange, and BeBr: Best Relocate applied to the output of the Best Exchange

Table 6.22: Class A instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
A1.txt	229886	229886	0	229886	0	230547	0.28
A2.txt	180119	180119	0	180119	0	180119	0
A3.txt	163405	163642	0.14	163405	0	163405	0
A4.txt	155796	155796	0	156033	0.15	156033	0.15

Table 6.23: Class A Instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
A1.txt	229963	0	46391	0	15713	0.28
A2.txt	274072	0	39024	0	14641	0
A3.txt	311584	0.14	33055	0	14782	0
A4.txt	295864	0	31679	0.15	12760	0.15

- o. f. is the value of the objective function
- Gap % is the Gap with BKV
- time (ms) is the CPU time required to solve the instance

6.2.2 Table 6.50 - 6.51

In table 6.50, the average values of o.f. results re presented. The second version with a sequence of local search of Best Relocate follow a Best Exchange (BeBr), score a Gap of 5.79%, while the other sequence proposed, the Best Relocate follow to a Best Exchange (BrBe), obtain the worst Gap.
From the time consumption point of view, the proposed metaheuristic obtain a good result for instances with size ≤ 50 (see 6.2). In fact, the second version with a sequence of local search of Best Relocate follow a Best Exchange (BrBe) give a performance less than -57% in respect to the best known. This promising result worsens along with the increase in the size of the instance, due to the fact that the

Table 6.24: Class B instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
B1.txt	239080	239080	0	239080	0	239152	0.03
B2.txt	198048	198048	0	198048	0	198048	0
B3.txt	169372	169372	0	169372	0	169372	0

Table 6.25: Class B Instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
B1.txt	402444	0	70884	0	23148	0.03
B2.txt	533486	0	70205	0	28819	0
B3.txt	575029	0	53657	0	25335	0

Table 6.26: Class C instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
C1.txt	250556	251665	0.44	254477	1.56	256694	2.45
C2.txt	215020	220941	2.75	219587	2.12	219609	2.13
C3.txt	199346	199346	0	203396	2.03	201164	0.91
C4.txt	195366	195367	0	200179	2.46	199975	2.35

Table 6.27: Class C instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
C1.txt	1412641	0.44	153440	1.56	66581	2.45
C2.txt	1600588	2.75	137964	2.12	64650	2.13
C3.txt	1926792	0	117221	2.03	65587	0.91
C4.txt	1902683	0	118679	2.46	63175	2.35

Table 6.28: Class D instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
D1.txt	322530	322530	0	322740	0.06	324169	0.50
D2.txt	316708	318431	0.54	316709	0	318327	0.51
D3.txt	239479	240122	0.26	242074	1.08	239934	0.19
D4.txt	205832	207710	0.91	208411	1.25	212420	3.20

Table 6.29: Class D instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
D1.txt	1767024	0	168201	0.06	63376	0.50
D2.txt	1456836	0.54	163405	0	57877	0.51
D3.txt	1651440	0.26	142557	1.08	58132	0.19
D4.txt	1883220	0.91	119846	1.25	56179	3.20

Table 6.30: Class E instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
E1.txt	238880	239756	0.36	243490	1.93	245334	2.70
E2.txt	212263	213139	0.41	216592	2.03	212376	0.05
E3.txt	206659	209713	1.47	219769	6.34	210332	1.77

Table 6.31: Class E instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
E1.txt	3100368	0.36	227704	1.93	109417	2.70
E2.txt	3243705	0.41	187720	2.03	98608	0.05
E3.txt	3443746	1.47	184124	6.34	99478	1.77

Table 6.32: Class F instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
F1.txt	263173	278467	5.81	278308	5.75	276955	5.23
F2.txt	265213	265654	0.16	268747	1.33	274205	3.39
F3.txt	241120	252069	4.54	261528	8.46	248583	3.09
F4.txt	233861	246290	5.31	250750	7.22	244775	4.66

Table 6.33: Class F instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
F1.txt	8925326	5.81	291379	5.75	251453	5.23
F2.txt	9460159	0.16	295700	1.33	270500	3.39
F3.txt	10811699	4.54	250216	8.46	253253	3.09
F4.txt	10808895	5.31	241038	7.22	252832	4.66

Table 6.34: Class G instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
G1.txt	306306	311656	1.74	320774	4.72	318834	4.09
G2.txt	245441	251883	2.62	245797	0.14	254230	3.58
G3.txt	229507	233340	1.67	237916	3.66	234289	2.08
G4.txt	232521	239918	3.18	249935	7.48	243065	4.53
G5.txt	221730	232867	5.02	234243	5.64	228815	3.19
G6.txt	213457	225584	5.68	230180	7.83	220220	3.16

Table 6.35: Class G instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
G1.txt	9962271	1.74	312073	4.72	233332	4.09
G2.txt	9844463	2.62	245420	0.14	213118	3.58
G3.txt	10678444	1.67	220377	3.66	207890	2.08
G4.txt	10811734	3.18	232184	7.48	225885	4.53
G5.txt	10804967	5.02	208948	5.64	214016	3.19
G6.txt	10812317	5.68	211037	7.83	202012	3.16

Table 6.36: Class H instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
H1.txt	268933	280643	4.35	282393	5	282196	4.93
H2.txt	253365	265800	4.90	269911	6.53	265310	4.71
H3.txt	247449	267337	8.03	271547	9.73	263318	6.41
H4.txt	250221	265296	6.02	263882	5.46	261202	4.38
H5.txt	246121	267859	8.83	272212	10.60	258976	5.22
H6.txt	249135	265399	6.52	271486	8.97	263943	5.94

Table 6.37: Class H instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
H1.txt	10816072	4.35	425785	5	376634	4.93
H2.txt	10802912	4.90	411953	6.53	368306	4.71
H3.txt	10811470	8.03	386020	9.73	360284	6.41
H4.txt	10809833	6.02	397714	5.46	376910	4.38
H5.txt	10807211	8.83	392120	10.60	370601	5.22
H6.txt	10809947	6.52	387392	8.97	380201	5.94

Table 6.38: Class I instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
I1.txt	350246	369286	5.43	373723	6.70	375738	7.27
I2.txt	309944	326055	5.19	350493	13.08	343550	10.84
I3.txt	294507	324183	10.07	326582	10.89	324421	10.15
I4.txt	295988	314540	6.26	336296	13.61	321907	8.75
I5.txt	301236	318322	5.67	330599	9.74	317924	5.54

Table 6.39: Class I instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
I1.txt	10826415	5.43	1647814	6.70	1013776	7.27
I2.txt	10822947	5.19	1450553	13.08	938782	10.84
I3.txt	10839929	10.07	1260896	10.89	919495	10.15
I4.txt	10812359	6.26	1293910	13.61	999395	8.75
I5.txt	10826445	5.67	1354541	9.74	1043845	5.54

Table 6.40: Class J instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
J1.txt	335006	352801	5.31	364295	8.74	358919	7.13
J2.txt	310417	343975	10.81	341786	10.10	329208	6.05
J3.txt	279219	307191	10.01	316174	13.23	300840	7.74
J4.txt	296533	317177	6.96	325587	9.79	315258	6.31

Table 6.41: Class J instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
J1.txt	10837724	5.31	1837470	8.74	1038197	7.13
J2.txt	10822553	10.81	1718190	10.10	1013428	6.05
J3.txt	10803924	10.01	1581680	13.23	1013338	7.74
J4.txt	10814367	6.96	17301.05	9.79	1017676	6.31

Table 6.42: Class K instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
K1.txt	394071	427179	8.40	430964	9.36	423374	7.43
K2.txt	362130	403350	11.38	404155	11.60	393960	8.79
K3.txt	365694	411016	12.39	405485	10.88	391312	7.00
K4.txt	348950	389552	11.63	396134	13.52	377429	8.16

Table 6.43: Class K instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
K1.txt	10825798	8.40	3525764	9.36	1901732	7.43
K2.txt	10863111	11.38	3273691	11.60	1900787	8.79
K3.txt	10855352	12.39	3375189	10.88	1977590	7.00
K4.txt	10854125	11.63	3353628	13.52	1994505	8.16

Table 6.44: Class L instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
L1.txt	417896	509328	21.87	513273	22.82	502715	20.29
L2.txt	401228	455992	13.64	482345	20.21	462025	15.15
L3.txt	402678	459041	13.99	471290	17.03	456409	13.34
L4.txt	384636	444263	15.50	464329	20.71	433024	12.58
L5.txt	387565	454654	17.31	467954	20.74	440572	13.67

Table 6.45: Class L instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
L1.txt	10812987	21.87	3600623	22.82	3600545	20.29
L2.txt	10904618	13.64	3603643	20.21	3601094	15.15
L3.txt	11159460	13.99	3604219	17.03	3601536	13.34
L4.txt	10977078	15.50	3602095	20.71	3600042	12.58
L5.txt	11014096	17.31	3602039	20.74	3600530	13.67

Table 6.46: Class M instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
M1.txt	398593	439846	10.35	444242	11.45	419208	5.17
M2.txt	396917	450343	13.46	456418	14.99	439169	10.64
M3.txt	375696	401994	6.99	416000	10.72	407542	8.47
M4.txt	348140	389300	11.82	393610	13.06	376677	8.19

Table 6.47: Class M instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
M1.txt	10861022	10.35	3601527	11.45	2523681	5.17
M2.txt	10830164	13.46	3602309	14.99	2343494	10.64
M3.txt	10820885	6.99	3601606	10.72	2513456	8.47
M4.txt	10825775	11.82	3600788	13.06	2525238	8.19

Table 6.48: Class N instances: o. f. values

		first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	BKV	o. f.	Gap (%)	o. f.	Gap (%)	o. f.	Gap (%)
N1.txt	408101	467525	14.56	478759	17.31	446672	9.45
N2.txt	408066	471885	15.63	478949	17.37	435498	6.72
N3.txt	394338	457466	16	465787	18.11	425676	7.94
N4.txt	394788	463324	17.36	447772	13.42	N/A	

Table 6.49: Class N instances: time consumption

	first version		second version			
			LS Opt.: Fe Fr		LS Opt.: Be Br	
instance	time (ms)	Gap (%)	time (ms)	Gap (%)	time (ms)	Gap (%)
N1.txt	11028568	14.56	3604835	17.31	3601242	9.45
N2.txt	10805727	15.63	3604012	17.37	3601319	6.72
N3.txt	10816953	16	3602007	18.11	3602163	7.94
N4.txt	11372768	17.36	3600882	13.42	N/A	

Table 6.50: o.f. Gap comparison - GRASP Versions on symmetric instances

first version	second version	
	LS Opt.: Fe Fr	LS Opt.: Be Br
(%)	(%)	(%)
6.78	8.55	5.79

Table 6.51: Time comparison - GRASP Versions on symmetric instances

	Time BKV	first version		second version			
				LS Opt.: Fe Fr		LS Opt.: Be Br	
instances	(ms)	(ms)	(%)	(ms)	(%)	(ms)	(%)
From A1 to E3	120000	1445082,5	104	114764,22	-4	52125,44	-57
From F1 to N4	300000	10733639,76	3478	1888128,46	529	1464490,56	388

time limit we imposed is 3600 seconds, far greater than that of comparison.

Chapter 7

Conclusion

7.1 Summary of work

In this thesis we propose a GRASP for the VRPB. In the construction phase, it determines a number of Linehaul-Backhaul pairs which is equal to the number of routes required in VRPB instances. These pairs are promising because they are likely to be found in high quality VRPB routes. Next, two open routes are created from each node of the pair to the depot. At each step, the next node to be included in the open route is randomly selected from a RCL, which is a list of nodes that can be visited after the current one. Finally, these routes are joined to create a feasible route for the VRPB. These steps are repeated for all pairs, until a feasible VRPB solution is determined.
Three different strategies (variants) are proposed for the RCL construction. In the first variant, all nodes can be used as candidate for the RCL, and at each of them a probability proportional to its distance of the current node is assigned; in the second variant, a fixed percentage of probability limit the number of node that can be used in the candidate list; in the third variant, a fixed number of nodes is used, regardless of the probability. The first variant is a pure random strategy, the third is greedy strategy, while the second is a mixed approach.
The second phase, called local search phase, improves the construction phase solution, using several local search sequences. These sequences are based on node relocate and node exchange moves with first-improvement (Fe) and best improvement (Be).

7.2 Future research directions

The GRASP generates an high number of solutions, each of which may separately contain high quality routes. Therefore, it is of interest to store in memory all solutions and select by a set partitioning formulation the subset of routes with minimum solution cost. In a very recent experimentation, we solve all instances in [GJB89], obtaining promising results. More precisely, from Class A1 to Class E2 we always find BKVs; from instance E3 to instance H6, the Gap is 0.86% with 3 BKVs found; from instance I1 to instance K4 the Gap is 1%, whereas it is about 2% for instances from L1 to N6. A tuning phase of the algorithm is currently in progress in order to find the best setting of parameters. Future work will be done adding the possibility of accepting infeasible solutions during the search process, in order to visit new promising areas of the solutions space which cannot be reached by standard local search methods. Another interesting development could address the insertion of some ideas on granularity to speed-up the local search phase.

Bibliography

[ABY97] Michael F Argüello, Jonathan F Bard, and Gang Yu. A GRASP for aircraft routing in response to groundings and delays. *Journal of Combinatorial Optimization*, 1(3):211–228, 1997.

[APC11] E Arráiz and AD Palhazi Cuervo. An iterated local search algorithm for the vehicle routing problem with backhauls. In *Proceedings of the 9th Metaheuristic International Conference (MIC)*, pages 339–343, 2011.

[ARR02] Renata M Aiex, Mauricio GC Resende, and Celso C Ribeiro. Probability distribution of solution time in grasp: An experimental investigation. *Journal of Heuristics*, 8(3):343–373, 2002.

[Atk98] JB Atkinson. A greedy randomised search heuristic for time-constrained vehicle scheduling and the incorporation of a learning strategy. *Journal of the Operational Research Society*, 49(7):700–708, 1998.

[Bar97] Jonathan F Bard. An analysis of a rail car unloading area for a consumer products manufacturer. *Journal of the Operational Research Society*, 48(9):873–883, 1997.

[BLP07] Mourad Boudia, Mohamed Aly Ould Louly, and Christian Prins. A reactive GRASP and path relinking for a combined production–distribution problem. *Computers & Operations Research*, 34(11):3402–3419, 2007.

[Bra06] José Brandao. A new tabu search algorithm for the vehicle routing problem with backhauls. *European Journal of Operational Research*, 173(2):540–555, 2006.

[BRRC02] MG Baldoquín, G Ryan, R Rodriguez, and A Castellini. Un enfoque hibrido basado en metaheurísticas para el problema del cartero rural. *Proceedings of XI CLAIO, Concepción de Chile, Chile*, 2002.

[CB02] Carlos Carreto and Barrie Baker. A GRASP interactive approach to the vehicle routing problem with backhauls. In *Essays and Surveys in Metaheuristics*, pages 185–199. Springer, 2002.

[CCCM86] Nicos Christofides, V Campos, A Corberán, and E Mota. An algorithm for the rural postman problem on a directed graph. In *Netflow at Pisa*, pages 155–166. Springer, 1986.

[CMS02] Angel Corberán, R Martı, and José M Sanchis. A GRASP heuristic for the mixed chinese postman problem. *European Journal of Operational Research*, 142(1):70–80, 2002.

[CS03] Robert H Currie and Said Salhi. Exact and heuristic methods for a full-load, multi-terminal, vehicle scheduling problem with backhauling and time windows. *Journal of the Operational Research Society*, 54(4):390–400, 2003.

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

[dCRS98] P Fernández de Córdoba, LM Garcıa Raffi, and JM Sanchis. A heuristic algorithm based on monte carlo methods for the rural postman problem. *Computers & operations research*, 25(12):1097–1106, 1998.

[dlP04] María Gulnara Baldoquín de la Peña. Heuristics and metaheuristics approaches used to solve the rural postman problem: A comparative case study. In *Proceedings of the Fourth International ICSC Symposium on Engineering of Intelligent Systems (EIS 2004)*. Citeseer, 2004.

[DR59] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[FB89] Thomas A Feo and Jonathan F Bard. Flight scheduling and maintenance base planning. *Management Science*, 35(12):1415–1432, 1989.

[FGV95] Thomas A Feo and Jose Luis Gonzalez-Velarde. The intermodal trailer assignment problem. *Transportation Science*, 29(4):330–341, 1995.

[FTV94] Matteo Fischetti, Paolo Toth, and Daniele Vigo. A branch-and-bound algorithm for the capacitated vehicle routing problem on directed graphs. *Operations Research*, 42(5):846–859, 1994.

[GA09] Yuvraj Gajpal and Prakash L Abad. Multi-ant colony system (macs) for a vehicle routing problem with backhauls. *European Journal of Operational Research*, 196(1):102–117, 2009.

[GJB89] Marc Goetschalckx and Charlotte Jacobs-Blecha. The vehicle routing problem with backhauls. *European Journal of Operational Research*, 42(1):39–51, 1989.

[GO06] Hassan Ghaziri and Ibrahim H Osman. Self-organizing feature maps for the vehicle routing problem with backhauls. *Journal of scheduling*, 9(2):97–114, 2006.

[Hjo95] Curt A Hjorring. The vehicle routing problem and local search metaheuristics. 1995.

[HM01] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European journal of operational research*, 130(3):449–467, 2001.

[HS87] J Pirie Hart and Andrew W Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6(3):107–114, 1987.

[JPY88] David S Johnson, Christos H Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of computer and system sciences*, 37(1):79–100, 1988.

[KB95] George Kontoravdis and Jonathan F Bard. A GRASP for the vehicle routing problem with time windows. *ORSA journal on Computing*, 7(1):10–23, 1995.

[LGWL04] Zhiye Li, Songshan Guo, Fan Wang, and Andrew Lim. Improved GRASP with tabu search for vehicle routing with both time window and limited number of vehicles. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 552–561. Springer, 2004.

[LK73] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[MGB99] Aristide Mingozzi, Simone Giorgi, and Roberto Baldacci. An exact method for the vehicle routing problem with backhauls. *Transportation Science*, 33(3):315–329, 1999.

[OB16] Jonathan Oesterle and Thomas Bauernhansl. Exact method for the vehicle routing problem with mixed linehaul and backhaul customers, heterogeneous fleet, time window and manufacturing capacity. *Procedia CIRP*, 41:573–578, 2016.

[OL96] Ibrahim H Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[OW02] Ibrahim H Osman and Niaz A Wassan. A reactive tabu search metaheuristic for the vehicle routing problem with back-hauls. *Journal of Scheduling*, 5(4):263–285, 2002.

[PR00] Marcelo Prais and Celso C Ribeiro. Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment. *INFORMS Journal on Computing*, 12(3):164–176, 2000.

[RP06a] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

[RP06b] Stefan Ropke and David Pisinger. A unified heuristic for a large class of vehicle routing problems with backhauls. *European Journal of Operational Research*, 171(3):750–775, 2006.

[RR99] LIP Resende and MGC Resende. A GRASP for frame relay PVC routing. 1999.

[RR03] Mauricio GC Resende and Celso C Ribeiro. A GRASP with path-relinking for private virtual circuit routing. *Networks*, 41(2):104–114, 2003.

[RR05] Mauricio GC Resendel and Celso C Ribeiro. Grasp with path-relinking: Recent advances and applications. In *Metaheuristics: progress as real problem solvers*, pages 29–63. Springer, 2005.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 417–431. Springer, 1998.

[Sol87] Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[Sos00] Danuta Sosnowska. Optimization of a simplified fleet assignment problem with metaheuristics: Simulated annealing and GRASP. In *Approximation and Complexity in Numerical Optimization*, pages 477–488. Springer, 2000.

[SW00] Petra Schuurman and Gerhard J Woeginger. Approximation schemes-a tutorial. In *Lectures on scheduling*. Citeseer, 2000.

[TMSZ06] Reza Tavakkoli-Moghaddam, AR Saremi, and MS Ziaee. A memetic algorithm for a vehicle routing problem with backhauls. *Applied Mathematics and Computation*, 181(2):1049–1060, 2006.

[TPS96] Sam R Thangiah, Jean-Yves Potvin, and Tong Sun. Heuristic approaches to vehicle routing with backhauls and time windows. *Computers & Operations Research*, 23(11):1043–1057, 1996.

[TV96] Paolo Toth and Daniele Vigo. A heuristic algorithm for the vehicle routing problem with backhauls. In *Advanced methods in transportation analysis*, pages 585–608. Springer, 1996.

[TV97] Paolo Toth and Daniele Vigo. An exact algorithm for the vehicle routing problem with backhauls. *Transportation science*, 31(4):372–385, 1997.

[TV99] Paolo Toth and Daniele Vigo. A heuristic algorithm for the symmetric and asymmetric vehicle routing problems with backhauls. *European Journal of Operational Research*, 113(3):528–543, 1999.

[VCGP14] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research*, 234(3):658–673, 2014.

[VMOR12] Stefan Voß, Silvano Martello, Ibrahim H Osman, and Catherine Roucairol. *Meta-heuristics: Advances and trends in local search paradigms for optimization*. Springer Science & Business Media, 2012.

[Was07] N Wassan. Reactive tabu adaptive memory programming search for the vehicle routing problem with backhauls. *Journal of the Operational Research Society*, 58(12):1630–1641, 2007.

[WN99] Laurence A Wolsey and George L Nemhauser. Integer and combinatorial optimization. *John Willey And Sons*, 1999.

[ZK10] Emmanouil E Zachariadis and Chris T Kiranoudis. A strategy for reducing the computational complexity of local search-based methods for the vehicle routing problem. *Computers & Operations Research*, 37(12):2089–2105, 2010.

[ZK12] Emmanouil E Zachariadis and Chris T Kiranoudis. An effective local search approach for the vehicle routing problem with backhauls. *Expert Systems with Applications*, 39(3):3174–3184, 2012.