Corso di Laurea Magistrale in
INGEGNERIA GESTIONALE
Classe delle Lauree LM-31

Tesi di Laurea Magistrale

Branch-and-Price and Heuristic Algorithms for the Service Network Design and Hub Location Problem

Relatori:

Prof. Rosario SCATAMACCHIA

Prof. Dr. Marco LÜBBECKE

Alexander HELBER, M.Sc.

Candidato:

Alessio BUFANO

Anno Accademico 2021/2022

Abstract

The design of freight transport networks is becoming an even more relevant component in the context of the worldwide increasing popularity of e-commerce and increasing export volumes. The focus of this thesis is on a combined transport problem where multiple itineraries are possible for commodities with the same origin and destination locations. The problem targets both the strategic positioning of transshipment warehouses – the so-called hubs – and the tactical planning of freight transport. The aim is to achieve the best trade-off between operational costs and service performance. We consider, among others, important real-world conditions on the routing of goods: modular capacities on transfer links between hubs, maximum delivery times of goods, and limits on the number of transshipments. Overall, the whole combination of these problem characteristics has never been treated in the previous literature. For the considered problem, we propose two mathematical formulations and a Branch-and-Price algorithm. Besides, we introduce various heuristic approaches to obtain good-quality solutions with limited computational time. Extensive computational experiments show the effectiveness of the proposed algorithms in solving realistic instances, enabling strategic network design in real-world applications.

Keywords: Combined Freight Transport, Service Network Design, Hub Location, Mixed Integer Linear Programming, Branch-and-Price, Heuristic

Acknowledgements

First of all, I would like to thank the Polytechnic of Turin and the RWTH Aachen University for the special opportunity they gave me to write this thesis. After all, we did a good work, even though I lived three very intense and stressful months and I met a lot of difficulties. But maybe it is proper this, that made me proud.

Thank you, Marco, for having accepted me as abroad student and having always suggested the right things.
Thank you, Alex, for having always been there present to help me in the most difficult moments, redirecting me on the right way.
Thank you, Judith, for having been a solar person to talk with. I really think the Operations Research Chair would not be the same without you.
Thank you, Ben, for having helped me when the cluster had problems.

Finally, thank you, Rosario, for having always believed in me, and having supported me in the final crucial moments.

Directly from my heart
∼ *Alessio*

Table of Contents

List of Tables

List of Figures

Acronyms

B&B
: Branch-and-Bound

B&P
: Branch-and-Price

B&P&C
: Branch-and-Price-and-Cut

CG
: Column Generation

DB
: Dual Bound

DW
: Dantzig-Wolfe

HLP
: Hub Location Problem

ILP
: Integer Linear Programming

LB
: Lower Bound

LP
: Linear Programming

MILP

Mixed Integer Linear Programming

MP

Master Problem

PB

Primal Bound

PP

Pricing Problem

RMP

Restricted Master Problem

SNDHLP

Service Network Design and Hub Location Problem

SNDP

Service Network Design Problem

T.L.

Time Limit

UB

Upper Bound

Chapter 1

Introduction

The goal of this thesis is to deepen a real-world topic of raising interest over the last years. We actually live in a world where we need only to type some words and click a button to order what we desire or effectively necessitate: the online commerce is in a stable and unceasing expansion. Moreover, the constant growth of the world population and the increasing welfare are strictly related with a greater demand of food, goods and services. These lead to broadening importation, and clearly exportation. Every day, all over the world, roads, rails, air and maritime routes are plenty of trucks, vans, trains, aircraft, cargo ships, and many other vehicles that transport commodities from a location to another.
In this context, it is fundamental to design the freight transport network and to eventually choose a good location for warehouses where goods can be transshipped. The positive outcomes of this planning are diverse:

- To speed up the delivery process.
- To boost firms' performance, by means of efficient use of resources and high service levels.
- To improve customers' satisfaction levels.
- To avoid additional storing-time costs or economic penalties for shipping companies.
- To reduce the environmental pollution.
- To prevent freight damages.

In particular, a very sensitive part of the whole process is represented by the transshipment at collection and sorting points – the so-called hubs. But this is what distinguishes direct transport from combined one. Indeed, whereas in direct transport – as the name itself suggests – the demand travels directly from an origin to a destination, in the latter the main part of the transportation is performed by the transfer vehicles operating an internal network of hubs, and only the initial and the final legs of the trip are carried out by the vehicles on the access links to/from the transshipment network. Actually, the operations management is a complicated process, which is more crucial when the distances among the locations are larger.

Hence, the object of the study is a real-world combined transport Service Network Design and Hub Location Problem (which is denoted as SNDHLP) that simultaneously deals with the planning of demand units itineraries and the strategic locations of the hub facilities. The problem must serve a given set of transport requests. Each request consists of a certain number of demand units to be transported from the customers' origins to the customers' destinations. Each customer location may be the origin and/or the destination of more than one request, but all the commodities with the same origin and the same destination constitute one request. The problem is constrained to many real-world conditions:

- In a combined transport, the requests' commodities must be transshipped at hub facilities, which means that the direct transport from an origin to a destination is forbidden.

- Requests are splittable, or, in other words, the demand units with the same origin and destination locations are not obliged to follow the same routing itinerary.

- There is a fixed limited number of hubs that must be opened, without any cost.

- Each customer location can be served by multiple hubs: these constitute the set of allowed hubs of the customer. The related links between customers and allowed hubs – or vice versa – are called access arcs, and have a cost for kilometer and demand unit.

- The first hub in which the request's demand units are transshipped must be an allowed start-hub for the origin of any given customer and similarly the last hub must be an allowed end-hub for the destination of any given customer.

- The links between hubs are called transfer arcs, and have a modular capacity – which means that the capacity of each link is equal to a multiple of a given module (see Pióro and Medhi (2004)). This modular capacity need to be provided by operating integer amounts of identical vehicles having a specific transportation capacity – the given module. Hence, these links have a cost for kilometer and vehicle used.

- There is a limited number of transshipments at hubs – the so-called hops – for a single request.
- Every request must be delivered within a maximum travel time, including also the transshipments.

In conclusion, the problem is to decide the itineraries of the requests, the selection of hubs to be opened, and the number of vehicles operating on each transfer arc. The objective is to minimize the overall delivery costs of requests, which include both the costs for operating vehicles among the hubs' connections and the costs for using the access arcs.

Our practical contribution to the studied problem is the presentation of different approaches to solve the introduced problem. The main characteristic of this problem is its non-polynomial size. Hence, it is necessary to implement a Branch-and-Price algorithm to solve the instances of large size. Furthermore, to help the Branch-and-Price tool, some non-exact techniques are proposed: various heuristic based on our assumptions over clever ways to prioritize the hubs. In addition, a matheuristic algorithm tries to improve the solution space. The obtained results are comforting, as we can plan and design with a good tolerance real-world situations in only some hours.

The remaining part of the thesis is structured as follows. Chapter 2 presents a recall of the theoretical concepts the work is based on. Chapter 3 introduces the problem, and presents the relevant literature review of similar problems. In the Chapter, we also present two mathematical formulations of the considered problem. The proposed solution approaches are discussed in Chapter 4. Then, in Chapter 5 we present and discuss the results of the computational experiments to evaluate the effectiveness of the adopted solution methods. Finally, Chapter 6 concludes the work with some remarks on the results obtained and some suggestions for possible future developments.
Further, the developed code for the creation of the models of our problem and their resolution with the proposed approaches is reported in Appendix A, B, C, and D.

Chapter 2

Combinatorial Optimization fundamentals

The aim of this chapter is to introduce the reader the theoretical concepts on which the thesis work is based on. In particular, in Section 2.1 some basic concepts of Operations Research are recalled. Then, in Section 2.2 we present an overview on Column Generation and Branch-and-Price algorithms. Finally, Section 2.3 recalls the heuristic methods for optimization problems.

2.1 Operations Research recalls

This section briefly recalls some basic concepts that we employ in this work. It presents a rapid overview on the foundation of the Operations Research, starting from the introduction to the linear programming models and arriving to the Branch-and-Bound method.

2.1.1 Introduction to Linear Programming models

A linear programming (LP) model is a mathematical model whose requirements are represented by linear relationships. The aim of an LP model is to maximize or minimize an objective function, represented by a linear expression of the problem variables, which is subject to linear inequalities and non-negativity constraints of the variables. In general, a linear program is expressed in the canonical form:

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & \leq & b \\ & x & \geq & 0 \end{array} \tag{2.1}$$

The above expression represents a minimization problem of a linear function $c^T x$ derived from the set of non-negative variables $x \in \mathbb{R}^n_+$, each with a cost – or a profit in case of maximization problems – $c_j \in c^T$. The variables are subject to a set of linear constraints, each represented by a linear inequality $a_j^T x \leq b_j$, where $a_j^T \in A$ and $b_j \in b$ are respectively the array of coefficients and the right-hand side coefficient of the related j-th constraint.
The feasible domain of an LP model can be geometrically interpreted as a polyhedron $\mathcal{P} = \{x \mid Ax \leq b\,,\ x \geq 0\}$, that is a convex set defined as the intersection of finite half spaces, each of which is defined by a linear inequality. Hence, every $x \in \mathcal{P}$ is a feasible solution of the problem.
The fundamental theorem of linear optimization is a consequence: if a linear model $\max\left\{c^T x \mid x \in \mathcal{P}\right\}$ is feasible ($\mathcal{P} \neq \emptyset$) and is not unbounded from below (or from above in case of maximization problems), it must have an optimal solution $x^\star$ of finite objective value $z^\star = c^T x^\star$, and $x^\star$ is an extreme point (vertex) of $\mathcal{P}$.
An equivalent reformulation of any LP model is the standard form, obtained by adding a slack variable $s_j \geq 0$ in each constraint, to transform them into linear equalities $Ax = b$. The standard form does not change the solution space, although it helps to introduce the concept of basic solution[1]. Indeed, $x \in \mathbb{R}^n_+$ is called a basic feasible solution of an LP problem if and only if there is a basis B with $A_B x_B = b$ and $x_N = 0$, where $x_B \geq 0$, and this corresponds to exactly one vertex of $\mathcal{P}$.
The solving method for many LPs is the simplex algorithm (see Dantzig and Thapa (1997)), which starts from a basic solution, and checks if there are some non-basic variables with a negative reduced cost that can replace a variable in the current basis. The reason is that the reduced cost of a variable $r_j = c_j - z_j = c_j - \lambda_j a_j$ represents the amount by which the objective function coefficient would have improved if the variable entered the basis. In particular, λ_j are the simplex multipliers associated with the j-th constraint. There are some pivoting rules to choose both the entering non-basic variable and the leaving basic variable. In any case, the simplex algorithm iterates until the optimality condition for a basic feasible solution is fulfilled: all the variables' reduced costs are non-negative ($r_j \geq 0 \ \ \forall\, j$).

In integer linear programming (ILP) or mixed integer linear programming (MILP) models, all or some of the variables are required to be integer $x \in \mathbb{Z}^n_+$. In these cases the smallest polyhedron $\mathcal{P}$ that contains $\mathcal{H}$ is called convex hull $conv(\mathcal{H})$ and comprises only vertices with integer coordinates. Thus, solving an LP over $conv(\mathcal{H})$ automatically gives an integer solution.

[1] A basis $B = (B_1, ..., B_m)$ is an ordered subset of m indices of linear independent columns $A_{B_1}, ..., A_{B_m}$ of $A \in \mathbb{R}^{m\times(m+s)}$ – where $m + s = n$ – whereas all the other s column indices are a non-basis $N = (N_1, ..., N_s)$

2.1.2 The concept of duality

Another milestone in linear programming is the concept of duality. Indeed, for each LP primal problem $\min\left\{c^T x \mid Ax \geq b\,,\ x \geq 0\right\}$ is always possible to write its corresponding dual problem by associating with every primal constraint $a_j^T x \gtreqless b_j$ a dual variable λ_j – that is ≥ 0, free or ≤ 0 according to the constraint sense, and with every primal variable $x_j \gtreqless 0$ or free a dual constraint $\lambda^T A_j \lesseqgtr c_j$ – where λ^T is the row array comprising all the λ_j. The sense of the dual constraint is opposite to the sign of the primal variable – so a dual constraint of minority is related to positive primal variable, and vice versa, whereas an equality constraint corresponds to a primal free variable. The related dual objective function is $\max \lambda^T b$.

The strength of this concept is resumed by two theorems:

Weak Duality given a primal minimization problem with a feasible solution x and its corresponding dual problem with feasible solution λ, then $c^T x \geq \lambda^T b$. A direct consequence is the infeasibility criterion of a LP problem, because when one of the two problems has an unbounded objective function, the other one does not have feasible solutions.

Strong Duality if the primal LP has a feasible optimal solution $x^\star$, then the dual LP has a feasible optimal solution $\lambda^\star$, and the respective optimal objective values coincide: $c^T x^\star = \lambda^{\star T} b$.

What clearly emerges from these two theorems is that the primal optimality and the dual feasibility are the same concept: dual variables corresponding to primal active constraints take the role of basic variables.

2.1.3 The Branch-and-Bound method

A method to solve a generic MILP model $\min\left\{c^T x \mid x \in \mathcal{H}\right\}$ is the Branch-and-Bound (denoted as B&B). This method solves optimization problems by breaking them down into smaller sub-problems and using a bounding procedure to prune the search space $\mathcal{S} \subseteq \mathcal{H}$ and eliminate sub-problems that cannot contain the optimal solution.
The search space $\mathcal{S}$ is a rooted tree of candidate solutions, and exploring branches of this tree means to check against upper and lower estimated bounds on the optimal solution, and discard a node of the tree (i.e., a sub-problem) if it cannot produce a better solution than the best one found so far. Hence, the B&B depends on efficient estimation of the lower and upper bounds of branches, and it performs an exhaustive search if there are no bounds available.

The first step of the B&B algorithm is to solve the continuous relaxation of the MILP, obtaining the corresponding linear problem LP_0, which is the so-called root node of the search tree. If an optimal solution of LP_0 is integer, it also corresponds to an optimal solution of the problem $x^\star$ and the algorithm ends. Although if it is not integer, the branching starts by selecting a fractional variable in the LP_0 solution and then splitting the root node into two or more sub-problems which both have an additional constraint for the selected variable.
For instance, in a binary branching, in one of the two sub-problems the branching variable is bounded from below by its integer rounding up, whereas in the other sub-problem the variable is bounded from above by its integer rounding down. Then, every sub-problem LP_t is solved, and eventually the branching procedure is repeated. In particular, if the sub-problem has an integer solution value better than the current best integer solution of the problem, that value becomes the current primal bound PB of the problem (which is an upper bound for minimization problems and a lower bound for maximization ones).
However, not all the nodes are explored, thanks to the pruning of unpromising subtrees. In order to close the current node t and not to generate its subtrees, the prune can be by infeasibility, by integrality or by bound. Actually, the procedure evaluates its LP_t and closes the node if the problem is infeasible (also its children will be infeasible), if it has an integer solution (no more branching variables), or if its dual bound is no better than the current primal bound (also its subtrees will not produce a better solution). The reason of the prune by bound is that the value of any integer feasible solution with value z of a mixed integer problem $\min\left\{c^T x \;\mid\; Ax \geq b\,,\;\; x \in \mathbb{Z}_+^n \times \mathbb{Q}_+^q\right\}$ gives a primal bound on the optimal solution $z^\star$, i.e., $z \geq z^\star$. Whereas optimizing over any relaxation of the MILP gives a dual bound on the optimal solution $z^\star$, i.e., $z \leq z^\star$ (the signs of inequalities are inverted in case of maximization MILPs). Hence, if the found dual bound of LP_t is worse than the current best primal bound, there cannot be obtained improvements in its children.

2.2 Column Generation and Branch-and-Price

2.2.1 The Column Generation method

As pointed out by Nemhauser (2012), column generation (denoted as CG) refers to linear programming algorithms designed to solve problems in which there is a huge number of variables compared to the number of constraints, and the simplex algorithm step of determining whether the current basic solution is optimal or finding a variable to enter the basis is done by solving an optimization problem rather than by enumeration. Indeed, the main idea of CG is to start solving the considered program with only a subset of its variables. Then, iteratively, variables with potentialities to improve the objective function are added to the problem. This dynamic variables' addition occurs via the insertion of the column-coefficients into its constraint matrix, hence the name of the method.
The hope when applying a CG algorithm is that only a very small fraction of the variables' columns will be generated. This hope is supported by the observation that for large problems a considerable majority of the columns is irrelevant for solving the problem. As a matter of fact, most columns will be non-basic and have their corresponding variable equal to zero in any optimal solution. Thus, there is no difference if they are or not in the model because the optimal solution can be found without them.
Desrosiers and Lübbecke (2005) state that the column generation algorithm is the primal simplex algorithm with a minor but essential difference in the pricing step: rather than explicitly calculating the reduced costs of variables, the former solves an auxiliary optimization program that implicitly searches for a variable of negative reduced cost, or proves that none exists. In particular, the algorithm considers two problems: the restricted master problem and the sub-problem. The restricted master problem (denoted as RMP) is the original problem that considers only a subset of variables, whereas the sub-problem is a new problem created to identify an improving variable to be added to the RMP.
The original linear program containing many variables – indexed by the set $\mathcal{X}$ – to solve is called master problem (denoted as MP), and it is assumed feasible and with finite objective value:

$$
\begin{aligned}
z^{\star}_{\mathrm{MP}} := \min \quad & \sum_{x \in \mathcal{X}} c_x \lambda_x \\
\text{s.t.} \quad & \sum_{x \in \mathcal{X}} a_x \lambda_x \leq b \\
& \lambda_x \geq 0 \qquad \forall\, x \in \mathcal{X}
\end{aligned}
\tag{2.2}
$$

The first step of the algorithm is to choose a small subset of variables $\mathcal{X}' \subset \mathcal{X}$ and build the so-called restricted master problem, which is assumed feasible from the choice of the restricted subset:

$$
\begin{aligned}
z_{\mathrm{RMP}} := \min \quad & \sum_{x \in \mathcal{X}'} c_x \lambda_x & & \\
\text{s.t.} \quad & \sum_{x \in \mathcal{X}'} a_{ix} \lambda_x \leq b_i & & \forall\, i \in \{1, ..., m\} \\
& \lambda_x \geq 0 & & \forall\, x \in \mathcal{X}' \subset \mathcal{X}
\end{aligned}
\tag{2.3}
$$

Let π be the non-negative dual vector associated with the inequality constraints of the master (2.2), the sub-problem called pricing problem (denoted as PP) implicitly computes reduced cost $\bar{c}(\pi)$ amongst all $\bar{c}_x = c_x - \pi^T a_x$ of all the variables $\lambda_x \;\; \forall\, x \in \mathcal{X}$:

$$
\begin{aligned}
\bar{c}(\pi) := \min_{x \in \mathcal{X}} \quad & c_x - \sum_{i=1}^{m} \pi_i a_{ix} & & \\
\text{s.t.} \quad & c_x = c(x) & & \\
& a_{ix} = a_i(x) & & \forall\, i \in \{1, ..., m\}
\end{aligned}
\tag{2.4}
$$

Solving the sub-problem (2.4) leads to two possible scenarios:

- If $\bar{c}_x \geq 0 \;\; \forall\, x \in \mathcal{X}$, then $\bar{c}(\pi) \geq 0$, which proves the optimality of the master problem (2.2) and in particular, the optimal solution of the MP is found by solving the RMP $z^\star_{\mathrm{MP}} = z_{\mathrm{RMP}}$ because $\lambda^\star_x = \lambda_x \;\; \forall\, x \in \mathcal{X}'$ and $\lambda^\star_x = 0 \;\; \forall\, x \in \mathcal{X} \setminus \mathcal{X}'$.
- Otherwise, if $\bar{c}_x < 0$, a new variable λ_x, where $x \in \mathcal{X} \setminus \mathcal{X}'$, with a negative reduced cost will be added to the RMP (2.3) by adding x to $\mathcal{X}'$. As a consequence, the RMP will be re-optimized with added column a_x of cost c_x to obtain a new λ and a new π to be passed to the PP.

Furthermore, Desrosiers and Lübbecke (2005) pointed out another important property of the column generation algorithm: the use of bounds. Actually, the RMP is a restriction of the MP, thus z_{RMP} iteratively approaches $z^\star_{\mathrm{MP}}$ from above and so it represents an upper bound for the optimal value. Additionally, the presence of a lower bound allows to evaluate the current solution quality. It is possible to establish the master problem lower bound – the so-called Lagrangian bound – from the value $\kappa \geq \sum_{x \in \mathcal{X}} \lambda_x$. Indeed, the objective value of the RMP cannot be reduced by more than κ times the smallest reduced cost $\bar{c}^\star(\pi)$:

$$
z_{\mathrm{RMP}} + \kappa \bar{c}^\star(\pi) \;\leq z^\star_{\mathrm{MP}} \leq\; z_{\mathrm{RMP}} \tag{2.5}
$$

The above condition (2.5) proves the optimality of the master problem when there are no more variables with negative reduced cost (as $\bar{c}^\star(\pi) = 0$).

The pricing problem offers large opportunities for speeding up the overall CG process, since it is usually solved very often. Indeed, it is better to perform the so-called heuristic pricing, where there are specific choices about the subset of variables on which computing the reduced costs and how variables are picked from that subset, in order to solve the PP to optimality only in the last CG iteration to prove the optimality of the MP.

The CG iterative process can be resumed by the following sketch:

Algorithm 1.1: Column Generation Algorithm

input : RMP with feasible subset $\mathcal{X}' \subset \mathcal{X}$, PP
output : Optimal primal-dual solutions $\lambda^\star_{\mathrm{MP}}$, $\pi^\star$ and optimum $z^\star_{\mathrm{MP}}$ for the MP

repeat
  Solve the RMP to obtain an optimal primal-dual solutions λ_{RMP}, π of cost c_{RMP}
  Solve the PP to obtain the minimum reduced cost $\bar{c}(\pi)$ with corresponding $x \in \mathcal{X}$
  Generate the variable λ_x to add to the RMP with encoding $\begin{bmatrix} c_x \\ a_x \end{bmatrix}$ via $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{x\}$
until $\bar{c}(\pi) \geq 0$
return λ_{RMP}, π *and* z_{RMP}

2.2.2 Dantzig-Wolfe Decomposition

An extension of the CG algorithm is the Dantzig-Wolfe decomposition: an algorithm for solving linear programming problems with special structure. Actually, this is a mathematical reformulation to express some constraints under an alternative geometric interpretation, deriving the master and the pricing problem from it rather than by direct construction.
Indeed, the DW decomposition relies on the Minkowski-Weyl theorem (see Schrijver (1986)). The latter affirms that there are two equivalent representations of a polyhedron: the half-spaces one $\mathcal{X} = \{x \in \mathbb{R}^n \mid Dx \geq d\}$ and the vertex one through the polyhedron's extreme points $\{x_p\}_{p \in P}$ and extreme rays $\{x_r\}_{r \in R}$. Hence, this theorem warrants that each $x \in \mathcal{X}$ can be represented as a convex combination of extreme points plus a non-negative combination of extreme rays of the polyhedron:

$$x = \sum_{p \in P} x_p \lambda_p + \sum_{r \in R} x_r \lambda_r, \quad \sum_{p \in P} \lambda_p = 1, \quad \lambda \in \mathbb{R}_+^{|P|+|R|} \tag{2.6}$$

In particular, given an LP problem, the DW decomposition groups its constraints in two subsets, reformulates one of them with the Minkowski-Weyl theorem, and performs a substitution in the other and in the objective function. Thus, from the LP compact problem $z^\star := \min \left\{ c^T x \mid Ax \geq b\,,\ Dx \geq d\,,\ x \in \mathbb{R}^n_+ \right\}$, applying the reformulation on $\mathcal{D} = \left\{ Dx \geq d\,,\ x \in \mathbb{R}^n_+ \right\}$, the equivalent extensive formulation is:

$$\begin{aligned}
z^\star := \min \quad & \sum_{p \in P} (c^T x_p)\lambda_p + \sum_{r \in R} (c^T x_r)\lambda_r \\
\text{s.t.} \quad & \sum_{p \in P} (A x_p)\lambda_p + \sum_{r \in R} (A x_r)\lambda_r \geq b \\
& \sum_{p \in P} \lambda_p = 1 \\
& \lambda_p \geq 0 \qquad \forall\, p \in P \\
& \lambda_r \geq 0 \qquad \forall\, r \in R \\
& \sum_{p \in P} x_p \lambda_p + \sum_{r \in R} x_r \lambda_r = x
\end{aligned} \tag{2.7}$$

Then, from the master problem (2.7), it is possible to derive the RMP expressed with the relative small subsets $P' \subset P$ and $R' \subset R$ and obtain its primal solution λ of cost z_{RMP} with the dual values π_b, associated with the substituted constraint, and π_0, referred to the convexity constraint $\sum_{p \in P} \lambda_p = 1$. The dual values are used to find the negative reduced cost variables in the pricing problem $\bar{c}^\star := \min_{x \in \mathcal{D}} \bar{c}_x(\pi_b, \pi_0) = \min \left\{ \min_{p \in P} \bar{c}_p\,,\ \min_{r \in R} \bar{c}_r \right\}$ with $\bar{c}_p = c_p - \pi_b^T a_p - \pi_0 \ \ \forall\, p \in P$ and $\bar{c}_r = c_r - \pi_b^T a_r \ \ \forall\, r \in R$.

Thus, the DW reformulation pricing problem is:

$$c^\star := -\pi_0 + \min\Big\{(c^T - \pi^T A)x \ \mid \ x \in \mathcal{D}\Big\} \tag{2.8}$$

Also in this case, the CG terminates when $\overline{c}^\star \geq 0$ as there are no more negative reduced cost columns. Otherwise, if $\overline{c}^\star \leq 0$ and finite, the (2.8) solution is an extreme point x_p and the new column $\Big[c^T x_p, (Ax_p)^T, 1\Big]^T$ is added to the RMP. Whereas if $\overline{c}^\star = -\infty$ it is possible to identify an extreme ray x_r as solution to $(c^T - \pi^T A)x = 0$ and add the relative column $\Big[c^T x_r, (Ax_r)^T, 0\Big]^T$ to the RMP. However, the bounds condition (2.5) is modified removing the κ factor, because the latter is 1 in case of finite negative solution to (2.8) or does not matter if $\overline{c}^\star = -\infty$.

Furthermore, it is possible to extend the DW decomposition to integer linear programs or mixed integer linear programs. The difference with LPs is that variables are in $\mathbb{Z}^n_+$ – and not in $\mathbb{R}^n_+$ – so the Minkowski-Weyl theorem is applied on the convexification of the reformulated domain. Indeed, in MILPs each $x \in \mathcal{D}$ can be represented as a convex combination of extreme points plus a non-negative combination of extreme rays of the domain's convex hull $conv(\mathcal{D})$.

Moreover, the DW decomposition can be applied to the dual problem. This is often used for reformulating MILPs with two linked sets of variables, as these problems deal with complicating variables instead of complicating constraints. This reformulation is named Benders decomposition and is a "row generation" as it iteratively generates new inequalities to add to the master problem (see Benders (1962)). The strategy is to divide the variables of the original problem into two subsets so that a first-stage master problem is solved over the first set of variables, and the values for the second set of variables are determined in a second-stage sub-problem for a given first-stage solution. Next, if the pricing problem determines that the fixed first-stage decisions are infeasible, a new row is generated and added to the MP, which is re-solved until no more inequalities can be generated.

2.2.3 Branch-and-Price

As stated by Savelsbergh (2001), the Branch-and-Price is a generalization of the LP-based Branch-and-Bound specifically designed to handle MILPs containing many variables. The Branch-and-Price (denoted as B&P) method can be seen as a hybrid between column generation and Branch-and-Bound, as its basic idea is to apply CG at every explored node of the B&B search tree. Indeed, at the start of the algorithm some columns are left out of the LP relaxation because most of them will have their associated variable equal to zero in an optimal solution. Then, identically to CG, to check the optimality of an LP solution, the pricing problem

is solved to try to find columns with a negative reduced cost. If such columns are identified, the LP is re-optimized. Otherwise, the branching occurs when no profitable columns are found and the LP solution is not integer.
Moreover, Barnhart et al. (1998) highlighted some difficulties in the B&P application due to the so-called tailing-off effect of the column generation: the large number of iterations needed to prove the optimality of the LP solution, which can potentially happen at every node of the search tree. Fortunately, the B&B framework has some inherent flexibility that can be exploited, and so instead of solving the LP to optimality, the CG could be prematurely ended to work with bounds on the final LP value. Actually, this is the reason most B&P algorithms are problem-specific since the problem must be formulated in such a way so that effective branching rules can be established. However, it is crucial that the pricing problems are aware of the branching decisions to avoid the generation or regeneration of columns which violate them.
In particular, the inequality (2.5) offers a large opportunity for speeding up the B&P process, by means of the so-called early branching: a technique that typically goes along with the computation in each pricing iteration of a Lagrangian bound $LB = z_{\text{RMP}} + \kappa\overline{c}^{\star}(\pi)$, which represents the lower bound of the current Branch-and-Bound node. Hence, when in a B&B node the Lagrangian gap $LG = \frac{LB - z_{\text{RMP}}}{LB}$ is lower than a certain stop-early threshold, the pricing iterations are stopped on that node and the Branch-and-Price proceeds analyzing the next node of the search tree – if exists.

Besides, an efficient extension of the B&P algorithm is the Branch-and-Price-and-Cut (denoted as B&P&C), that involves the use of cutting planes. Gilmore and Gomory (1961) introduced cutting planes as valid linear inequalities added to the problem in order to iteratively refine a feasible set or the objective function. Such procedures are commonly used to find optimal integer solutions to MILPs. Indeed, by solving the linear relaxation of the given feasible MILP, there will always be an optimal extreme point. But if the latter is not integer, there is guaranteed to exist a linear inequality that separates the optimum from the convex hull of the true feasible set. Such an inequality is a cutting plane that can be added to the relaxed LP.
In the B&P&C context, cutting planes $Fx \leq f$ can be directly included in the original problem with a consequent change only in the PP's objective function (as there is a corresponding dual value α). Alternatively, cutting planes can be enforced only in the pricing problem by simply reducing its domain from $\mathcal{D}$ to $\mathcal{X}_F = \{x \in \mathcal{D} \mid Fx \leq f\}$, and this can even lead to a stronger dual bound.

2.3 Heuristic methods

This last section of the chapter presents an overview on the resolution methods which differ from the previously introduced in this chapter, that are exact approaches leading to optimal solutions.

2.3.1 Heuristic algorithms

The term "heuristic" is derived from the Greek word "heurisko" which means "I find, discover". Actually, heuristics – also known as heuristic techniques – refer to any method of problem-solving that relies on practical approaches that may not be perfect, rational, or optimal but can still achieve an immediate estimation. As highlighted by Pearl (1984), heuristic strategies are based on prior experiences with similar problems. Typically, these methods both yield the desired outcome and expedite the process of finding a suitable solution in impractical situations, but sometimes, they can lead to systematic errors.
In particular, in the context of mathematical programming, a heuristic algorithm is a procedure that determines feasible near-optimal solutions to an optimization problem, which is an NP-hard problem by itself. The corresponding trade-off is that the algorithm may sacrifice optimality, completeness, accuracy, or precision in favor of speed (see Eiselt and Sandblom (2000)). Nonetheless, heuristics are extensively used for various reasons, such as for problems that do not have a precise solution or whose formulation is unknown, when the computation required for a problem is complex, or for calculating bounds on an optimal solution in Branch-and-Bound solution processes.
Optimization heuristics are divided into two main categories based on the organization of the solution domain:

Construction Methods These are also known as greedy algorithms. They operate in phases, with each step optimizing the choice in an attempt to find the overall optimal solution for the problem. A well-known example of these methods is used in the famous "travelling salesman problem", by visiting the closest unvisited city at each step of the journey.

Local Search Methods These techniques use an iterative approach: beginning with an initial solution, they explore the current solution's neighborhood and eventually replace it with a better one. A common example of this heuristic strategy is represented by swapping positions of jobs to be processed in manufacturing systems, with the goal of minimizing the completion time.

Nevertheless heuristic algorithms do not represent a universal result, they are beneficial tools when it is not possible to utilize exact methods. Indeed, the use of these techniques has become important in solving current complex real-world problems in different field of applications. The reason is that they can provide adaptable strategies for solving complex problems with the benefit of being easily implemented and requiring less computational power. Besides, throughout the years, these algorithms have advanced, leading to the creation of hybrid systems that incorporate selected aspects from different types of heuristics.

2.3.2 Matheuristics

A strong application of heuristic techniques is represented by the so-called matheuristics. Boschetti and Maniezzo (2022) describe matheuristics as optimization algorithms, not specific to any particular problem, which use mathematical programming techniques to generate heuristic solutions. The elements that are specific to a problem are only incorporated in the lower-level mathematical programming, local search, or constructive components. Actually, as highlighted by Martina Fischetti and Matteo Fischetti (2016), the hallmark of matheuristics is the central role played by the mathematical programming model, around which the overall heuristic is built. They utilize some of the features that are derived from the mathematical model of the problem of interest in part of the algorithm. However, creating an effective heuristic is a skill that cannot be constrained by precise guidelines. The latter concept is especially accurate with matheuristics, which are not a fixed paradigm but rather a conceptual framework for designing heuristics that are mathematically sound.
As observed in Matteo Fischetti and Lodi (2011), an early demonstration of the effectiveness of the matheuristic concept is the general-purpose local branching strategy (see Lodi and Matteo Fischetti (2003)). The latter technique shares similarities with local search heuristics, but instead of using specific neighborhoods, it introduces general linear inequalities to the MIP model. Although it is an exact method, it is intended to enhance the heuristic behavior of the MIP solver by alternating strategic branchings to define solution neighborhoods and tactical branchings to explore them. The outcome is a high-quality solution early in the computation process. Therefore, it is reasonable to solve heuristically auxiliary MIPs, instead of LP relaxations.
This local branching strategy can be considered as a precursor of matheuristics. Indeed, given the reference solution $\overline{x}$ of a MIP with a non-empty set of binary variables $\mathcal{B} \neq \emptyset$, it aims to improve the solution by a specific not-too-far neighborhood $\mathcal{N}$:

$$\triangle\,(x\,,\overline{x}) \;=\; \sum_{j\,\in\,\mathcal{B}\,:\overline{x}_j=0} x_j \;+\; \sum_{j\,\in\,\mathcal{B}\,:\overline{x}_j=1} (1-x_j) \;\leq\; \mathcal{N} \tag{2.9}$$

Chapter 3

Presentation of the Problem

In this chapter we introduce the practical literature background from which our problem object of study is derived: the service network design models and the hub location problems. Then, we describe our Service Network Design and Hub Location Problem (which will be denoted as SNDHLP from now on), contrasting and comparing it with similar problems already studied in the literature over the years. The mathematical notation of the SNDHLP is described in Section 3.3. Moreover, in Sections 3.4 and 3.5, we present the mathematical statements of our problem, formalizing the two possible approaches treated in our work. Finally, the last Section 3.6 compares these two formulations.

3.1 Background and Description of the Problem

3.1.1 Problem Background

The Service Network Design and Hub Location Problem takes into account two main decision aspects: the strategic decision for facility locations and the tactical planning of the freight transportation. The main focus of these decisions is the firm's efficiency in terms of profitability and service performance. The Cambridge English Dictionary (2023) provides multiple definition for the term "service" depending on the context. The most accurate description of service for our problem is referred to the union of some of them. From our point of view, a service is the act of doing a helpful activity for someone else, which involves dealing with customers and providing a particular thing people may necessitate. Specifically, the service of our interest is the freight transport, which is the physical process of transporting commodities and merchandising goods and cargo, by using one or more way of shipment (McLeod and Curtis (2020)).

In general, network design models are widely used to represent strategic planning issues in transportation systems and not only – telecommunications, logistics, and production-distribuition systems, etc. In particular, as stated by Crainic (2000), a service network design problem (SNDP) is typically developed to assist the set of main tactical issues and decisions relevant for the transportation of goods: the selection and scheduling of the services to operate, the specification of the terminal operations, and the routing of freight. The focus is both on ensuring firm's profitability and answering service demand, especially for transportation systems where it is not possible to perform a tailored service for each customer and there are one or more vehicles which move goods of different origins and destinations in the network. This complex management of operations becomes all the more important the larger the distance between locations.

Furthermore, in multimodal transportation systems are frequently used hub networks to route commodities between many origins and destinations. Contreras (2021) defines hub networks as hierarchical structure where there are an access-level network, which connects the origin and destination nodes to hubs, and a hub-level network connecting hub nodes between them. In these networks, hubs are usually central facilities which work as sorting, transshipment, and consolidation points for commodities. Hence, instead of sending flows directly from origin to destination, hub facilities connect numerous origin-destination pairs by using a few links, in order to reduce set-up costs and enable economies of scale on routing costs through the flows' consolidation.
Among the hub network design problems, the hub location problem (HLP) aims to find the location of hubs and the allocation of demand nodes to these located hub nodes (Morton O'Kelly (1986)). Alumur et al. (2012) underline that in most of the studies in the HLPs' literature, some assumptions are taken in consideration to simplify the decisions: fully-interconnected hubs, as the hub arcs network connecting the hub nodes is assumed to define a complete graph on the set of hub nodes (presence of a direct hub link between every hubs' pair), no set-up costs for hubs and their links, and frequently origin-destination routes including hubs to avoid direct connections between customers pairs.
In addition, Contreras (2021) draws attention to how the arc selection decisions in the HLP hinges on the possible allocation strategies of origin/destination nodes to hubs:

- Multiple assignments, which is the simplest case as origins and destinations can be connected to more than one hub facilities. This allows a larger flexibility in the hub networks but could increase network design costs for the activation of the access arcs. The latter is not the case of freight transportation, as the access arcs correspond to already existing physical infrastructures and have no set-up costs. However, this allocation strategy might still be prohibitively expensive because it requires the presence of available vehicles to operate over multiple connections. So the choice strictly depends on the specific application.

- Single assignments, in which an origin or destination node is associated to only one hub facility, and so all the goods with same origin or destination are routed via the same access arc. This strategy is quite common and very useful in telecommunications or in small quantities' transportation where there are consolidated commodities to send to the same sorting point.
- n-allocation strategy, that represents the generalization and the trade-off of the other two assignments methods, as in this case the origin or destination node can be linked to at most n hubs.

3.1.2 Problem Description

The reason why in Section 3.1.1 we present the service network design problem and the hub location problem is that the problem we are dealing with is simultaneously concerned about decisions on the hubs' locations and on the tactical planning of request routing. Indeed, in the context of freight transportation, our attention is directed towards the intermodal transport of goods.
As reported by the Logistische Informations Systeme AG (2023), intermodal transport refers to a transport chain in which two or more modes of transport are used, whereby the transported goods themselves are not transhipped, but only the loading unit changes the mode of transport. The distinguishing features of intermodal transport are related to its potentialities of saving costs and gaining efficiency, from an environment-friendliness' perspective too. Indeed, this method improves security, reduces damage and loss, and permits freight to be transported faster.
In particular, the focus of our problem is on a special type of intermodal transport: the combined transport, which has the additional characteristics that the main part of the transportation is performed by the transfer vehicles between the internal hubs network, and only the initial and the final leg of the trip are carried out by the vehicles on the access links to/from the transshipment network. Actually, we study a real-world integrated tactical Service Network Design and Hub Location Problem for combined transport. Consequently, the problem has a greater complexity than the other two single cases of SNDP and HLP, but at the same time provides more efficient solutions.
In our SNDHLP, there is a given set of transport requests. Each of them consists of a certain number of demand units that must be transported from a customer origin location to a customer destination location. Each customer location may be the origin and/or the destination of more than one request, but all the demand units with the same origin and the same destination constitute one request. In

order to perform what a combined transport is, the requests' demand units must be transshipped at hubs. Indeed, we are taking into consideration the common HLP's assumption that commodities have to be routed via at least one hub, and so the direct connection origin-destination is forbidden.
In the initial stage all hubs are closed, but there is a fixed limited number of hubs that must be opened, without any set-up costs for opening. The decision which hubs to activate and use is part of the problem. The allocation strategy adopted for origin/destination links to the hubs network is the multiple assignments one, in order to allow, if the hubs are open, the customer to send goods over multiple routes, also for the same single request. In fact, requests are splittable, which means that the demand units of a request are not obliged to follow the same routing path, but they might be divided over several routing itineraries. Besides, the request's different itineraries may have also different start and/or end in the hubs' network. In particular, for each customer, we can define the set of allowed hubs, comprising all the hubs for which there is a direct link connecting the hub to the customer. If this link starts from the customer, the specific hub is an allowed start-hub for the customer, whereas if it starts from the hub, the latter is an allowed end-hub for the customer. We assume that, among the allowed hubs of a customer, there is also its own location as both an allowed start-hub and an allowed end-hub, with a relative costless link. These direct links from/to customers to/from their allowed hubs are called access arcs, as they provide the access to the hubs' network, and do not have any capacity limit. Further, the links between any two hubs in the hubs' network are called intra-hubs arcs or transfer arcs. These have a modular capacity that needs to be provided by operating identical vehicles – every with the same transportation capacity.
The only condition the transshipment in the hubs must satisfy is that the first hub in which the request's commodities are transshipped is an allowed start-hub for the origin customer, and specularly the last hub is an allowed end-hub for the destination customer. As we consider the multiple assignments' strategy for non-hub nodes, there might be more allowed start-hubs or end-hubs for each request, and they can also coincide in some routes – the same hub is both start-hub and end-hub (and this represents the simplest case of combined transport).
However, to simplify the problem, there are some necessary assumptions to take into account:

- The transfer-links' capacities are non-restrictive and so, at any point in time, unlimited number of vehicles may use transfer-links without affecting the total travel time of the other vehicles.

- Hubs have non-restrictive transshipment capacities: the same hub can manage an unlimited number of transshipped requests without any waiting time. Thus, scheduling aspects such requests' departure timing are not part of the problem.

- Requests may also start or end at the same location of potential hubs, as long as each origin and destination customer has its own location as allowed hub. This implies that the HLP's assumption of minimum one hub routes is not treated in a strong way, as routes must contain at least one hub, but this can be represented by a hub having the same location of a customer. To clarify, the reason of this assumption is to obtain most rational and logical results for short distances' origin-destination pairs.

- The number of transshipments at hubs – where a transshipment at hub is named hop – for a request is bounded to at most four for the real-world applications of our SNDHLP[2]. This implicitly indicates that the maximal possible length of paths is of five arcs, whereas the minimal is of two (for the previous assumption). The limit of number of hops is again necessary to guarantee reasonable solutions, and avoid long and non-realistic intra-hubs itineraries.

- Every request must be delivered within a maximum transport time, which includes also the transshipments. In this case, the motivation is associated with ensuring adequate service performance.

- The equipment is homogenous: there are only one type of demand unit and vehicle. This justifies why all the transfer vehicles have the same limited capacity, and so determining the number of vehicles to be used over a certain transfer-link is part of the problem.

All these assumptions and the previously described considerations are reported graphically in Figures 3.1 and 3.2, and then compactly resumed in Figure 3.3, with a focus on the actual allowed and not allowed arcs in a possible routing itinerary for an example of request between a customers origin-destination pair.

In conclusion, the problem final goals are to determine the requests' itineraries, to select the hub facilities to be opened, and to compute the number of vehicles operating on each intra-hubs arc. The objective is to minimize the overall requests' combined delivery costs, which include both the costs for operating vehicles in the hubs network and of the costs for using the access arcs.

[2]In any case, where necessary in the models' mathematical formalization, we will always talk about number of hops without explicitly imposing this number to 4

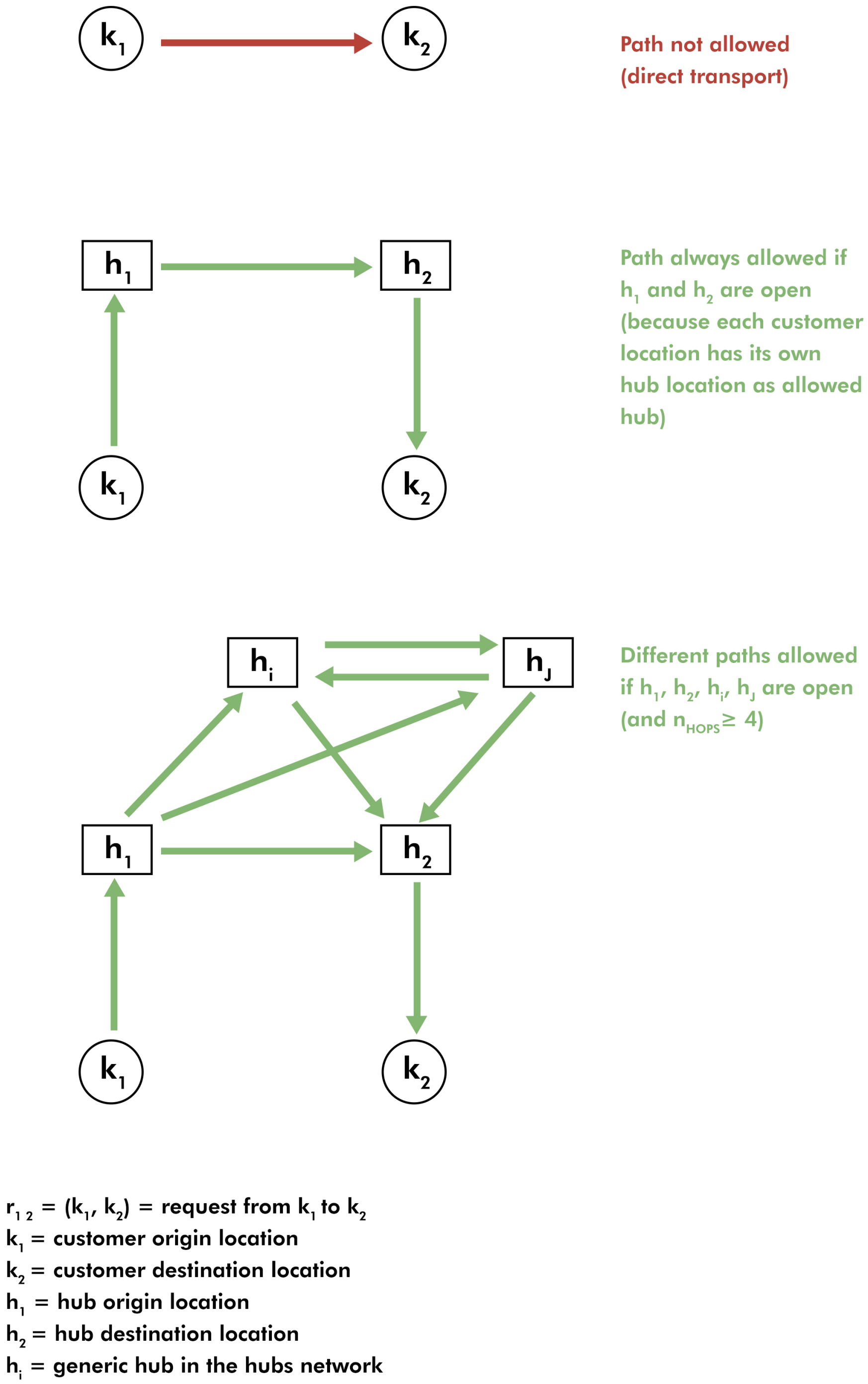

Figure 3.1: Examples of Direct and Combined Transport Paths

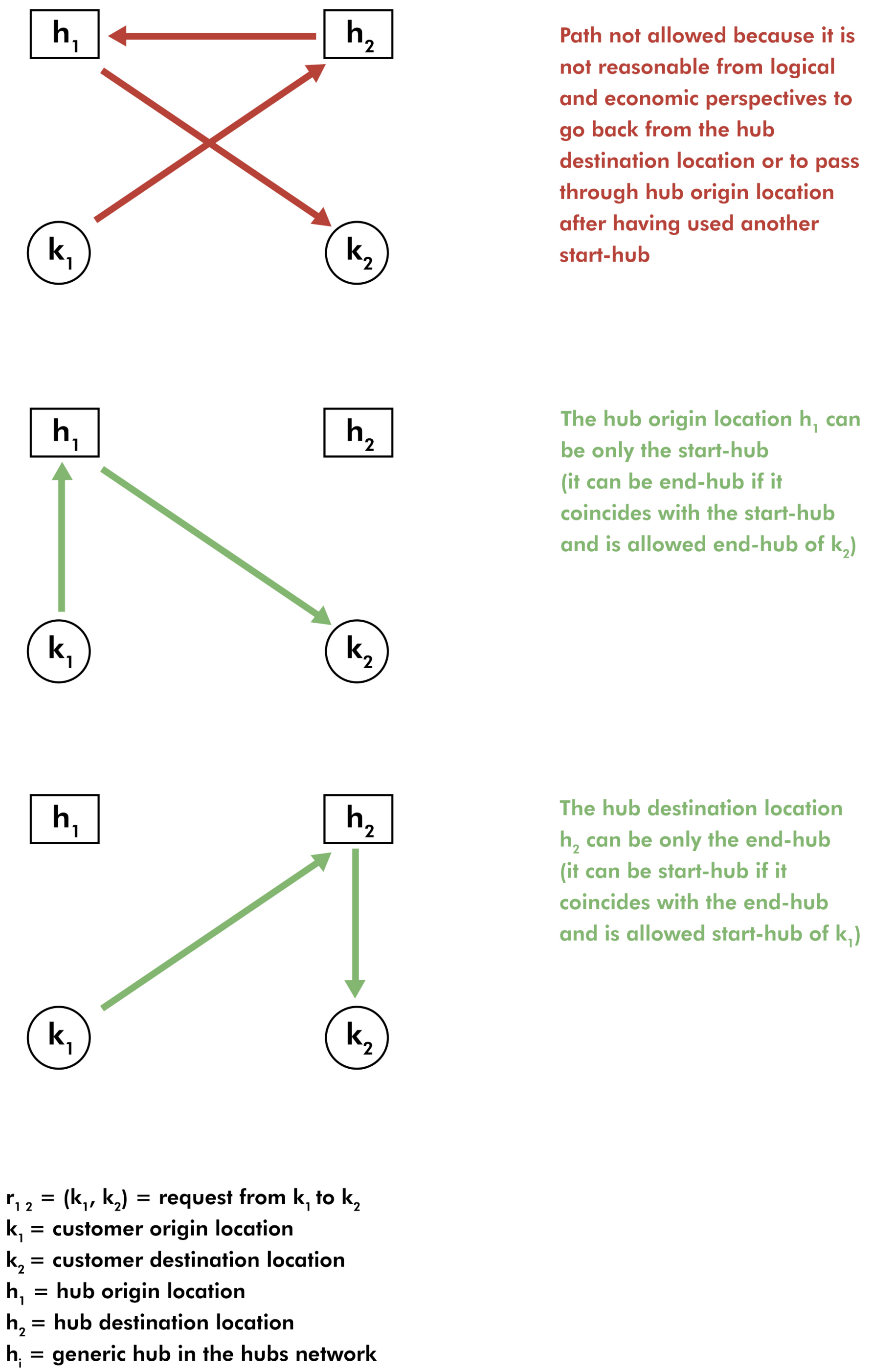

Figure 3.2: Allowed and Not-Allowed Paths including hubs with the same locations of customers

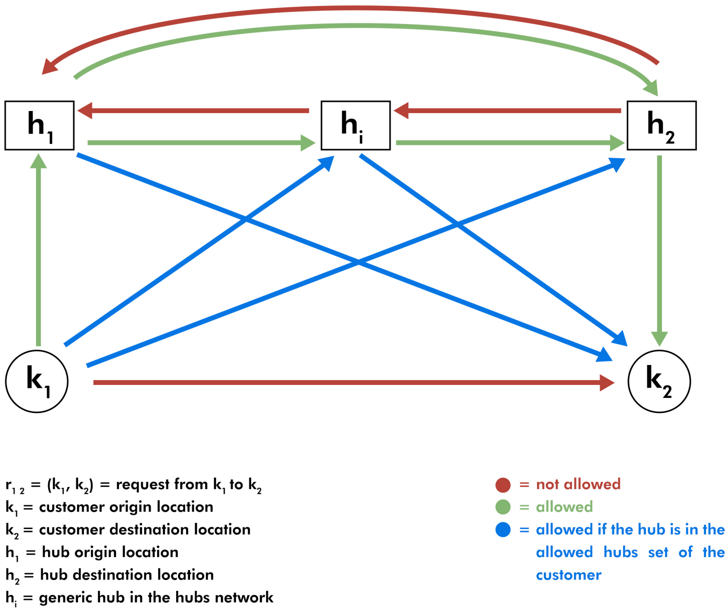

Figure 3.3: Allowed Arcs in Combined Transport Paths

3.2 Literature Review of SNDHLP

From Morton O'Kelly (1986), who was the first to describe an HLP, several researchers have studied the themes of service network design or hub location problem presenting various solution approaches, even if in the majority of the cases they treated only one of the two topics.
Our SNDHLP gives special prominence to the article by Irnich et al. (2016), who analyze the application of a Branch-and-Price-and-Cut algorithm to a SNDHLP which have assumptions similar to ours. Indeed, our model is partially derived from that article, but distinguishes from it as it does not consider the presence of fixed hubs and is not as restrictive in the number of allowed hubs for each customer – they consider in average a maximum of 2.5 potential start- or end-hubs. Another difference is that our problem prohibits the direct transport. Moreover, we differentiate in the solution approaches: we apply the Column Generation only with the Branch-and-Price without any cutting planes, although they also included different types of cuts, and we try to solve the problem with heuristics too.
Regarding the other articles which studied a similar topic, we present a short overview of them (considering only the ones which have as direct application transportation systems). Among these, the majority of them consider, as our SNDHLP, a p-median constraint for the hub location problem, while in the remaining articles there are fixed opening costs for the hubs. In the latter group, the Yoon and Current (2008) direct transport problem embedding a multi-commodity flow model with variable arc capacities was solved via a dual-based heuristic approach. However, their main focus was on small transportation, and so they do not take into account the hops' constraint and the delivery time limit.
Alongside Irnich et al. and Yoon and Current, the number of studies in the literature of SNDHLP which allowed the direct transportation is very restricted. Special relevance assumed the paper by Zhang et al. (2013) because it considered also the costs for CO_2 emissions and had a bi-level heuristic resolution: the upper level searches for an optimal subset of hubs to open, whereas the lower one performs a shortest-path algorithm for the multi-commodity flow assignment over a multimodal network. In addition, they only target the case of unsplittable requests with hub fixed opening costs.
On the other hand, among the past studies that prohibit the direct transport, Campbell (2009) was the first to introduce the time-limit constraint for the delivery of the request's units, even if he did not use any custom algorithm for solving the problem. The main distinguishing factor with us is in the absence of modular transfer arcs capacities. Alumur et al. (2012) took into account the possibility of distinct transportation modes on each transfer arc, with the decision on the design of the hub network that increases in complexity, and an efficient matheuristic was developed to find solutions. However, this increased complexity obliged the authors

to choose a single assignment allocation strategy and not to split requests. A similar heuristic approach for a multimodal transport problem with fixed costs for opening hub facilities was embraced by Serper and Alumur (2016), who proposed a variable neighborhood search algorithm to determine hubs' locations and capacities, transportation modes to serve at hubs, allocation of non-hub nodes to hubs, and the number of vehicles of each type to operate on the hub network to route the demand between origin-destination pairs, but without any time in route limit.
Finally, another interesting study by de Camargo et al. (2017) applied the Benders decomposition technique with special selection/stabilization cuts for the incomplete hub location problem with and without hop-constraints. Besides, they modeled the problem by a Leontief substitution system approach with tight linear bounds that can explicitly incorporate hops constraints for each origin-destination pair of demands. However, the main difference with us is again in the absence of modular intra-hubs arcs capacities.

In conclusion, we can state that our SNDHLP presents a combination of characteristics that has never been treated in this form in the past literature of these topics. In general, the most common assumption is the multiple allocation strategy, even if some studies do not mention it. Whereas, the main distinguishing characteristic of our study from the previous ones is represented by the use of both modular transfer arcs' capacities and maximum delivery time limit. This combination was only present in the Irnich et al. (2016)'s paper. However, in contrast with the latter, we do not allow fixed hubs and direct transport, and we investigate heuristic approaches instead of cutting planes. About the maximum number of hops, this feature was introduced originally by Morton O'Kelly (1986) who allowed only paths with one or two hubs. At the beginning, this was a very easy consideration for simple models, but over the years enlarging the permitted hops to some other numbers was very rare. Regarding the resolution approaches in the literature of the problem, there are several and range from classical enumeration solver to diverse heuristic algorithms, and from Benders decomposition to Branch-and-Price-and-Cut.
The content of this section is resumed in Table 3.1.

	Our SNDHLP	Irnich *et al.* (2016)	Yoon and Current (2008)	Zhang *et al.* (2013)	Campbell (2009)	Alumur *et al.* (2012)	Serper and Alumur (2016)	de Camargo *et al.* (2017)	O'Kelly (1986)
Network Design	✓	✓	✓	✓	✓	✓	✓	✓	
Hub Location Constraint	p-median	p-median	fixed opening costs		p-median	p-median, fixed costs	fixed costs	p-median	p-median
Fixed Hubs		✓	✓						
Limited Number of Hops	4	4						6	2
Maximum Delivery Time	✓	✓			✓	✓	✓		
Splittable Requests	✓	✓	✓		✓			✓	
Unsplittable Requests	✓	✓	✓	✓	✓	✓	✓	✓	
Possibility of Direct Transport		✓	✓	✓					
Arc Capacities	modular	modular	binary	binary	binary	different vehicles	different vehicles	different vehicles	none
Assignment Allocation Strategy	multiple	multiple	multiple	single	multiple	single	single	single	none
Solving Approach	Branch-and-Price and Heuristics	Branch-and-Price-and-Cut	Dual-based Heuristic	Bi-level Heuristic	MIP Solver	Mat-heuristic	Mat-heuristic	Benders Decompo-sition	MIP Solver

Table 3.1: Analogies and differences of the problem characteristics with similar problems of the past literature

3.3 Mathematical Notation of the Problem

We now formalize in mathematical terms what we have delineated in Section 3.1.2. Let $G = (V, A)$ be a digraph[3] where V is the node set and A is the arc set. The set of nodes $V = \{C \cup H\}$ comprises both the set of customer nodes C that are the requests' origins or destinations and the set of hubs H in which requests are temporarily stored during the delivery process. Each request $r \in R \subseteq \{C \times C\}$ contains d^r demand units. These must be delivered within a maximum transport time T^r, and without exceeding the allowed maximum number of transshipments at hubs n_{HOPS}.
Moreover, from the hub set we can define for each customer $k \in C$ the set of its allowed hubs $H_k = \left\{H_k^+ \cup H_k^-\right\} \subseteq H$. In particular, we need to introduce the set $\delta^+(k)$ containing the outgoing arcs from the node k, and the set $\delta^-(k)$ comprising the ingoing arcs in the node k. The set of allowed hubs of a costumer has two constituting subsets: $H_k^+ = \{i \in \{(k,i) \in \delta^+(k)\} \ \ \forall\, i \in H\}$, which contains all the allowed start-hubs for the specific customer k, and $H_k^- = \{i \in \{(i,k) \in \delta^-(k)\} \ \ \forall\, i \in H\}$, which contains all the allowed end-hubs for the customer k. Then, for a specific request $r = (k_1, k_2)$ can be characterized the two sets of allowed start-hubs $H^{+^r} = H_{k_1}^+$ and allowed end-hubs $H^{-^r} = H_{k_2}^-$. In particular, the number of hubs that must be opened for serving all the customer locations is n_{H}.
Further, the arc set A contains all the arcs that connect one node to another, and each arc has a length l_a, expressed in kilometers, and a travel time t_a, expressed in minutes – strictly related to the arc length. Precisely, we can describe the two subsets of A: the transfer arcs $A_t = \{H \times H\}$ between any two hubs, with costs per kilometer and vehicle used c_t, and the access arcs $A_s = \{(k,i), (i,k) \ \ \forall\, k \in C,\ i \in H_k\}$, with costs per kilometer and demand unit transported c_s. The latter set includes the arcs which connect the origins and destinations to the hubs network, and so all the direct links that go from an origin to its allowed start-hubs or that arrive to a destination from its allowed end-hubs. The intra-hubs vehicles have a given transportation capacity K, and the final number of vehicles over a transfer link is clearly a direct consequence of this value.

[3]In graph theory, a directed graph (or digraph) is a graph that is made up of a set of vertices connected by directed edges (see Bang-Jensen and Gutin (2001)).

3.4 A Mixed Integer Linear Programming Model Formulation

The compact formulation of our SNDHLP is an arc-based model. This is a polynomial-sized model which contains for each request a set of variables associated to each arc of the request digraph. Thus, we define a feasible combined transport route for a request $r = (k_1, k_2)$ as an elementary set of arcs $(a_1, \ldots, a_n)$, where $2 \leq n \leq n_{\text{HOPS}} + 1$, connecting the origin k_1 to the destination k_2, and passing at most through n_{HOPS} hubs and at least through one. In the specific eventuality of only one hub routes, this is both an allowed start-hub and an allowed end-hub for the request $i \in \{H^{+^r} \cap H^{-^r}\}$. Besides, all these feasible routes satisfy the condition of having a total transport time lower than the maximal allowed T^r. Another necessary – but implicit in the graph construction – condition is that the first and the last arcs of a request itinerary are access arcs, whereas the eventual arcs in the middle are transfer arcs $a_1 \in \left\{k_1 \times H^+_{k_1}\right\},\; a_n \in \left\{H^-_{k_2} \times k_2\right\},\; a_j \in A_t \;\; \forall j \in \{2, \ldots, n-1\}$. The unsplittable requests' compact model contains three types of decision variables: binary variables h_i to specify if the hub $i \in H$ is opened. General integer variables v_a count the number of vehicles that use the intra-hubs arc $a \in A_t$. Binary variables x^r_a indicate the fraction of the demand units of request $r \in R$ which are transported via the arc $a \in A$.

The mathematical model of the arc-based SNDHLP for the unsplittable requests' case is formalized as follows:

$$
\begin{aligned}
\min \quad & \sum_{a \in A_t} c_t l_a v_a \;+\; \sum_{r \in R} \sum_{a \in A_s} c_s l_a d^r x_a^r && && (3.1)\\
\text{s.t.} \quad & \sum_{a \in \delta^+(k_1)} x_a^r = 1 && \forall\, r = (k_1, k_2) \in R && (3.2)\\
& \sum_{a \in \delta^-(k_2)} x_a^r = 1 && \forall\, r = (k_1, k_2) \in R && (3.3)\\
& \sum_{a \in \delta^+(i)} x_a^r - \sum_{a \in \delta^-(i)} x_a^r = 0 && \forall\, i \in H\,,\ r \in R && (3.4)\\
& \sum_{r \in R} d^r x_a^r \le K v_a && \forall\, a \in A_t && (3.5)\\
& \sum_{i \in H} h_i = n_{\mathrm{H}} && && (3.6)\\
& \sum_{a \in \delta_i^-} x_a^r \le h_i && \forall\, i \in H\,,\ r \in R && (3.7)\\
& \sum_{a \in A} x_a^r \le n_{\mathrm{HOPS}} + 1 && \forall\, r \in R && (3.8)\\
& \sum_{a \in A} t_a x_a^r \le T^r && \forall\, r \in R && (3.9)\\
& h_i \in \{0, 1\} && \forall\, i \in H && (3.10)\\
& v_a \in \mathbb{N}_0 && \forall\, a \in A_t && (3.11)\\
& x_a^r \in \{0, 1\} && \forall\, r \in R,\ a \in A && (3.12)
\end{aligned}
$$

The objective function (3.1) seeks to minimize the total costs incurred by the demand units' transportation over access arcs and by operating vehicles on the intra-hubs links. The three conditions (3.2), (3.3), and (3.4) ensure that every demand unit of a request is transported. In particular for each request $r = (k_1, k_2) \in R$, these three linear equalities correspond to a graph's flow conservation constraints

$$\sum_{a \in \delta^+(v)} x_a^r - \sum_{a \in \delta^-(v)} x_a^r = \begin{cases} 1 & \text{if } v = k_1 \\ -1 & \text{if } v = k_2 \\ 0 & \text{otherwise} \end{cases} \quad \forall\, v \in V,$$

which computes the difference between the sum of outgoing arcs and the sum of ingoing arcs from/to a specific node. Then, the complete delivery of a request is obtained by setting the flow conservation value equal to 1 if a node is the origin of the request, to -1 if it is the request's destination, or to 0 in all the other cases. Constraint (3.5) computes the used capacity and determines the number of necessary vehicles per intra-hubs arc. Constraint (3.6) guarantees that the required number of hubs is opened, whereas condition (3.7) forces the opening of a hub if a

request itinerary passes through it. The two constraints (3.8) and (3.9) explicitly impose a limitation for the number of transshipments and a time limit for the specific request transport. However, the latter two constraints can also be removed to solve a relaxed version of the problem. Finally, the last three constraints (3.10), (3.11) and (3.12) define the domains of the variables.
The above model is for the unsplittable requests' case and so guarantees the transportation of all the demand units of the same request over the same route.

In order to split the requests' routing it is necessary to first modify the constraint (3.12) as follows: $x_a^r \in [0, 1] \quad \forall\, r \in R\,,\ a \in A$. In fact, this changes the domain of request arcs variables, transforming them from binary to continuous in the $\{0, 1\}$ interval. Then, we need to introduce other three variables: binary variables e_a^r indicating if a part of the request r is transported over the arc a or not, integer variables s_i^r counting the number of transshipments of the part of request r up to the hub i, and continuous variables w_i^r representing the transport time of the part of request t until the hub i. These three additional variables are auxiliary to the three decision ones, to warrant the accomplishment of the two constraints of maximum transport time and number of hops of each part of the single requests. In order to satisfy them is sufficient to add to the model three new constraints related to the maximum delivery time and three related to the allowed number of shipments, and then to remove the previous (3.9) and (3.8). Besides, the model comprises a new constraint that links the continuous request arcs variables to the new binary ones and this is the one that actually permits to track the eventual different routes of each request and make them satisfy the constraints of time and transshipments.
Thus, the splittable requests arc-based model can be formalized as follows:

$$\min \quad \sum_{a \in A_t} c_t l_a v_a \;+\; \sum_{r \in R} \sum_{a \in A_s} c_s l_a d^r x_a^r \tag{3.13}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(k_1)} x_a^r = 1 \qquad \forall\, r = (k_1, k_2) \in R \tag{3.14}$$

$$\sum_{a \in \delta^-(k_2)} x_a^r = 1 \qquad \forall\, r = (k_1, k_2) \in R \tag{3.15}$$

$$\sum_{a \in \delta^+(i)} x_a^r - \sum_{a \in \delta^-(i)} x_a^r = 0 \qquad \forall\, i \in H\,,\ r \in R \tag{3.16}$$

$$\sum_{r \in R} d^r x_a^r \le K v_a \qquad \forall\, a \in A_t \tag{3.17}$$

$$\sum_{i \in H} h_i = n_{\mathrm{H}} \tag{3.18}$$

$$\sum_{a \in \delta_i^-} x_a^r \le h_i \qquad \forall\, i \in H\,,\ r \in R \tag{3.19}$$

$$x_a^r \leq e_a^r \qquad \forall\, r \in R\,,\; a \in A \tag{3.20}$$

$$s_i^r \geq e_a^r \qquad \forall\, r = (k_1, k_2) \in R\,,\; a = (k_1, i) \in \delta^+(k_1) \tag{3.21}$$

$$e_a^r + s_i^r \leq s_j^r + n_{\text{HOPS}}(1 - e_a^r) \qquad \forall\, r \in R\,,\; a = (i, j) \in A_t \tag{3.22}$$

$$s_j^r \leq n_{\text{HOPS}} \qquad \forall\, r \in R\,,\; j \in H^{-r} \tag{3.23}$$

$$w_i^r \geq t_a e_a^r \qquad \forall\, r = (k_1, k_2) \in R\,,\; a = (k_1, i) \in \delta^+(k_1) \tag{3.24}$$

$$t_a e_a^r + w_i^r \leq w_j^r + T^r(1 - e_a^r) \qquad \forall\, r \in R\,,\; a = (i, j) \in A_t \tag{3.25}$$

$$t_a e_a^r + w_j^r \leq T^r \qquad \forall\, r = (k_1, k_2) \in R\,,\; a = (j, k_2) \in \delta^-(k_2) \tag{3.26}$$

$$h_i \in \{0, 1\} \qquad \forall\, i \in H \tag{3.27}$$

$$v_a \in \mathbb{N}_0 \qquad \forall\, a \in A_t \tag{3.28}$$

$$x_a^r \in [0, 1] \qquad \forall\, r \in R\,,\; a \in A \tag{3.29}$$

$$s_i^r \in \mathbb{N}_0 \qquad \forall\, i \in H \tag{3.30}$$

$$w_i^r \in \mathbb{R}^+ \qquad \forall\, a \in A_t \tag{3.31}$$

$$e_a^r \in \{0, 1\} \qquad \forall\, r \in R\,,\; a \in A \tag{3.32}$$

The objective function (3.13) seeks to minimize the total costs incurred by the demand units' transportation over access arcs and by operating vehicles on the intra-hubs links. The three conditions (3.14), (3.15), and (3.16) ensure that every demand unit of a request is transported. In particular for each request $r = (k_1, k_2) \in R$, these three linear equalities correspond to a graph's flow conservation constraints, equal to 1 if the node is the origin of the request, to -1 if it is the destination of the request, or to 0 in all the other cases.
Constraint (3.17) computes the used capacity and determines the number of necessary vehicles per intra-hubs arc. Constraint (3.18) guarantees that the required number of hubs is opened, whereas condition (3.19) forces the opening of a hub if a request itinerary passes through it. The constraint (3.20) is the linking condition between continuous and binary request arc variables. The three constraints (3.21), (3.22) and (3.23) define the maximal number of transshipments condition by imposing a limitation to the relative hops-counting variable of the start, middle and end hubs of a request. Similarly, the three conditions (3.24), (3.25) and (3.26) limit the relative time-counting variable of the start, middle and end hubs of a request, representing the maximal allowed time constraint. However, the latter six constraints can also be removed to solve a relaxed version of the problem. Finally, the last six constraints (3.27)-(3.32) express the domains of the variables.

3.5 An Extended Model Formulation

Our SNDHLP can be formalized in mathematical terms with a path-based model too. The latter represents the extended formulation of the SNDHLP. Indeed, the path-based model can be seen as a reformulation of the arc-based one derived from a Dantzig-Wolfe decomposition that generates the paths variables from a combination of the arcs ones, according to the flow conservation constraints. Hence, in this case, there are request paths variables substituting the request arcs ones, whereas the general structure of constraints remains quite the same. However, the two conditions of limited delivery time and maximum number of hops are implicit in the model formulation.

Again, let $G = (V, A)$ be a digraph with arcs set A and nodes set $V = \{C \cup H\}$, comprising both the set of customer C the set of hubs H. The intra-hubs vehicles have a given transportation capacity K, and the number of hubs to open is n_{H}. Each request $r = (k_1, k_2) \in R \subseteq \{C \times C\}$ contains d^r demand units, and is characterized by the two sets of allowed start-hubs $H^{+^r} = H^+_{k_1}$ and allowed end-hubs $H^{-^r} = H^-_{k_2}$. The arc set A is the union of two subsets: the transfer arcs $A_t = \{H \times H\}$ between any two hubs, with costs per kilometer and vehicle used c_t, and the access arcs $A_s = \{(k, i), (i, k) \ \forall\, k \in C,\ i \in H_k\}$, with costs per kilometer and demand unit transported c_s. Each arc $a \in A$ has length l_a in kilometers. The request maximal time in route is T^r, whereas the maximum number hops is defined by the value n_{HOPS}.

For this alternative formulation, we define a feasible combined transport path for a request $r = (k_1, k_2)$ as a simple path from the origin k_1 to the destination k_2, passing at most through n_{HOPS} hubs and at least through one – and in the specific eventuality of only one hub, it is both an allowed start-hub and an allowed end-hub for the request $i \in \{H^{+^r} \cap H^{-^r}\}$. Further, the set of feasible paths for the request r is represented by $P^r = \{(a_1, \ldots, a_n) \mid a_1 \in \left\{k_1 \times H^+_{k_1}\right\},\ a_n \in \left\{H^-_{k_2} \times k_2\right\},$ $a_j \in A_t \ \forall\, j \in \{2, \ldots, n-1\},\ 2 \leq n \leq n_{HOPS} + 1\}$. All these feasible paths have a total time in route lower than (equal to) the maximal allowed T^r.

Additionally, let P_i and P_a be the two sets that comprise the feasible paths of all the requests which pass through hub i and arc a respectively. From these, we characterize: $P^r_i = P_i \cap P^r$ as the set of feasible paths of request r containing the hub i, and $P^r_a = P_a \cap P^r$ as the set of feasible paths of request r containing the arc a.

The extended model contains three types of decision variables: binary variables h_i indicating if the hub $i \in H$ is opened. General integer variables v_a addressing the number of vehicles that use the intra-hubs arc $a \in A_t$. Continuous variables $y^r_p \in [0, 1]$ determine the fraction of the demand units of request $r \in R$ which are transported via path $p \in P^r$.

Given these definitions, our path-based SNDHLP can be modeled as follows:

$$
\begin{aligned}
\min \quad & \sum_{a \in A_t} c_t l_a v_a \quad + \quad \sum_{r \in R} \sum_{p \in P^r} \sum_{a \in p \cap A_s} c_s l_a d^r y_p^r && && (3.33)\\
\text{s.t.} \quad & \sum_{i \in H} h_i \;=\; n_{\mathrm{H}} && && (3.34)\\
& \sum_{p \in P^r} y_p^r \;=\; 1 && \forall\, r \in R && (3.35)\\
& \sum_{r \in R} \sum_{p \in P_a^r} d^r y_p^r \;\leq\; K v_a && \forall\, a \in A_t && (3.36)\\
& \sum_{p \in P_i^r} y_p^r \;\leq\; h_i && \forall\, i \in H\,,\; r \in R && (3.37)\\
& h_i \;\in\; \{0,1\} && \forall\, i \in H && (3.38)\\
& v_a \;\in\; \mathbb{N}_0 && \forall\, a \in A_t && (3.39)\\
& y_p^r \;\in\; [0,1] && \forall\, r \in R\,,\; p \in P^r && (3.40)
\end{aligned}
$$

Again, this is the splittable requests' case, where the objective function (3.33) aims to minimize the total costs incurred by the demand units' transportation over access arcs and by operating vehicles on the intra-hubs links. Constraint (3.34) makes sure that the required number of hubs is opened. By condition (3.35) is guaranteed that every demand unit of a request is transported. Constraint (3.36) computes the used capacity and determines the number of necessary vehicles per intra-hubs arc. Condition (3.37) forces the opening of a hub if a request path passes through it. Finally, the last three constraints (3.38), (3.39) and (3.40) define the variables' domain, and by changing the domain of (3.40) to the binary one we target the case of unsplittable requests.

3.6 Comparison between the two model formulations

As we have seen in Section 3.4, the arc-based model of SNDHLP is a polynomial-sized model whose resolution does not require the implementation of pricing procedures. On the other hand, in Section 3.5 is pointed out that the path-based SNDHLP is a non-polynomial-sized model – except in case of a limited number of locations. The latter can be seen as an extended formulation of the arc-based one obtained from a Dantzig-Wolfe decomposition which reformulates the various combination of the arcs variables constituting a route for a specific request – in accordance with the flow conservation condition – into paths variables.
The main difference between the two models is in the number of variables, that is the reason of the relative polynomial and non-polynomial sizes of the two approaches:

Arc-based SNDHLP The number of the variables characterizing the compact model is of the order of $\mathcal{O}(n^4)$, where n is the number of locations. Indeed, if every customer location is the origin of a request going to each other customer location, we deal with a total number of requests equal to $n*(n-1)$. Then, if we assume that all the locations represent also potential hubs, in the "worst" case of a fully interconnected hubs network, there will be $n*(n-1)$ transfer arcs. Moreover, we have a certain number of access arcs equal to m, depending on how many allowed hubs each customer has.
Thus, we have a total number of arcs equal to $n*(n-1)+m$ for each request. As a consequence, the total number of request arcs variables is $n^2*((n-1)^2+m) \lesssim n^4$. This implies the polynomial size of the arc-based model for realistic instances, towards the explicit enumeration of all the arcs variables x_a^r.

Path-based SNDHLP The number of variables of the extended formulation is of the order of $\mathcal{O}(n^n)$. We take into account both the two previous description's hypotheses of all n locations as potential hubs and fully interconnected hubs network. The number of paths is given by $(m_{k_1}+m_{k_2})*\binom{n}{n_{\text{HOPS}}}$ for a single request – where m_{k_1} and m_{k_2} are respectively the number of allowed start-hubs for the customer origin location and of allowed end-hubs for the customer destination location, both lower than n.
Hence, we have $n*(n-1)*\underbrace{(m_{k_1}+m_{k_2})}_{\lesssim n}*\underbrace{\binom{n}{n_{\text{HOPS}}}}_{\lesssim n^{n_{HOPS}}} \lesssim n^{(3+n_{\text{HOPS}})}$ as total number of requests paths variables. This means that the explicit paths' enumeration resolution is tractable in polynomial time only if n is a very small

number. However, this does not mirror a real-world situation of the organization of freight transportation, and makes it necessary the adoption of different solution approaches for models with larger number of locations.

In conclusion, apart from minor changes in the model construction due to the inner differences between arcs and paths variables, the main distinction in the two formulations is in the final number of the treated variables for each request.
This number is hugely exponential in the path-based formulation, as the number of request paths is bounded from above by $2 * n * \binom{n}{n_{\text{HOPS}}}$, and corresponds to a model prohibitively solvable by explicit enumeration methods. For this reason, the path-based SNDHLP represents the principal object of interest of our study, in order to implement and test the effectiveness of distinct solution approaches.
Opposite, the number of variables is very reduced in the arc-based case: we deal with a number of arcs for each request bounded from above by $2 * n^2$, in case of fully interconnected graph. This enables the use of the arc-based SNDHLP as a benchmark for the optimality or close-to-optimality resolution. Furthermore, without imposing the two constraints of limited transport time and maximum number of transshipments, it is also possible to easily solve a relaxation of the compact model, which tries to explore longer routes too.

Chapter 4

Solution Approaches

In this chapter we delineate the diverse solution approaches used to search for an optimal solution to our SNDHLP. We start presenting in Section 4.1 the specific column generation applied through the Branch-and-Price and its features. The following Section 4.2 is dedicated to the different heuristic techniques applied.

4.1 A Branch-and-Price approach

This section shows the column generation algorithm applied for solving the path-based model. Firstly, we introduce the master problem. Then, we present the auxiliary problem used to obtain a feasible starting solution for the initialization of the restricted master problem. Finally, we describe our pricing problem and how the Branch-and-Price process works.

4.1.1 Master Problem

In the final part of Section 3.6, we underline the non-tractability, for a realistic number of locations, of the path-based SNDHLP by a generic MILP solver. Indeed, the path-based model represents our master problem on which applying the column generation algorithm. We recall that it is formulated as follows:

$$
\begin{array}{lrcll}
z^\star_{\mathrm{MP}} := \min & \displaystyle\sum_{a\,\in\,A_t} c_t l_a v_a & + & \displaystyle\sum_{r\,\in\,R}\sum_{p\,\in\,P^r}\sum_{a\,\in p\,\cap\,A_s} c_s l_a d^r y^r_p & \\
\text{s.t.} & \displaystyle\sum_{i\,\in\,H} h_i & = & n_{\mathrm{H}} & \\
& \displaystyle\sum_{p\,\in\,P^r} y^r_p & = & 1 & \forall\, r \in R \\
& \displaystyle\sum_{r\,\in\,R}\sum_{p\,\in\,P^r_a} d^r y^r_p & \leq & K v_a & \forall\, a \in A_t \\
& \displaystyle\sum_{p\,\in\,P^r_i} y^r_p & \leq & h_i & \forall\, i \in H\,,\ r \in R \\
& h_i & \in & \{0,1\} & \forall\, i \in H \\
& v_a & \in & \mathbb{N}_0 & \forall\, a \in A_t \\
& y^r_p & \in & [0,1] & \forall\, r \in R\,,\ p \in P^r
\end{array}
\tag{4.1}
$$

In particular, the set of variables which makes not possible the explicit enumeration resolution is P^r, because it contains an exponential number of paths for every single request.
As explained in Section 2.2.1, in order to apply a column generation algorithm to a MILP problem, it is necessary to define a restriction on the master problem variables, as in an optimal solution the majority of them will be in the non-basis and have value 0. Hence, it naturally follows that the restriction must be applied on the set of the request paths variables P^r.
The first crucial assumption we make is that we never want the CG algorithm comes across an infeasible solution during the solving process. This implies that the restricted master problem needs to have a starting feasible solution. In fact, this will consequently ensure that in every iteration of the column generation there will be at least one feasible optimal solution – the starting one.

4.1.2 Auxiliary Problem

In order to obtain an always feasible restricted master problem, we need to define an auxiliary problem. Undoubtedly, to get this feasible starting solution, we require a restriction on the set of the request paths variables which does not exclude the necessary ones for that solution.
The unique way to obtain for each request r this feasible restricted set $P^{r\prime} \subset P^r$

is to solve an auxiliary integer linear problem which, first of all, guarantees the feasibility of the SNDHLP, but also provides a feasible set of open hubs $H' \subset H$ – if it exists. The only infeasibility condition of our SNDHLP is associated with the problem's p-median constraint: a too small number of hubs to be opened n_{H} might not accomplish the service of all the requests. Hence, if this number is not sufficient to open at least one allowed hub for each customer location, the SNDHLP is infeasible. We remark that this case of infeasibility is a direct consequence of our obligation of combined transport for each request.

Actually, the auxiliary problem looks for a combination of hubs to open which ensures that every customer has at least one of its allowed hub opened, because in the worst possible situation all the requests starting from or arriving to that customer location will have only one hub as allowed start-hub or end-hub – if a combination exists.

The auxiliary problem is formulated as follows:

$$\begin{aligned}
\min \quad & 0 \\
\text{s.t.} \quad & \sum_{i \in H} h_i = n_{\mathrm{H}} && (4.2)\\
& \sum_{i \in H_k} h_i \geq 1 \qquad \forall\, k \in C && (4.3)\\
& h_i \in \{0,1\} \qquad \forall\, i \in H
\end{aligned}$$

As it is a feasibility-verification problem, the auxiliary problem does not need any objective function. The linear equality (4.2) is the same constraint of the SNDHLP model to open a given number of hubs. By condition (4.3) we guarantee the feasibility of the SNDHLP – and obviously of this auxiliary problem too – imposing the opening of an allowed hub for each customer.

The result of the auxiliary problem is one of the possible open hubs' combination – if at least one exists – which constitutes the new set H'.

4.1.3 Restricted Master Problem

Once a possible combination of open hubs H' is obtained from the auxiliary problem, we have all the necessary tools to define a restriction on the master variables and initialize the restricted master problem.

The restricted set $P^{r\prime}$ definition for each request $r = (k_1, k_2) \in R$ is carried out by removing all the hub nodes not present in H' from the specific request digraph, and then selecting the five cheapest feasible simple paths from k_1 to k_2 of maximum length $n_{\mathrm{HOPS}} + 1$ and maximum time in route T^r. Specifically, the total cost of each path is computed as expressed by the next equation:

$$TC_p = \sum_{a \in p \cap A_s} c_s l_a + \sum_{a \in p \cap A_t} \frac{c_t}{K} l_a \tag{4.4}$$

It is important to underline that this cost represents in any case an approximated estimation of the real final cost a path might have, for two reasons: the first is that the cost of transfer arcs does not consider the number of vehicles passing through that arc, but simply we divide everything for their capacity K. Further, in case of splittable requests, the whole cost eventually represents only a percentage of the final cost, as we do not know how many demand units will be routed on this path and if it will be used or not.
After having defined all the restricted request paths sets for every request, also the two sets P_i^r and P_a^r – comprising the request paths which contain the specific hub i and arc a – will be restricted as they are the result of an intersection with P^r.
Then, we can formalize the restricted master problem:

$$
\begin{aligned}
z_{\mathrm{RMP}} := \min \quad & \sum_{a \in A_t} c_t l_a v_a \quad + \quad \sum_{r \in R} \sum_{p \in P^{r\prime}} \sum_{a \in p \cap A_s} c_s l_a d^r y_p^r \\
\text{s.t.} \quad & \sum_{i \in H} h_i \; = \; n_{\mathrm{H}} \\
& \sum_{p \in P^{r\prime}} y_p^r \; = \; 1 && \forall\, r \in R \\
& \sum_{r \in R} \sum_{p \in P_a^{r\prime}} d^r y_p^r \; \leq \; K v_a && \forall\, a \in A_t \\
& \sum_{p \in P_i^{r\prime}} y_p^r \; \leq \; h_i && \forall\, i \in H\,,\; r \in R \\
& h_i \in \{0,1\} && \forall\, i \in H \\
& v_a \in \mathbb{N}_0 && \forall\, a \in A_t \\
& y_p^r \in [0,1] && \forall\, r \in R,\; p \in P^{r\prime}
\end{aligned}
\tag{4.5}
$$

As it is clearly visible from the above model, the difference with the master problem (4.1) is in the set characterizing the constraints of full request delivery, number of transfer vehicles, and obliged opening of a hub when a path passes through it.

4.1.4 Pricing Problem

Once the restricted master problem (4.5) has been formulated, we can certainly solve it with the classical enumeration method of all the paths, as it has a very small number of variables. However, the optimal solution of the RMP z_{RMP} has a very low probability to be an optimal solution of the master problem too. Rather, as we have seen in the inequality (2.5), this value represents surely an upper bound for the final optimal solution $z^\star_{\mathrm{MP}}$. In order to reduce this upper bound, we need to define the pricing problem useful for adding the missing columns to the restricted problem and solving its continuous relaxation.
The simplest way to generate new columns associated with request paths variables

is to solve a classical shortest past problem, intensely studied in the Operations Research literature. Although, the pricing problem cannot be only a shortest path problem, because it must take into account also the two constraints of maximum travel time and maximum number of hops. Therefore, our pricing problem is a resources-constrained shortest path problem which, for each request r, seeks for the path with the most negative reduced cost.
To compute the reduced cost, we need the dual values referred to the primal constraints:

- β_r is the dual variable associated with the full delivery of the request r demand units constraint (3.35) of the path-based SNDHLP.
- γ_a is the dual referred to the master constraint (3.36) computing the number of necessary vehicles per each transfer arc a.
- σ_i^r is the dual variable linked to the path-based condition (3.37) that opens a hub i if a path of the request r pass through it.

Both γ_a and σ_i^r are negative, as they are associated with lower or equal inequalities, because from the dual problem's perspective it is not convenient to open an extra hub or use an extra vehicle, whereas β_r is free – because referred to a linear equality – but generally positive, as it is better to over-serve a request rather than not. Hence, the reduced cost of each path is given by:

$$c_p = \sum_{a \in p \cap A_s} c_s d^r l_a - \sum_{a \in p \cap A_t} d^r \gamma_a - \sum_{i \in p \cap H} \sigma_i^r - \beta_r \tag{4.6}$$

Finally, we can formalize the mathematical model of the pricing problem PP_r for each request $r = (k_1, k_2)$:

$$\min \quad \sum_{a \in p \cap A_s} c_s d^r l_a - \sum_{a \in p \cap A_t} d^r \gamma_a - \sum_{i \in p \cap H} \sigma_i^r - \beta_r \tag{4.7}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(v)} x_a^r - \sum_{a \in \delta^-(v)} x_a^r = \begin{cases} 1 & \text{if } v = k_1 \\ -1 & \text{if } v = k_2 \\ 0 & \text{otherwise} \end{cases} \quad \forall\, v \in V \tag{4.8}$$

$$\sum_{a \in A} x_a^r \leq n_{\text{HOPS}} + 1 \quad \forall\, r \in R \tag{4.9}$$

$$\sum_{a \in A_s} t_s l_a x_a + \sum_{a \in A_t} t_t l_a x_a \leq T^r \quad \forall\, r \in R \tag{4.10}$$

$$x_a \in \{0, 1\} \quad \forall\, a \in A$$

The objective function (4.7) looks for the minimum reduced cost of the path p. The condition (4.8) express the flow conservation to guarantee that the chosen arcs

constitute a path. The two inequalities (4.9) and (4.10) represent the maximum resources constraints.
The pricing problem is solved for each request at every iteration of the algorithm, until for all the requests there are no more variables having a negative reduced – which means we have found an optimal solution of the continuous relaxation of the restricted master problem.

4.1.5 Branching Rules

After having introduced the auxiliary problem and the pricing problem, we can now delineate how our column generation algorithm proceeds:

1. solve the auxiliary problem to find a combination of feasible open hubs $H' \subset H$
2. for each request r, remove all the hubs $i \in H \setminus H'$ from the request graph
3. generate, for each request r, all the simple paths derived from the request graph, and then define the restricted set of request paths' variables $P^{r\prime} \subset P^r$ by selecting the five paths with the cheapest approximate total cost
4. formulate the restricted master problem, through $P^{r\prime}$
5. start a loop:
    - solve the RMP, and obtain an upper bound z_{RMP} for the optimal solution
    - for each request r, define and solve its pricing problem
    - add in the RMP the column related to the found path if it has a negative reduced cost
    - if the minimum reduced cost of each request is non-negative, stop the loop. Otherwise, restart from solving the RMP

In the column generation process, there is no guarantee that all the paths of an optimal solution to the original problem are generated. Hence, in order to speed up the process and ensure its complete correctness, we implement a Branch-and-Price algorithm where we explicitly consider a branching rule for the pricing problem. In each pricing iteration, we look for the upper bounds of the variables, and we forbid the hubs and transfer arcs having a local upper bound lower than 1 in the relative branch node. This avoids the generation of paths which do not match the Branch-and-Bound decisions. In particular, we define the set of forbidden hubs $H^{\text{F}} = \{i \mid \text{UB}_i < 1\}$, and the set of forbidden transfer arcs ${A_t}^{\text{F}} = \{a \mid \text{UB}_a < 1\}$. Then, we remove the arcs and hubs comprised in these two sets from the request graph and, as a consequence, we guarantee that all the generated columns are in accordance with the branching decisions.

4.2 Heuristic methods

In this section we present a group of different solution approaches for the path-based SNDHLP, which might lead to a non-optimal solution, but have a very fast solving time. The heuristics implemented rely on the prioritization of the hubs variables based on a specific criterion, but without affecting the feasibility of the SNDHLP in any case – if the problem is originally feasible.
Besides, all these heuristic techniques could substitute the Auxiliary Problem seen in Section 4.1.2 to determine the starting set of open hubs for the RMP of the B&P algorithm. Indeed, they are all based on the Auxiliary Problem, but comprise an objective function necessary for the hubs prioritization and eventually some additional constraints useful to that objective function.

4.2.1 Most Accessed Hubs Heuristic

The first heuristic approach is named "Most Accessed Hubs Heuristic": it prioritizes the hubs according to the number of ingoing access arcs they have – which is equal to the number of the outgoing ones, according to the model construction. The motivation is that if a hub has a greater number of access arcs, probably has a greater possibility of being present in an optimal solution, because it serves more customer locations and so is more important.
Firstly, we can formalize the Most Accessed Hubs Heuristic auxiliary problem in mathematical terms:

$$\max \quad \sum_{i \in H} {n_i}^{\text{numACCESS}} h_i \tag{4.11}$$

$$\text{s.t.} \quad \sum_{i \in H} h_i = n_{\text{H}} \tag{4.12}$$

$$\sum_{i \in H_k} h_i \geq 1 \qquad \forall\, k \in C \tag{4.13}$$

$${n_i}^{\text{numACCESS}} = \sum_{a \in \delta^-(i)} x_a \qquad \forall\, a \in A_s\,,\ \ i \in H \tag{4.14}$$

$$x_a = 1 \qquad \forall\, a \in A_s \tag{4.15}$$

$$h_i \in \{0,1\} \qquad \forall\, i \in H$$

The objective function (4.11) seeks to maximize the resulting sum of the opened hubs. This sum is a direct consequence of the equality (4.14) which imposes the multiplicative factor of each hub ${n_i}^{\text{numACCESS}}$ equal to the sum of its ingoing arcs, that are all assumed to be 1 from constraint (4.15). The conditions (4.12) and (4.13) are the same of the auxiliary problem of the column generation algorithm that ensure the opening of n_{H} hubs and the feasibility of the SNDHLP by imposing that each customer location has at least one allowed hub open.
The solution of the previous problem is a set of open hubs $H^{\text{MostAccessed}} \subset H$. Hence, in order to implement the heuristic resolution of the path-based SNDHLP, we remove from the original hubs set H all the hubs that are not included in $H^{\text{MostAccessed}}$ or, in other words, we operate the substitution of the original H with the new set $H^{\text{MostAccessed}}$, and we define the corresponding new hubs graph. Then, for each request we generate all the feasible paths from the new request graph, and we obtain a new path-based model. The latter can be considered a compact formulation because the number of paths variables is limited by the heuristic auxiliary problem solution. On the other hand, the drawback is that we have no warranty that this heuristic solution is an optimal one, or how much it is far from the optimality.

4.2.2 Greatest Demand Requests Heuristic

The next heuristic approach studied is named "Greatest Demand Requests Heuristic". As its name suggests, this heuristic method initially sorts the requests by their ascending number of demand units. Then, the number associated with the hub priority is equal to the ranking of the greatest request for which the hub is an allowed start-hub or end-hub. Alternatively from the first heuristic approach, this is an attempt to verify if there might be a correlation between the dimension of the requests and the importance of the hubs in the optimal routing solution, i.e. try to open first the hubs linked to most demanding customer locations.
The Greatest Demand Requests Heuristic auxiliary problem is mathematically expressed by:

$$\max \quad \sum_{i \in H} {n_i}^{\text{maxRANK}} h_i \tag{4.16}$$

$$\text{s.t.} \quad \sum_{i \in H} h_i \;=\; n_{\text{H}} \tag{4.17}$$

$$\sum_{i \in H_k} h_i \;\geq\; 1 \qquad \forall\, k \in C \tag{4.18}$$

$${n_i}^{\text{maxRANK}} = \max \Big\{ j \;\mid\; i \in \Big\{ H^{+r} \cup H^{-r} \Big\},\; r_j \in R_{\text{S}} \Big\} \quad \forall\, i \in H \tag{4.19}$$

$$R_{\text{S}} \;=\; \Big\{ (r_1, \ldots, r_{n*(n-1)}) \;\mid\; a \leq b \;\Leftrightarrow\; d^{r_a} \leq d^{r_b} \Big\} \tag{4.20}$$

$$h_i \;\in\; \{0, 1\} \qquad \forall\, i \in H$$

The objective function (4.16) addresses the maximization of the resulting sum of the opened hubs. This sum is a direct consequence of the equality (4.19) which defines the hub multiplicative factor ${n_i}^{\text{maxRANK}}$ as the last position of the pre-prioritized request R_{S} for which this is an allowed start-hub or end-hub. The requests are sorted ascending by their demand units dimension – as expressed in condition (4.20). The constraints (4.17) and (4.18) are the same of the auxiliary problem of the column generation algorithm that ensure the opening of n_{H} hubs and the feasibility of the SNDHLP by imposing that each customer location has at least one allowed hub open.
The solution of the previous problem is a set of open hubs $H^{\text{GreatestDemand}} \subset H$. Thus, in order to implement the heuristic solving of the path-based of SNDHLP, we remove from the original hubs set H all the hubs that are not comprised in $H^{\text{GreatestDemand}}$ or, in other words, we substitute the original H with the new set $H^{\text{GreatestDemand}}$, and we define the corresponding new hubs graph. Then, for each request we generate all the feasible paths from the new request graph, and we obtain a new path-based model. The latter can be considered a compact formulation because the number of paths variables is limited by the heuristic auxiliary problem solution.

4.2.3 Additive Greatest Demand Requests Heuristic

The third heuristic method is named "Additive Greatest Demand Requests Heuristic". Indeed, it is similar to the previous method presented in Section 4.2.2, and again, it initially sorts the requests by their ascending number of demand units. But, the hub priority is represented by the sum of all the rankings corresponding to the requests for which the hub is an allowed start-hub or end-hub. This heuristic can be seen as an alternative approach to the second one to verify if there might be a correlation between the dimension of the requests and the importance of the hubs in an optimal routing solution. However, distinctly from the second approach which wants to open first the hubs connected to the most demanding customer locations, this tries to consider all the requests assigning an overall score to the hub.
The formulation of the Additive Greatest Demand Requests Heuristic auxiliary problem is:

$$
\begin{aligned}
\max \quad & \sum_{i \in H} {n_i}^{\text{sumRANK}} h_i && && (4.21)\\
\text{s.t.} \quad & \sum_{i \in H} h_i = n_{\text{H}} && && (4.22)\\
& \sum_{i \in H_k} h_i \geq 1 && \forall\, k \in C && (4.23)\\
& {n_i}^{\text{sumRANK}} = \sum_{r \in R} n_i^r && \forall\, i \in H && (4.24)\\
& n_i^r = \begin{cases} \{j \mid r_j \in R_{\text{S}}\} & \text{if } i \in \{H^{+r} \cup H^{-r}\} \\ 0 & \text{otherwise} \end{cases} && \forall\, r \in R,\ i \in H && (4.25)\\
& R_{\text{S}} = \Big\{(r_1, \ldots, r_{n*(n-1)}) \mid a \leq b \Leftrightarrow d^{r_a} \leq d^{r_b}\Big\} && && (4.26)\\
& h_i \in \{0,1\} && \forall\, i \in H &&
\end{aligned}
$$

The objective function (4.21) aims to maximize the resulting sum of the opened hubs. This sum is a direct consequence of the equality (4.24) which computes the multiplicative factor of each hub ${n_i}^{\text{sumRANKING}}$ as the sum of the factors n_i^r associated with the pre-sorted requests R_{S} by their ascending demand units dimension – as expressed in condition (4.26). Specifically, the requests' factors are retrieved from the condition (4.25): if the hub is an allowed start-hub or end-hub for the request, this factor is equal to the position of the request in the sorted set $n_i^r = j$, otherwise it is 0. The constraints (4.22) and (4.23) are the same of the auxiliary problem of the column generation algorithm that ensure the opening of n_{H} hubs and the feasibility of the SNDHLP by imposing that each customer location has at least one allowed hub open.

The solution of the previous problem is a set of open hubs $H^{\text{AdditiveDemands}} \subset H$. Hence, in order to implement the heuristic resolution of the path-based SNDHLP, we remove from the original hubs set H all the hubs that are not comprised in $H^{\text{AdditiveDemands}}$ or, in other words, we operate the substitution of the original H with the new set $H^{\text{AdditiveDemands}}$, and we define the corresponding new hubs graph. Then, for each request we generate all the feasible paths from the new request graph, and we obtain a new path-based model.

4.2.4 Shortest Access Arcs Heuristic

The last heuristic resolution algorithm is named "Shortest Access Arcs Heuristic". This method gives more priority to the hubs according to the average length of their outgoing access arcs – the latter value is equal to the average length of the ingoing access arcs, for how the SNDHLP instances are built. The rationale of the heuristic is to investigate an eventual correlation between the choice of opening a hub in an optimal solution and its average distance from customer locations.
The auxiliary problem of the Shortest Access Arcs Heuristic is formulated as follows:

$$\min \quad \sum_{i \in H} {n_i}^{\text{avgDIST}} h_i \tag{4.27}$$

$$\text{s.t.} \quad \sum_{i \in H} h_i = n_{\mathrm{H}} \tag{4.28}$$

$$\sum_{i \in H_k} h_i \geq 1 \qquad \forall\, k \in C \tag{4.29}$$

$${n_i}^{\text{avgDIST}} = \begin{cases} \frac{{n_i}^{\text{totalDIST}}}{{n_i}^{\text{numACCESS}}} & \text{if } {n_i}^{\text{numACCESS}} > 0 \\ M & \text{otherwise} \end{cases} \qquad \forall\, i \in H \tag{4.30}$$

$${n_i}^{\text{totalDIST}} = \sum_{a \in \delta^+(i)} l_a \qquad \forall\, a \in A_s\,, \quad i \in H \tag{4.31}$$

$${n_i}^{\text{numACCESS}} = \sum_{a \in \delta^+(i)} x_a \qquad \forall\, a \in A_s\,, \quad i \in H \tag{4.32}$$

$$x_a = 1 \qquad \forall\, a \in A_s \tag{4.33}$$

$$h_i \in \{0, 1\} \qquad \forall\, i \in H$$

The objective function (4.27) seeks to minimize the resulting negative sum of the opened hubs. This sum is a direct consequence of the equality (4.30) which derives the multiplicative factor of each hub ${n_i}^{\text{avgDIST}}$ as the ratio between the total distance of all the outgoing access arcs of the hub ${n_i}^{\text{totalDIST}}$ and the relative number ${n_i}^{\text{numACCESS}}$. But, if the latter value is 0, the average distance is imposed to be M, where M is a very big integer number. The two factors are obtained respectively from the conditions (4.31) and (4.32), thanks also to the assumption that all the access arcs are equal to 1 of the constraint (4.33). The constraints (4.28) and (4.29) are the same of the auxiliary problem of the column generation algorithm that ensure the opening of n_{H} hubs and the feasibility of the SNDHLP by imposing that each customer location has at least one allowed hub open.
The solution of the previous problem is a set of open hubs $H^{\text{ShortestAccess}} \subset H$. Hence, in order to implement the heuristic resolution of the path-based SNDHLP, we remove from the original hubs set H all the hubs that are not included in $H^{\text{ShortestAccess}}$ or, in other words, we operate the substitution of the original H with the new set $H^{\text{ShortestAccess}}$, and we define the corresponding new hubs graph.

4.2.5 A Matheuristic approach

The last solution approach adopted is a matheuristic, which consists in the perturbation of the path-based solution obtained from a heuristic method. The goal is to try to improve this current optimal value through a local search method, by imposing new constraints derived from the previous heuristic solution.
Actually, we do not want to distort too much the heuristic value. Hence, given the heuristic solution $z^{\star}_{\text{HEUR}}$ and the values of its variables of hubs $h_{i\text{HEUR}} \in H^{\text{heuristicApproach}}$ and transfer arcs' vehicles $v_{a\text{HEUR}}$, the two new constraints for the path-based model are:

- Imposition of remaining open hubs as a percentage of the heuristic set of open hubs: given the latter set, we establish that the 75% of them (rounded down to the lower integer) must remain open in the new perturbed solution.
- Limitation in the perturbed solution of the number of heuristic transfer arcs' vehicles to the 150% of their current heuristic value. But only if the latter value is greater than 0, because otherwise we would implicitly impose an upper bound equal to 0 on the vehicles of that transfer arcs, excluding consequently all the possible request paths containing it.

Thus, the formulation of the matheuristic approach to perturb the path-based model is the following:

$$
\begin{array}{llll}
\min & \displaystyle\sum_{a\in A_t} c_t l_a v_a + \sum_{r\in R}\sum_{p\in P^r}\sum_{a\in p\cap A_s} c_s l_a d^r y_p^r & & \\
\text{s.t.} & \displaystyle\sum_{i\in H} h_i = n_{\text{H}} & & \\
& \displaystyle\sum_{p\in P^r} y_p^r = 1 & \forall\, r\in R & \\
& \displaystyle\sum_{r\in R}\sum_{p\in P_a^r} d^r y_p^r \le K v_a & \forall\, a\in A_t & \\
& \displaystyle\sum_{p\in P_i^r} y_p^r \le h_i & \forall\, i\in H\,,\ r\in R & \\
& \displaystyle\sum_{i\in H^{\text{heuristicApproach}}} h_i \ge \lfloor 0.75 * n_{\text{H}}\rfloor & & (4.34)\\
& v_a \le \lfloor 1.5 * v_{a\text{HEUR}}\rfloor & \forall\, a\in A_t \quad \text{if } v_{a\text{HEUR}} > 0 & (4.35)\\
& h_i \in \{0,1\} & \forall\, i\in H & \\
& v_a \in \mathbb{N}_0 & \forall\, a\in A_t & \\
& y_p^r \in [0,1] & \forall\, r\in R\,,\ p\in P^r &
\end{array}
$$

As it is easily understandable, in this matheuristic path-based model, the only new conditions are (4.34) – which enforces the opening of the 75% of the heuristic hubs

– and the (4.35), which expresses the maximal number of vehicles for the specific transfer arc whose value was already greater than 0.

This matheuristic model is solved through the Branch-and-Price algorithm. In particular, it is eventually possible to perturb a previous SNDHLP model multiple times. This means that we can create a new matheuristic model as this, also after having already perturbed the heuristic solution the first time – or for a certain number of times. Specifically, each matheuristic perturbation takes as input a SNDHLP model already solved and its solution, imposes the two new constraints and applies the Branch-and-Price algorithm. Indeed, these two new constraints will have an influence not only on the problem final objective value, but also on the dual values used by the B&P.

In the end, by solving this, single or multiple, perturbed problem we find a new optimal value $z^{\star}_{PERTURBED}$ which hopefully improves the starting heuristic solution, namely $z^{\star}_{PERTURBED} < z^{\star}_{HEUR}$.

Chapter 5

Computational Results

This chapter presents the results of the computational experiments performed on various instances. In Section 5.1 we introduce the solver and the environments used to test the instances. The details about instances: datasets of origin, generation of problem parameters, and instances of major interest are explained in Section 5.2. Then, next Sections are reserved to the exposition of the organization of our experiments and the related results.

5.1 Introduction on the solver environments

The solution algorithms have been implemented in Python and principally used the *PySCIPopt* library for the model generation and the solution tools of SNDHLP instances. This library refers to one of the fastest non-commercial solvers for MILPs: SCIP. The latter is a framework for constraint integer programming and Branch-and-Price. Additionally, other two relevant Python libraries for the solution of our problem are *networkx*, which presents useful tools for the graphs' management, and *cspy*, that implements a method for solving a resource constrained shortest path problem.
The first experiments were performed on a standard PC with an Intel® Core AMD A9-9410 CPU at 2,9 GHz and 4 GB of RAM. The main testing phase was run on the cluster of the RWTH Aachen University's Operations Research Department, which comprises a total of 56 machines with 16 GB of RAM and 8 machines with 128 GB of RAM, all of them equipped with Quad Core Processor Intel® Xeon® L5630 CPU at 2.13GHz.

5.2 Presentation of the problem instances

5.2.1 Instances datasets

To assess the effectiveness of various methods for solving our SNDHLP, three popular datasets are utilized as examples. One such dataset, retrieved from Krishnamoorthy et al. (2000), is the AP (Australia Post) that contains 200 postcode districts in Australia, along with their locations and pairwise travel demands. The studies in Hub Location and Network Design from M.E. O'Kelly et al. (2023) provide us the CAB (Civil Aeronautics Board) dataset, which includes 100 nodes representing passenger interactions between cities in the United States. Finally, the TR (Turkish Postal) dataset consists of 81 cities within the Turkish postal system and includes pairwise distances and travel demands (see Çetiner et al. (2010)). To create smaller datasets, we simply select the first n nodes from the CAB, TR, and AP datasets, generating instances from 10 to 50 locations with a step of 5 nodes. Then, for the specific experiments other different cut and selection methods have been applied on the three datasets to generate a greater number of instances to test.

5.2.2 Real-world instances

In order to understand the sizes of the instances to be tested, we made some researches on real-world applications of the general service network design problems and hub location problems. We concluded that there is not a universally recognized realistic size for these problems.
Then, we decided to contextualize our study to the place where the thesis problem has been studied for the majority of the time: RWTH Aachen University, and so the Nord-WestFalen region of Germany. We analyzed some transport applications, and we took into account the network of links present on the Arriva DB company for public transportation website. As we can see from Figure 5.1, there are exactly 35 different stations distributed over the Nord-WestFalen region, some of them closer than others, but which can be considered a real-world application for our SNDHLP. Hence, the most relevant computational experiments are the ones performed on the 35-locations instances. Besides, we consider numerous testing instances also in the neighborhood of 35: smaller instances of 25 and 30 locations, and larger cases of 40 locations with the perspective of a possible future expansion of the system.

Figure 5.1: Map of Station Locations in the Nord-WestFalen region

5.2.3 Setup of instances parameters

All our experiments have been conducted with a specific model construction. First of all, we consider fully interconnected internal networks of hubs – which means $n * (n-1)$ arcs totally – and the situation where all the locations have both the role of customers and of hubs. Besides, every customer location is both the origin and the destination for all the requests starting/arriving there – total of $n * (n-1)$ requests.
Then, we decided that each allowed hub of a customer has to be both an allowed start-hub and an allowed end-hub. Among the allowed hubs of each customer location there is the same hub location. Apart from that, each customer has a number of allowed start-hubs – and so of allowed end-hubs – equal to m, where m is the integer part of the value of a certain multiplicative factor times the total number of locations. Specifically, we set this multiplicative factor to 0.3 if the number of locations is lower or equal to 20, to 0.25 if lower than 40 and 0.2 otherwise. Then, we select the m closest locations to the specific customer, and we put the hubs of those locations in the allowed start-hubs and allowed end-hubs sets of the customer.

Additionally, in our experiments it was necessary to set appropriate problem parameters, in order to have reasonable solutions. The parameters to be set are: number of hubs to open, maximum number of transshipments, maximum transport time for a request, capacity of transfer vehicles, relationships between intra-hubs arcs costs and access arcs costs. Indeed, we want to avoid uncommon solution structures if these are not set properly. For this reason, we conduced many preliminary tests, changing these values, to figure out the best combination possible.
In the end, we set these values taking into account for every instance its average demand of all the requests (*avgDemand*), its average distance among all the locations pairwise (*avgDistance*), and its number of customers locations n:

Number of Hubs n_{H} we set this fixed value of hubs to open equal to the integer part of the function $n^{0.6}$, that guarantees a gradual growth of the open hubs with the greater size of instances.

Number of Transshipments n_{HOPS} this value is related to the number of hubs:

$$n_{\mathrm{HOPS}} = \begin{cases} 3 & \text{if } n_H \leq 6 \\ 4 & \text{if } 7 \leq n_H \leq 10 \\ 5 & \text{otherwise} \end{cases}$$

This guarantees a reasonable number of hops in any case, and in particular is limited to 3 only for small instances, whereas is equal to 4 in the realistic and semi-realistic instances.

Request Maximum Transport Time T^r this was the most critical value to be set, because we do not know the optimal paths a priori. Hence, we set it as

$5 * avgDistance + 1.5 * distanceOD$, where $distanceOD$ is the direct distance between a customer origin location and a customer destination location. This equality should consider the general structure of the hubs network and not forbid promising paths just for a matter of time. In addition, it is important to underline that we consider an average speed of $80km/h$ over access links to internal network of hubs and of $120km/h$ in transfer links.

Transfer Vehicles Capacity K this is the value that actually determines the final number of vehicles and the distribution they have in terms of overwhelmed intra-hubs arcs or well-distributed vehicles. The best preliminary results where the ones where we fix this to $5 * avgDemand$.

Relationship between Costs of Access c_s **and Intra-Hubs Arcs** c_t This relies on the hypotheses done on the speed over arcs and on the vehicles' capacity too. Thus, we impose that the access arcs cost must be equal to twice the ratio between the cost of transfer arcs and the transfer vehicles capacities $c_s = 2 * \frac{c_t}{K}$.

5.3 Organization of the Experiments

The computational experiments have been organized in the following way: after some first generic tests of the different solution approaches to set the appropriate values for the problem parameters, we decided to select the best resolution combination to fast the process and obtain better results as close to optimality as possible.
In order to obtain this combination, the experiments have been divided in five initial phases and a final comparison one:

Heuristic Experiments to find which is the best heuristic approach proposed, we solve the complete path-based model having as open hubs the ones obtained from the feasible solutions of the four different heuristic methods presented in Sections 4.2.1-4.2.4 and of the auxiliary problem presented in Section 4.1.2. The goal is to figure out which of the heuristics gives the best objective value.

Matheuristic Experiments to find the best trade-off between the local branching improvements and the number of perturbations of the heuristic solutions. Hence, from the two best heuristics delineated in the first testing phase, we apply the matheuristic shown in Section 4.2.5 with Branch-and-Price for 10 consecutive times or until no more improvements are found. The objective is to find an appropriate number of perturbations, as a compromise between the percentage of solution improvements over the different perturbations and the relative solving time.

Branch-and-Price Experiments to find the best and the fast possible combination of solving features which improve the column generation algorithm. Indeed, we test the Branch-and-Price algorithm presented in Section 4.1.5 in three different ways:

- Solving the restricted master problem with the set of hubs of the best heuristic method defined by the heuristic experiments, and then apply the B&P.
- Solving the RMP with the set of hubs of the best heuristic method and then perturbing it for a number of times equal to the one delineated in the matheuristic experiments. Then, this multiple-perturbed problem represents the starting RMP of the B&P algorithm.
- Solving the restricted master problem with the set of hubs of the auxiliary problem (4.1.2), but imposing the objective value of the RMP as primal bound for the Branch-and-Price algorithm, and then try to see if there is any improvement in terms of solving time or final solution value, compared to the normal B&P.

Hence, here the goal is to find which one of these three represent the best combination of features Branch-and-Price, or if it could seem reasonable to combine them too.

Early Branching Branch-and-Price Experiments to figure out if this additional tool can help to fast the solving process of the Branch-and-Price, taking into account the best combination of features obtained in the previous testing phase. In particular, the early branching B&P consists in the computation in each pricing iteration of the current B&B node's Lagrangian gap $LG = \frac{LB - z_{\text{RMP}}}{LB}$ – where LB is the node lower bound. When this value is lower than the fixed early-branching threshold of 0.05, the pricing iterations are stopped on that B&B node and the Branch-and-Price proceeds to the next node of the search tree.

Arc-based Experiments to grasp the efficacy of the SCIP open source solver compared to the commercial solver Gurobi in solving the polynomial-time arc-based model.

Final Comparison Experiments to compare the best results of the previous phases with the benchmark of the arc-based model, and then understand the effectiveness of our proposed solution approaches.

In particular, the first five experiments phases are performed on a limited number of instances, because they serve only as skimming for the sixth phase, and with a time limit of only one hour.

5.4 Preliminary Experiments

This section is dedicated to the first five phases of experiments: heuristics, matheuristics, Branch-and-Price with different features, and the arc-based SNDHLP solved with SCIP and Gurobi. They have been carried out all on forty-three different instances targeting only the case of splittable requests, and with a time limit of only 1 hour. In particular, the instances are taken in equal number from the three datasets: AP, CAB and TR, considering the ones from 10 to 50 locations. These represent a total of twenty-seven instances. In addition, other sixteen instances are obtained by cutting the 50-locations instance of the Turkish dataset in four different ways, each for retrieving a new instance of 20, 25, 30 or 35 locations. Hence, in the end, we tested three instances of 10, 15, 40, 45, and 50 locations, and seven instances of 20, 25, 30, and 35 locations.

5.4.1 Heuristic Experiments

The heuristic tests have been carried out considering the auxiliary problem and the four heuristic approaches presented in the previous chapter:

- Auxiliary problem – named "Auxiliary" in Table 5.1, which verifies the feasibility of the SNDHLP and eventually has as result a random feasible combination of open hubs (see Section 4.1.2).
- Most Accessed Hubs Heuristic – denoted as "MostAccHubs" in Table 5.1, that prioritizes the hubs on the basis of how many customers they serve (see Section 4.2.1).
- Greatest Demand Requests Heuristic – called "GrDemReq" in Table 5.1, which gives more relevance to the hubs connected with the customer having the greatest demand units to send or receive (see Section 4.2.2).
- Additive Greatest Demand Requests Heuristic – denoted as "AddGrDemReq" in Table 5.1, that chooses the hubs after having assigned them a multiplicative factor on the basis of the inverse demand-size ranking of each request served by them (see Section 4.2.3).
- Shortest Access Arcs Heuristic – called "ShAccArcs" in Table 5.1, which orders the hubs by the shortest average distance from the customers served (see Section 4.2.4).

heuristic	#locations	gap (%)	time (s)	#nodes
Auxiliary	10	0.00	0.57	65
MostAccHubs	10	0.00	0.44	28
GrDemReq	10	0.00	0.43	39
AddGrDemReq	10	0.00	0.42	47
ShAccArcs	10	0.00	0.49	86
Auxiliary	15	0.00	2421.30	210345
MostAccHubs	15	0.00	1933.11	218982
GrDemReq	15	0.00	2273.06	269229
AddGrDemReq	15	0.00	1302.82	164401
ShAccArcs	15	0.00	1076.54	125010
Auxiliary	20	0.42	T.L. (in 5 over 7)	116812
MostAccHubs	20	0.87	T.L. (in 6 over 7)	98676
GrDemReq	20	0.48	T.L. (in 5 over 7)	121009
AddGrDemReq	20	0.32	T.L. (in 6 over 7)	87537
ShAccArcs	20	0.17	T.L. (in 3 over 7)	124576
Auxiliary	25	1.12	T.L.	23098
MostAccHubs	25	0.94	T.L.	19169
GrDemReq	25	0.99	T.L.	31314
AddGrDemReq	25	0.83	T.L.	21437
ShAccArcs	25	1.19	T.L.	43721
Auxiliary	30	0.74	T.L.	14314
MostAccHubs	30	0.64	T.L.	8357
GrDemReq	30	0.81	T.L.	18655
AddGrDemReq	30	0.66	T.L.	8468
ShAccArcs	30	0.78	T.L.	16012
Auxiliary	35	∞	T.L.	1
MostAccHubs	35	∞	T.L.	1
GrDemReq	35	25.08	T.L.	1
AddGrDemReq	35	88.43	T.L.	1
ShAccArcs	35	48.29	T.L.	1
Auxiliary	40	∞	T.L.	1
MostAccHubs	40	∞	T.L.	1
GrDemReq	40	8.14 (in 1 over 3)	T.L.	1
AddGrDemReq	40	∞	T.L.	1
ShAccArcs	40	37.43 (in 1 over 3)	T.L.	1
Auxiliary	45	∞	T.L.	1
MostAccHubs	45	∞	T.L.	1
GrDemReq	45	∞	T.L.	1
AddGrDemReq	45	∞	T.L.	1
ShAccArcs	45	∞	T.L.	1
Auxiliary	50	∞	T.L.	1
MostAccHubs	50	∞	T.L.	1
GrDemReq	50	∞	T.L.	1
AddGrDemReq	50	0.32 (in 1 over 3)	T.L.	1
ShAccArcs	50	∞	T.L.	1

Table 5.1: Comparison among the different heuristic methods experiments

Table 5.1 presents the average results for the instances with a certain number of locations – *"#locations"* column – referred to the tested heuristic method (*"heuristic"* column). It reports the primal-dual gap, the solution time – reported as "T.L." when the time limit is reached – and the number of branching nodes explored.
In general all the heuristics have good computational results for instances with a number of locations lower or equal to 30: almost all the solutions have a gap lower than 1%. However, from the real-world case on, it becomes difficult to find accurate solutions in only one hour, and so they are not so reliable and comparable – because, as the *"nodes"* column shows, they only solve the root node.
To choose which is the best heuristic method among the proposed ones, we essentially based on the objective value – which is not reported in the table. Actually, the approach that gives better solutions in the majority of the cases was the "Additive Greatest Demand Requests Heuristic". The result was not so surprising, because the goal of that heuristic technique was to prioritize the hubs after a previous prioritization of the requests. We obtain good results from the "Most Accessed Hubs" heuristic too. Whereas the other three non-exact approaches returned in most cases a set of hubs which is not so accurate, and that consequently leads to worse objective values.

5.4.2 Matheuristic Experiments

In the previous section, we have seen how the Additive Greatest Demand Heuristic is the best non-exact solution approach proposed. For this reason, we decided to see how much this already good solution can be improved by means of a matheuristic method which perturbs the solution space by imposing two new constraints on the previous values of variables (see Section 4.2.5).
In particular, we apply this matheuristic technique through the Branch-and-Price tool, which allows to use the present SNDHLP heuristic model as starting restricted master problem, and then add the useful variables considering the "normal" path-based constraints plus the two new constraints on the opening of at least the 75% of the already open hubs and the limitation in the number of transfer vehicles on the intra-hubs arcs, which clearly have an influence in the computation of the dual values of the variables constituting a possible path to be added. The number of perturbations possible is equal to 10, but with a check on the values of two consecutive perturbed solution values, because if these are equal, it has no more sense to continue running the algorithm.
Furthermore, we decided to test the matheuristic approach also on the second best non-exact method of resolution: the Most Accessed Hubs Heuristic. The motivation was to see if by applying the perturbations on this heuristic, in the end there are better improvements in the solution, or similar final solution value to the approach applied on the other heuristic technique.

heuristic	#locations	gap (%)	time (s)	final after final run	heuristic improvement #perturbations (%)	heuristic improvement after 3 runs (%)	heuristic improvement after 1 run (%)	#nodes in final run
AddGrDemReq	10	0.00	87.21	2	8.41	–	8.41	52
MostAccHubs	10	0.00	125.97	3	22.87	22.87	18.17	54
AddGrDemReq	15	3.13	T.L.	5	20.35	19.06	15.69	1262
MostAccHubs	15	3.18	T.L.	5	21.87	20.16	16.11	1198
AddGrDemReq	20	2.41	T.L.	6	22.76	22.12	20.74	263
MostAccHubs	20	2.59	T.L.	7	21.80	20.31	18.66	288
AddGrDemReq	25	2.98	T.L.	5	20.81	19.98	17.61	79
MostAccHubs	25	2.97	T.L.	7	21.58	20.02	17.98	65
AddGrDemReq	30	2.57	T.L.	4	21.05	20.87	17.32	14
MostAccHubs	30	2.19	T.L.	5	22.00	21.45	18.35	32
AddGrDemReq	35	∞	T.L.	–	–	–	–	1
MostAccHubs	35	∞	T.L.	–	–	–	–	1
AddGrDemReq	40	∞	T.L.	–	–	–	–	1
MostAccHubs	40	∞	T.L.	–	–	–	–	1
AddGrDemReq	45	∞	T.L.	–	–	–	–	1
MostAccHubs	45	∞	T.L.	–	–	–	–	1
AddGrDemReq	50	∞	T.L.	–	–	–	–	1
MostAccHubs	50	∞	T.L.	–	–	–	–	1

Table 5.2: Comparison of the perturbation of two heuristic methods

The crucial parameter of Table 5.2 – necessary to understand which is the best trade-off in number of matheuristic perturbations and solution improvements – is the improvement of the matheuristic objective value with respect to the heuristic objective value, obtained by simply dividing their difference per the heuristic objective value. In particular, we compute this improvement in three different moments: after the first matheuristic iteration (column *"heuristic improvement after 1 run"*), after the third perturbation (column *"heuristic improvement after 3 runs"*), and after the final iteration (column *"heuristic improvement after final run"*). Besides, we report the final number of perturbations (column *"final #perturbations"*).
The results delineated how a greater number of perturbations is not related with more significant improvements in the solution value. Indeed, the difference between the 3-runs heuristic improvement and the final heuristic improvement is very small and never greater than 1.5%. This means that we can limit the number of perturbations to three, without any particular problem. Further, also the difference between the 3-runs improvement and the first heuristic improvement is not so marked, but for sure a bigger number of perturbations can improve the solving process – even though it costs more solving time.

5.4.3 Branch-and-Price Experiments

In the previous two sections, we have analyzed the non-exact solution approaches, and we have found out that the best heuristic is the Additive Greatest Demand Requests one and the best number of perturbations in the matheuristic method is three. Now the goal is to use these results to improve the Branch-and-Price algorithm, and find the best and the fast possible combination of solving features. Indeed, in Table 5.3 we report the test of the Branch-and-Price algorithm presented in Section 4.1.5 with three different ways of solving the RMP:

- Solving the restricted master problem with the set of hubs of the Additive Greatest Demand Requests heuristic, and then we apply the B&P algorithm – this is indicated as "RMPheur" in the *"RMP solver"* column.
- Solving the RMP with the set of hubs of the Additive Greatest Demand Requests heuristic method and then perturbing it for at maximum three times. Then this multiple-perturbed problem represents the starting RMP of the Branch-and-Price – denoted as "RMPmatheur" in the *"RMP solver"* column.
- Solving the RMP with the set of hubs of the auxiliary problem (see Section 4.1.2), but using the value of the objective function of the RMP as primal bound for the Branch-and-Price algorithm – method named "AuxPrBound" in the *"RMP solver"* column – in order to figure out if there is any improvement for what concerns the computational time and the final solution value, compared to the normal B&P.

RMP solver	#locations	heuristic improvement (%)	gap (%)	time (s)	time for gap $\leq$ 3% (s)	time for gap $\leq$ 5% (s)	time for gap $\leq$ 10% (s)	#nodes	#pricing-Iterations
RMPheur	10	29.76	0.00	118.01	62.30	45.71	27.35	103	295
RMPmatheur	10	29.76	0.00	200.74	31.53	26.60	20.31	90	168
AuxPrBound	10	–	0.00	217.55	69.88	65.37	63.37	110	287
RMPheur	15	18.23	2.01	T.L.	712.13	442.80	292.87	744	1834
RMPmatheur	15	19.64	1.63	T.L.	286.90	49.77	38.56	684	1179
AuxPrBound	15	–	0.12	T.L.	265.80	95.47	92.38	807	1887
RMPheur	20	18.98	4.36	T.L.	1143.25 (in 5 over 7)	709.40 (in 6 over 7)	547.83	115	545
RMPmatheur	20	21.72	2.68	T.L.	238.45 (in 6 over 7)	164.85	147.28	135	386
AuxPrBound	20	–	4.31	T.L.	1013.17 (in 5 over 7)	548.99 (in 6 over 7)	511.53	116	442
RMPheur	25	16.61	2.29	T.L.	2156.40	2136.88	1963.29	13	184
RMPmatheur	25	17.72	2.09	T.L.	862.04	802.57	757.48	24	141
AuxPrBound	25	–	4.08	T.L.	2141.14 (in 5 over 7)	2141.14 (in 5 over 7)	1999.22 (in 6 over 7)	10	177
RMPheur	30	4.98	∞	T.L.	–	–	–	1	52
RMPmatheur	30	6.69	∞	T.L.	–	–	–	1	39
AuxPrBound	30	–	∞	T.L.	–	–	–	1	54
RMPheur	35	5.01	∞	T.L.	–	–	–	1	38
RMPmatheur	35	5.27	∞	T.L.	–	–	–	1	25
AuxPrBound	35	–	∞	T.L.	–	–	–	1	34
RMPheur	40	0.41	∞	T.L.	–	–	–	1	23
RMPmatheur	40	2.53	∞	T.L.	–	–	–	1	12
AuxPrBound	40	–	∞	T.L.	–	–	–	1	22
RMPheur	45	0.00	∞	T.L.	–	–	–	1	16
RMPmatheur	45	1.41	∞	T.L.	–	–	–	1	10
AuxPrBound	45	–	∞	T.L.	–	–	–	1	15
RMPheur	50	0.00	∞	T.L.	–	–	–	1	12
RMPmatheur	50	1.91	∞	T.L.	–	–	–	1	6
AuxPrBound	50	–	∞	T.L.	–	–	–	1	11

Table 5.3: Comparison of different RMP solution for the B&P algorithm

In Table 5.3 the goal was to find out which one of the three different initialization of the RMP represents the best combination of features for the final Branch-and-Price, or if it could seem reasonable to combine them too. For this reason, it was important to compare the heuristic improvement of the heuristic and matheuristic method, and the solving times necessary to reach a certain percentage of primal-dual gap. It is important to remark that in the matheuristic case this time has to take into account a richer starting RMP at the beginning, but a previous longer solving time than the other two methods – which is not specified in the table. In particular, due to the huge size of the problem, we consider reasonable and acceptable solutions the ones with a primal-dual gap lower or equal than 10%. Hence, we compute the time to arrive to this gap, and the time to reach the gap of 5% and then of 3% – columns *"time for gap $\leq$ 10%"*, *"time for gap $\leq$ 5%"*, and *"time for gap $\leq$ 3%"* – to see the speed of the process in approaching the close-to-optimality values.
In general, we noticed the shortest times in the case of the matheuristic restricted master problem. However, this can surely be considered a good starting point, but – as explained before – it is a direct consequence of the more complete starting RMP, which already comprises promising variables to the final problem solution. Considering the other two methods, none seemed to clearly show a "superiority", because they both have a positive aspect: the one a more accurate RMP, and the other a primal bound limit. For this reason, we decide that the best combination of features for initializing the RMP is the Additive Greatest Demand Requests heuristic resolution plus the imposition of its objective value as primal bound for the next B&P steps.

5.4.4 Early Branching Branch-and-Price Experiments

This forth skimming experimental phase served to test the early branching tool in the Branch-and-Price algorithm. As explained in Section 2.2.3, the inequality (2.5) offers a large opportunity for speeding up the B&P process, by means of the so-called early branching. Actually this consists in the computation in each pricing iteration of a Lagrangian bound LB, which represents the lower bound of the current Branch-and-Bound node. After having reviewed the trend of the Lagrangian gap $LG = \frac{LB - z_{\text{RMP}}}{LB}$ over different pricing iterations of different instances, we decide to set the early-branching threshold to 0.05. Hence, when in a B&B node the Lagrangian gap is lower than this threshold, the pricing iterations are stopped on that node and the Branch-and-Price proceeds analyzing the next node of the search tree.
Specifically, we tested the early branching tool both on the heuristic resolution of the RMP and on the matheuristic initialization of the RMP, imposing their primal bound as objective limit for the next B&P. Then we compute solving times for the three gaps of 3, 5 and 10 % to compare them with the same values of Table 5.3.

RMP solver	#locations	heuristic improvement (%)	gap (%)	time (s)	time for gap $\leq 3\%$ (s)	time for gap $\leq 5\%$ (s)	time for gap $\leq 10\%$ (s)	#nodes	#pricing-Iterations
RMPheur	10	31.72	0.00	112.23	52.45	24.20	13.76	131	248
RMPmatheur	10	31.72	0.00	255.62	41.25	2.23	1.70	94	153
RMPheur	15	19.23	1.62	T.L.	342.23	256.72	234.20	462	1097
RMPmatheur	15	19.64	1.73	T.L.	350.21	25.35	15.56	318	557
RMPheur	20	20.11	2.74	T.L.	2396.51	1470.43	417.43	123	417
RMPmatheur	20	21.72	2.86	T.L.	1100.65	233.23	100.14	103	264
RMPheur	25	19.84	2.21	T.L.	3471.64	2904.52	1437.10	22	131
RMPmatheur	25	20.51	2.24	T.L.	1823.85	1233.78	916.12	29	66
RMPheur	30	5.08	∞	T.L.	–	–	–	1	47
RMPmatheur	30	6.37	∞	T.L.	–	–	–	1	42
RMPheur	35	4.76	∞	T.L.	–	–	–	1	65
RMPmatheur	35	5.10	∞	T.L.	–	–	–	1	25
RMPheur	40	0.58	∞	T.L.	–	–	–	1	51
RMPmatheur	40	2.39	∞	T.L.	–	–	–	1	12
RMPheur	45	0.06	∞	T.L.	–	–	–	1	16
RMPmatheur	45	1.41	∞	T.L.	–	–	–	1	10
RMPheur	50	0.00	∞	T.L.	–	–	–	1	12
RMPmatheur	50	0.97	∞	T.L.	–	–	–	1	6

Table 5.4: Results of early branching on the B&P algorithm

In the previous Table 5.4, it is quite clear how both the final primal-dual gap values and the solving times to reach the three gap thresholds of 3, 5 and 10 % are generally better than the same ones in Table 5.3. So, we can easily conclude that the final B&P has to include the early branching tool. In addition, the values of the heuristic RMP are closer to those of the matheuristic approach. Thus, in the end, the best and fast combination of features for the Branch-and-Price algorithm is the resolution of the RMP with the Additive Greatest Demand Requests Heuristic, the imposition of the RMP primal bound as objective limit, and the use of the early-branching threshold of 0.05.

5.4.5 Arc-based Model Experiments

This last skimming phase is to compare the solutions of the commercial solver Gurobi with the open source SCIP, and then choose how to solve the same instances for having the benchmark in the final testing phase. However, differently from the other four skimming experiments, here we set a time limit of 2 hours.

As it was easily imaginable, the Table 5.5 confirms us how Gurobi – as commercial solver – is way better than SCIP both in terms of solving speed and final primal-dual gap. Besides, it is also capable to solve close-to-optimality instances of larger size by enumeration compared to SCIP, which cannot solve instances with more than 20 locations. As a consequence, it will be used as benchmark solver for the final experimental phase.

MIP solver	#locations	gap (%)	time (s)	time for gap $\leq 3\%$ (s)	time for gap $\leq 5\%$ (s)	time for gap $\leq 10\%$ (s)	#nodes
SCIP	10	0.00	122.37	62.18	50.07	34.29	38
Gurobi	10	0.00	27.46	10.73	4.52	2.94	62
SCIP	15	2.52	T.L.	4735.88	2932.77	765.02	1317
Gurobi	15	0.00	1116.72	97.73	50.93	42.21	1324
SCIP	20	5.25	T.L.	–	–	7416.73	13
Gurobi	20	0.00	5203.26	506.62	257.71	176.63	1977
SCIP	25	∞	T.L.	–	–	–	1
Gurobi	25	0.57	T.L.	522.57	495.29	302.98	13893
SCIP	30	∞	T.L.	–	–	–	1
Gurobi	30	0.79	T.L.	2098.39	1342.84	1102.35	15562
SCIP	35	∞	T.L.	–	–	–	1
Gurobi	35	∞	T.L.	–	–	–	1
SCIP	40	∞	T.L.	–	–	–	1
Gurobi	40	∞	T.L.	–	–	–	1
SCIP	45	∞	T.L.	–	–	–	1
Gurobi	45	∞	T.L.	–	–	–	1
SCIP	50	∞	T.L.	–	–	–	1
Gurobi	50	∞	T.L.	–	–	–	1

Table 5.5: Comparison of SCIP and Gurobi MIP solvers on the arc-based SNDHLP

5.5 Final Comparison Experiments

This section is dedicated to the last phase of our computational experiments, in which we took into account the best methods of the preliminary skimming phases, and we compared them. In particular, we considered as our main solution approach the Branch-and-Price with the RMP solved through the Additive Greatest Demand Heuristic and no perturbations, but including the primal bound limit and the early branching tool that speeds up the solving process. This is compared with the arc-based SNDHLP solved with Gurobi and the 3-perturbations matheuristic method applied over the Additive Greatest Demand Heuristic resolution of the path-based model. Further, Section 5.5.4 is reserved to some experiments on the same instances, but targeting the unsplittable requests case, to see if there are any differences with the splittable ones.
All these experiments have been performed over a total of ninety-eight instances: a total of sixty-eight referred to the real-world case and its neighborhood (seventeen instances for each problem with 25, 30, 35 or 40 locations), and a total of thirty instances related to the small ones and possible expansion of the system (six for each instance of 10, 15, 20, 45, and 50 locations). We used all the three datasets: AP, CAB, and TR, for each group of instances. Besides, seven different cutting methods of the AP and CAB datasets have been used for retrieving fourteen new realistic and neighborhood instances. Another different selection method applied also on the instance of the TR dataset derived three new small and possible-system-expansion instances.
The time limit imposed was of one day and a half – 36 hours – for the B&P and the matheuristics experiments (12 hours for every perturbation), and of 4 hours for the Gurobi arc-based model. For all the experiments we set a gap limit of 0.1%.
Finally, in Section 5.5.5 we decided to relax the arc-based model not taking into account the two constraints of maximum transport time and limited number of hops to see if the solution structure changes compared to the complete model.

5.5.1 Branch-and-Price Experiments

The final Branch-and-Price experiments consider the best combination of tools possible both to obtain solutions as close-to-optimality as possible and to speed up the solving process. Indeed, as Sections 5.4.3 and 5.4.4 show, the best B&P:

- solves the restricted master problem using the sets of hubs derived from the Additive Greatest Demand Heuristic problem
- imposes the objective value of the RMP solution as primal bound limit
- uses the early branching procedure in the pricing iterations

#locations	gap (%)	time (*min*)	time for gap $\leq 3\%$ (*min*)	time for gap $\leq 5\%$ (*min*)	time for gap $\leq 10\%$ (*min*)	#nodes	#pricing-Iterations
10	0.05	80.61	0.38	0.26	0.08	207	398
15	0.39	T.L. (in 4 over 6)	12.15	5.74	1.86	16053	19686
20	0.45	T.L.	63.95	24.86	9.34	8147	13664
25	2.96	T.L.	743.33 (in 9 over 17)	282.88 (in 14 over 17)	91.56	455	974
30	6.81	T.L.	–	308.67 (in 5 over 17)	152.48 (in 13 over 17)	77	334
35	6.57	T.L.	–	790.50 (in 2 over 17)	688.88 (in 13 over 17)	6	127
40	12.79	T.L.	–	–	1467.32 (in 4 over 17)	2	118
45	26.11	T.L.	–	–	–	1	112
50	83.23	T.L.	–	–	–	1	49

Table 5.6: Results of final Branch-and-Price Experiments

From Table 5.6, first of all we can see how after one day and a half we can solve the realistic instances of 35 locations with a tolerance of about 6.5%. This is a very positive result, which is furthermore enforced by the average solving time of less than 12 hours for reaching the 10% gap threshold.
Considering the close-to-reality cases, the results are even more promising for the 25 and 30 locations cases – they only require respectively 1 hour and a half and 2 hours and a half for the 10% threshold, and in 12 hours, the instances with 25 locations are solved very close-to-optimality. The bigger case of 40 locations still requires a day to obtain acceptable solution value (and only in a small number of tested instances). Obviously, this situation worsens the greater the number of locations is.

5.5.2 Matheuristic Experiments

The final experimental phase of the matheuristic approach considers 3 as reasonable number of perturbations – as seen in Section 5.4.2. The solution method initially solves the complete path-based SNDHLP via the Additive Greatest Demand Requests Heuristic, whose objective value is used as next primal bound limit. Then, it sets a time limit of 12 hours for each of the three iterations which applies the local branching technique described in Section 4.2.5 through the Branch-and-Price algorithm including the early branching procedure in the pricing iterations.

From Table 5.7, we evince that in general the matheuristic is less performing than the Branch-and-Price algorithm. Indeed, in the final perturbation it obtains similar results to the final B&P experiments (see Table 5.6) in terms of speed to reach the three gap thresholds of 3, 5 and 10 %, but this means that there have already been about 24 hours of resolution before that moment. Besides, the objective values are generally greater than the B&P solutions ones.

#locations	first heuristic improvement (%)	final heuristic improvement (%)	gap (%)	time (s)	time for gap $\leq 3\%$ (s)	time for gap $\leq 5\%$ (s)	time for gap $\leq 10\%$ (s)	#nodes
10	16.31	19.82	0,00	169.67	16.76	5.71	1.23	104
15	14.94	15.59	0.38	T.L.	128.22	16.53	14.95	1897
20	22.36	25.04	0.83	T.L.	133.56	104.29	89.12	251
25	32.29	37.58	1.67	T.L.	8045.78	5362.13	4059.62	46
30	26.99	30.18	8.91	T.L.	–	–	12254.84 (in 9 over 17)	21
35	7.70	8.01	23.79	T.L.	–	–	–	2
40	0.00	0.23	41.46	T.L.	–	–	–	1
45	0.00	0.00	88.61	T.L.	–	–	–	1
50	0.00	0.00	93.23	T.L.	–	–	–	1

Table 5.7: Results of final Matheuristic Experiments

5.5.3 Arc-based Model Experiments

The benchmark for the B&P experiments was the arc-based model solved through Gurobi. Given the huge power of the solver and the function of benchmark, we set a time limit of only 4 hours, in order to rapidly had comparable outputs. Besides, apart from the three gap thresholds present in all the other tables, we also report the solving time necessary to arrive to the gap of only 1% – column *"time for gap $\leq$ 1%"*.

What clearly emerges from the Table 5.8 is the very fast solving time for reaching the gap thresholds: in the real-world case of 35 locations, only about 70 minutes are required to obtain a tolerance of 1% – and a bit less than 1 hour for the 3% gap – with a final primal-dual gap of only 0.35% in 4 hours. The results are better for the semi-realistic instances with 25 and 30 locations, and a bit worse for the 40 locations case – with an average final gap in the neighborhood of 1%. However, given the huge number of variables, Gurobi too runs out of memory and is not able to compute any solutions for the instances with 45 or more locations.

#locations	gap (%)	time (s)	time for gap $\leq 1\%$ (s)	time for gap $\leq 3\%$ (s)	time for gap $\leq 5\%$ (s)	time for gap $\leq 10\%$ (s)	#nodes
10	0.00	22.19	9.63	8.75	3.66	2.94	79
15	0.00	607.26	148.31	77.21	60.34	42.21	2891
20	0.00	5671.76	319.62	224.06	162.51	125.63	6079
25	0.27	T.L. (in 11 over 17)	742.78	428.29	385.68	304.74	13932
30	0.38	T.L.	2461.54	1339.83	1262.24	1202.08	16876
35	0.35	T.L.	4320.29	3417.04	3115.43	3047.23	832
40	0.94	T.L.	10342.81 (in 12 over 17)	8718.45	8552.13	8195.07	143
45	∞	T.L.	–	–	–	–	1
50	∞	T.L.	–	–	–	–	1

Table 5.8: Results of final Experiments on the arc-based model solved with Gurobi

5.5.4 Unsplittable Requests Instances Experiments

After all the experiments targeting only the splittable requests case, one of these final tests considered instances with unsplittable requests ones. The reason why the majority of the tests has been performed on the splittable requests instances is the inner greater complexity of these. Thus, if the solution approaches worked on them, they would have worked also on the unsplittable requests instances. We decided to test also the latter ones comparing the Gurobi MIP solver of the arc-based SNDHLP – named "arc-based" in Table 5.9 – and our final Branch-and-Price approach explained in Section 5.5.1 – denoted "path-based" in Table 5.9. In this way, we tested both the two model formulations, to see whether there are any performance changes.

Generally, Table 5.9 shows a bit worse performance compared to the same instances of the splittable requests case. The result is partially surprising. Indeed, for what said before, we expected better or similar performance. However, this difference is not so significant, even if the Gurobi solver is not able to solve any real-world instance in 4 hours. Further, obviously, the objective value of all the instances is greater than the splittable case ones, as there is only a path – or a combination of arcs – possible for each request.

model	#locations	gap (%)	time (s)	time for gap $\leq 1\%$ (s)	time for gap $\leq 3\%$ (s)	time for gap $\leq 5\%$ (s)	time for gap $\leq 10\%$ (s)	#nodes	#pricing-Iterations
arc-based	10	0.00	15.19	9.80	8.75	3.66	2.94	62	–
path-based	10	0,00	119.67	87.95	56.76	32.66	21.45	189	395
arc-based	15	0.00	510.43	264.37	71.35	35.33	23.14	3487	–
path-based	15	0.43	T.L.	2086.53	928.25	823.21	228.35	27757	53902
arc-based	20	0.00	6959.36	376.31	201.42	162.55	103.60	23460	–
path-based	20	0.58	T.L.	9204.13	5163.56	2404.29	409.12	9914	14732
arc-based	25	0.21	T.L.	695.46	593.07	535.79	428.32	26586	–
path-based	25	3.45	T.L.	–	57478.38	15452.13	3699.87	464	1530
arc-based	30	0.37	T.L.	1956.80	1392.89	1347.37	1256.64	11591	–
path-based	30	8.55	T.L.	–	–	–	9154.23	54	236
arc-based	35	∞	T.L.	–	–	–	–	1	–
path-based	35	6.83	T.L.	–	–	–	13935.68	37	127
arc-based	40	∞	T.L.	–	–	–	–	1	–
path-based	40	15.69	T.L.	–	–	–	–	2	164
arc-based	45	∞	T.L.	–	–	–	–	1	–
path-based	45	36.18	T.L.	–	–	–	–	1	104
arc-based	50	∞	T.L.	–	–	–	–	1	–
path-based	50	83.23	T.L.	–	–	–	–	1	47

Table 5.9: Results of Unsplittable Requests Instances Experiments

5.5.5 Relaxed Arc-based Model Experiments

In this last testing phase, we decided also to relax the arc-based model and do not consider the two constraints of maximum transport time and limited number of transshipments. The reason is to understand how much these two constraints influence the solutions and the speed of the solving process, and then to see how differently the solutions are structured in terms of "paths" and used transfer vehicles. Table 5.10 highlights the improvements in terms of both solving times necessary to reach the gap thresholds and final primal-dual gap, compared to the complete arc-based SNDHLP seen in Table 5.8. In addition, in this case, Gurobi does not run out of memory and can solve the 45-locations instances and one of the six instances with 50 locations.
The motivation of these improvements is quite easy to figure out: there is a lower number of variables and constraints to consider in the model and so it solves faster. Indeed, there are no more the binary request arcs variables e_a^r, and the ones indicating transport time and number of hops up to the hub i – respectively variables w_i^r and s_i^r.
The relaxed splittable requests arc-based SNDHLP model looks like the complete unsplittable requests arc-based one without the two constraints (3.8) and (3.9), and with continuous requests arcs variables, instead of binary.
We recall that the model can be formalized in mathematical terms as follows:

$$
\begin{aligned}
\min \quad & \sum_{a \in A_t} c_t l_a v_a + \sum_{r \in R} \sum_{a \in A_s} c_s l_a d^r x_a^r \\
\text{s.t.} \quad & \sum_{a \in \delta^+(k_1)} x_a^r = 1 && \forall\, r = (k_1, k_2) \in R \\
& \sum_{a \in \delta^-(k_2)} x_a^r = 1 && \forall\, r = (k_1, k_2) \in R \\
& \sum_{a \in \delta^+(i)} x_a^r - \sum_{a \in \delta^-(i)} x_a^r = 0 && \forall\, i \in H,\ r \in R \\
& \sum_{r \in R} d^r x_a^r \le K v_a && \forall\, a \in A_t \\
& \sum_{i \in H} h_i = n_{\mathrm{H}} \\
& \sum_{a \in \delta_i^-} x_a^r \le h_i && \forall\, i \in H,\ r \in R \\
& h_i \in \{0,1\} && \forall\, i \in H \\
& v_a \in \mathbb{N}_0 && \forall\, a \in A_t \\
& x_a^r \in [0,1] && \forall\, r \in R,\ a \in A
\end{aligned}
\tag{5.11}
$$

#locations	gap (%)	time (s)	time for gap ≤ 1% (s)	time for gap ≤ 3% (s)	time for gap ≤ 5% (s)	time for gap ≤ 10% (s)	#nodes
10	0.00	6.19	5.63	3.75	2.66	1.14	123
15	0.00	363.06	136.82	26.43	25.21	22.32	3452
20	0.00	2784.92	86.53	28.25	28.14	24.33	8457
25	0.11	T.L. (in 12 over 17)	189.69	104.25	92.51	64.80	23945
30	0.26	T.L.	204.13	123.98	115.24	87.37	40574
35	0.29	T.L.	256.64	137.45	135.43	127.28	6586
40	0.45	T.L.	541.80	478.38	452.13	359.12	2514
45	1.43	T.L. (in 4 over 6)	2076.51	2007.14	1876.85	1798.00	1572
50	∞ (in 5 over 6)	T.L.	–	–	–	–	1

Table 5.10: Results of Experiments on the arc-based model without time and transshipments constraints

Chapter 6

Conclusions

In this thesis we have studied a particular version of a Service Network Design and Hub Location Problem for combined transport with the opportunity of multiple itineraries for shipping goods with the same starting and ending points. We take into account many important real-world constraints that affect the routing of commodities, such as capacity restrictions on links among the hubs, deadlines for delivery, and limits on the number of transshipments that can occur. The problem involves determining the optimal placement of hubs as well as planning the actual transportation of the freight. Our final goal is to strike a balance between service quality and operational expenses.
To solve the problem, we developed a Branch-and-Price algorithm that employs various features and a tailored heuristic initialization approach. In addition, we built up different customized heuristic solution techniques, which try to exploit different critical aspects of the problem and a matheuristic method.

We tested our algorithms using data from realistic instances, and our results show that they can accurately find close-to-optimality solutions, even for large instances, which makes it practical for planning purposes.
Specifically, the B&P algorithm is able to solve real-world sized instances with 35 locations in about 12 hours with a tolerance of only 5% – which is very low considering the size of the problem. These outcomes are even more promising for close-to-reality cases where the solution times decrease by more than twice – they require about 5 hours – and within the same computational time the final solutions are closer to the optimal value. The effectiveness of our algorithm decreases with even larger instances, but it still finds not-so-bad solutions within longer computational times.
The comparison of our B&P algorithm with the commercial solver Gurobi – that is not able to perform a Branch-and-Price method and can only solve the polynomial-sized arc-based model – offers interesting insights. As a commercial solver, it is

very fast compared to the open-source SCIP tools on which our B&P is based, and obtains the same results in about one hour, and better ones in some minutes more. However, the Gurobi solver runs out of memory when the number of locations increases and is not able even to start solving them, because of the huge number of variables. On the other hand, our B&P is always able to provide a solution to the problem, for large non-realistic instances too. A common characteristic is the slow convergence between primal and dual bounds, as the solution time to reach very good primal-dual gaps is very small, but then the two solvers require many hours to reach the value 0.0 and very often do not reach it.

About the other solution approaches proposed, they have been very helpful in the definition of the final improved version of our Branch-and-Price. Besides, the various heuristic methods proposed help us understand what is the most relevant parameter for the choice of the hubs to open: the demand of the requests, and not the number of customers served by a hub or the distances between customer and hub locations. In particular, to choose a proper set of hubs to open it is very useful to prioritize the requests on the basis of their demand sizes. Then, the ranking of each request is used to give a weighted score to the hubs which serve the related customers, and in the end the hubs with the greatest score are opened.
Another interesting characteristic observed is the structure of the paths present in the solutions. Indeed, when the number of locations is small, the number of 2-legs-of-trip paths is about the 30% of the total, whereas for realistic and larger-than-realistic instances, this percentage value goes down to about 20%. Besides, the structure of paths changes also in relation to the ratio between the costs of access arcs c_s and transfer arcs c_t. When the former is significantly smaller than the latter divided by the capacity of transfer vehicles, there is a complete predominance of 2-legs-of-trips paths, whereas they are very rare in the opposite case.

In conclusion, possible challenging future developments of this work are represented by the extension of the model targeting other realistic situations such as limited capacities, stocking costs and waiting times in hubs stations, or the use of different types of vehicles. Then the exploitation of economies of scale for common routes and most used hubs might become interesting tools to reduce the costs.
From the algorithmic point of view, the Branch-and-Price can be improved with the implementation of a heuristic pricing and then the use of tailored cutting planes. These engaging improvements can have a positive outcome on the performance of our solver, that might become more comparable to the commercial solver Gurobi. Finally, it would be interesting to investigate the use of our proposed algorithms for similar problems and to extend our study to other related real-world applications.

Appendix A

SNDHLP model creation

A.1 SNDHLP instance reader

In this section we present the general function for reading a generic instance from the used datasets, and then set the problem parameters for the SNDHLP model.

```
#this method is for reading a generic file of an instance of SNDHLP
def readSNDHLPfromFile(fileName):
    file = open(fileName, "r").readlines()

#create the set of customers (the number of customers is on the first line of the file) and hubs
    numberCustomers = int(file[0])
    digits = len(file[0]) #this is the number of digits + 1 in the customers number
    addingNodeCustomer = 10 #this is useful to create the corresponding customer node int
    for n in range(digits):
        addingNodeCustomer *= 10
#after the for cycle it would be at least 1000 (in case of less than 10 customers), otherwise it would be 10000 (for 10 to 99 customers), 100000 (for 100 to 999 customers) and so on
#to sum up the "duplicated" customers of the hub h is 100h (with the "string" "100" before the hub number)

    numberHubs = numberCustomers
    if(file[1] != "ALL\n"): #if this line is "ALL", it means that all the customers can also be hubs (in all our instances this line is equal to "ALL")
        numberHubs = int(file[1])
    customers = set()
```

```
        hubs = set()
        for i in range(numberCustomers):
            customerNode = i + 1 + addingNodeCustomer #to generate the "duplicated" customer referred to the same hub location
            customers.add(customerNode)
            hubs.add(i+1)

    #create the two sets of intraHubs arcs (all links between any two hubs) and of accessArcs (by taking in both directions a specific percentage of closest hubs to the customer + the customer-location hub)
        intraHubsArcs = {}
        accessArcsPercentage = 0.3 if numberCustomers<=20 else (0.25 if numberCustomers<40 else 0.2)
        numAccessArcs = int(accessArcsPercentage*numberCustomers)
        accessArcs = {}
        for i in range(numberCustomers):
            customerNode = i + 1 + addingNodeCustomer
            accessArcs[tuple([customerNode,i+1])] = 0.0
            accessArcs[tuple([i+1,customerNode])] = 0.0
            l = i + 2 #because we have to skip the first 2 lines of the files
            distanceLine = list(file[l].split())
            closeHubs = {}
            for j in range(numberCustomers):
                if i != j:
                    dist = round(float(distanceLine[j]), 2)
                    intraHubsArcs[tuple([i+1,j+1])] = dist
                    closeHubs[j+1] = dist
    #here we create the set of accessArcs taking the numAccessArcs closest hubs to the customer and then adding them in both direction
            closestHubs = sorted(closeHubs.items(), key=lambda x:x[1])
            for c in range(numAccessArcs):
                accessArcs[tuple([customerNode,closestHubs[c][0]])] = closestHubs[c][1]
                accessArcs[tuple([closestHubs[c][0],customerNode])] = closestHubs[c][1]

    #creates the requests dictionary
        requests = {}
        for i in range(numberCustomers):
            l = i + numberCustomers + 2 #because we have to skip the first 2 lines of the files + all the lines of the distances' matrix = #nodes
            demandLine = list(file[l].split())
            for j in range(numberCustomers):
                dem = round(float(demandLine[j]), 2)
```

```
                if dem > 0:
                    requests[tuple([i+1+addingNodeCustomer, j+1+addingNodeCustomer])] = dem

        return customers, hubs, accessArcs, intraHubsArcs, requests, numberHubs

    #this method obtains the average distance and the average demand of a specific instance of SNDHLP
    def getInstanceData(fileName):
        file = open(fileName, "r").readlines()
        numberCustomers = int(file[0])

    #computing the total distances between all the customers
        totalDistance = 0.0
        for i in range(numberCustomers):
            l = i + 2 #skip the first 2 lines of the files
            distanceLine = list(file[l].split())
            for j in range(numberCustomers):
                    totalDistance += round(float(distanceLine[j]), 2)

    #computing the total demands between all customer pairs
        totalDemands = 0.0
        for i in range(numberCustomers):
            l = i + numberCustomers + 2 #skip the first 2 lines + the lines of the distances
            demandLine = list(file[l].split())
            for j in range(numberCustomers):
                totalDemands += round(float(demandLine[j]), 2)

    #computing the average distance and demand for our specific instance
        avgDistance = round(float(totalDistance/(numberCustomers*(numberCustomers-1))), 2)
        avgDemand = round(float(totalDemands/(numberCustomers*(numberCustomers-1))), 2)
        return avgDistance, avgDemand, numberCustomers

    #this method set the time limit and the gap limit for the problem
        def setTimeAndGapLimits(master, timeLimit, gapLimit):
            if timeLimit is not None:
                master.setRealParam("limits/time", timeLimit)
            if gapLimit is not None:
                master.setRealParam("limits/gap", gapLimit)

    #the next method sets the problem parameters and limits for the constraints:
```

```
    #defining the number of hubs to be opened and the maximum number of hops on the basis of the number of customers, and the vehicles capacity and the costs of arcs on the basis of the average demand of requests, whereas the max transport time depends on the average distances
    def setProblemParameters(avgDistance, avgDemand, numberCustomers):

        numHubs = int(numberCustomers**0.6)
        numHops = 3 if numHubs<=6 else (4 if numHubs<=10 else 5)
        maxTransportTime = 5*avgDistance
        vehicleCapacity = max(1, 5*int(avgDemand))
        intraHubsCost = 200
        accessCost = round(2*(intraHubsCost/vehicleCapacity), 2)
        return numHubs, numHops, maxTransportTime, vehicleCapacity, intraHubsCost, accessCost
```

A.2 SNDHLP instance sets generator

In this section we present the general functions necessary to define an instance for the SNDHLP model.

```
from dataclasses import dataclass
import networkx as nx
from typing import Tuple

#this method creates a graph from the starting sets of hubs and their relative links
def createHubsGraph(hubs, intraHubsArcs):
    hubsGraph = nx.DiGraph(n_res=2)
    hubsGraph.add_nodes_from(hubs)
    for (arc, dist) in intraHubsArcs.items():
        arcTime = dist*0.5
#the intra-hubs arc time is the time in minutes necessary to cover the distance between the two hubs, and it is assumed equal to half the distance (considering the low traffic conditions in the hubs' network, and an average speed of 120km/h for transfer vehicles)
        hubsGraph.add_edge(arc[0], arc[1], res_cost=[1, arcTime], distance=dist, weight=0.0)
    return hubsGraph

#this creates the graph for a specific request by adding to the hubsGraph the two customers nodes and their relative accessArcs with the check that the added "outgoing" accessArcs go only from the customer origin to a hub and that the added "ingoing" accessArcs go only from a hub to the customer destination
def createRequestGraph(hubsGraph, origin, destination, hubs, accessArcs):
    requestGraph = hubsGraph.copy()
    for (arc, dist) in accessArcs.items():
        arcTime = dist*0.75
#the access arc time is the time in minutes necessary to cover the distance between the customer and the hub (or viceversa), and it is assumed equal to 0.75 times the distance (assuming more traffic conditions than the intrahubs arcs and an average speed of 80km/h for vehicles)
        if origin == arc[0] and arc[1] in hubs:
            requestGraph.add_edge(arc[0], arc[1], res_cost=[1, arcTime], distance=dist, weight=dist)
        elif destination == arc[1] and arc[0] in hubs:
            requestGraph.add_edge(arc[0], arc[1], res_cost=[1, arcTime], distance=dist, weight=dist)
    return requestGraph
```

```python

    @dataclass
    class Path:
        nodes: list[int] # the list of nodes in the path
        hubs: list[int] # the list of nodes that are hubs in the path (all the nodes a part from the origin and the destination)
        arcs: list[Tuple[int, int]] # the list of arcs in the path
        intraHubsArcs: list[Tuple[int, int]] # the list of "middle" arcs between hubs in the path (if there are)
        extremeArcs: list[Tuple[int, int]] # the first and the last arc of the path
        length: int # the lenght of the path = number of arcs in the path = len(arcs) = len(nodes) - 1
        totalDistance: float # the total distance in kilometers of the path
        totalTime: float # the total time in minutes for covering the entire path by a standard vehicle
        approximateTotalCost: float # the total cost of the path for a single demand unit = (distance1stArc + distanceLastArc)*costaccessArcs + distanceIntraHubsArc*intraHubsCostArcs/vehicleCapacity
        pathString: str # this is just the conversion to string of the list of nodes, that is useful for the problem model

    #this is the initialization of an object of this class when it is given a path and its graph + the arcs' costs and the capacity of vehicles through intra-hubs arcs
        def __init__(self, path, graph, accessCost, intraHubsCost, vehicleCapacity):
            self.nodes = path
            self.length = len(path) - 1
            self.hubs = path[1:self.length]
            self.arcs = list(nx.utils.pairwise(path))
            self.intraHubsArcs = self.arcs[1:self.length-1]
            self.extremeArcs = [self.arcs[0], self.arcs[self.length-1]]
            self.totalDistance, self.totalTime, self.approximateTotalCost = getPathAttributes(graph, self, accessCost, intraHubsCost, vehicleCapacity)
            self.pathString = str(path)

    #this method returns the total distance, the total time and the total approximate cost of a path
    def getPathAttributes(graph, path, accessCost, intraHubsCost, vehicleCapacity):
        distance = 0.0
        time = 0.0
        cost = 0.0
        for arc in path.intraHubsArcs:
```

```
            arcDistance = graph[arc[0]][arc[1]]["distance"]
            distance += arcDistance
            time += arcDistance*0.5
            cost += intraHubsCost*arcDistance/vehicleCapacity
        for arc in path.extremeArcs:
            arcDistance = graph[arc[0]][arc[1]]["distance"]
            distance += arcDistance
            time += arcDistance*0.75
            cost += accessCost*arcDistance

        totalDistance = round(distance, 2)
        totalTime = round(time, 2)
        totalCost = round(cost, 2)
        return totalDistance, totalTime, totalCost

    #to create the different subsets of the paths, the basic idea is simple: given a request r with origin r[0] and destination r[1], the set of feasible paths Pr is derived from all the accessArcs that start from r[0] or arrive in r[1] with all the possible intra-hubs links
    def getFeasiblePaths(hubsGraph, origin, destination, hubs, accessArcs, accessCost, intraHubsArcs, intraHubsCost, vehicleCapacity, maxPathLength, requestMaxTransportTime):
    #here by creating the requestGraph we guarantee the combined transport: avoiding direct transport origin-destination, as in any case a feasible path passes through at least one hub (that can also be a same location hub), and we also guarantee in all the paths the presence of a link between the extreme hub and the origin/destination (accessArc) because all the paths have necessarily the first and the the last arc that is an accessArc (there are no other arcs from origin or to destination, as we are adding them right now)
        requestGraph = createRequestGraph(hubsGraph, origin, destination, hubs, accessArcs)

    #this generates all the paths with maximum length maxPathLength for a request, each expressed as a sequence of nodes
        requestPaths = list(nx.all_simple_paths(requestGraph, origin, destination, maxPathLength))

    #as customers locations in the request are of the type "100h" where h is the number of the corresponding hub, we remove the first occurrence of "100" (for avoid problems with cases like 100100)
        originHub = int(str(origin).replace("100","", 1)) #this gives the corresponding same location hub of the origin customer of the request
```

```
        destinationHub = int(str(destination).replace("100","", 1)) # this gives the corresponding same location hub of the destination customer of the request

    #however in that list there are some "wrong" paths that has to be removed because
        #- they contain in their "middle arcs" an hub that coincides with the origin or the destination customer
        #- they have more than 3 nodes and the second node coincides with the destination or the second-to-last node coincides with the origin
    #and in both cases it has no sense that it continues exploring / has already explored other hubs after / before the origin or destination hub if they are not in the second or second-to-last position (but this is due to the necessary duplication of the customers done during the problem setup)
        pathsToRemove = []
        for path in requestPaths:
            if originHub in path[2:len(path)-2] or destinationHub in path[2:len(path)-2]:
                pathsToRemove.append(path)
            elif len(path)>3 and (path[1] == destinationHub or path[len(path)-2] == originHub):
                pathsToRemove.append(path)

    #the next 7 lines create the set Pr of all feasible paths for a request without the wrong paths to remove
        requestFeasiblePaths = []
        for path in requestPaths:
            if path not in pathsToRemove:
                newPath = Path(path, requestGraph, accessCost, intraHubsCost, vehicleCapacity)
                if newPath.totalTime <= requestMaxTransportTime: # this checks if the total time of the path is lower than the maximal allowed
                    requestFeasiblePaths.append(newPath)

        return requestFeasiblePaths

    #this creates the set Pri: a dictionary where the key is the hub i and the value is a list of all the paths containing that hub
    def getHubsFeasiblePaths(requestFeasiblePaths, hubs):
        requestHubFeasiblePaths = dict()
        for hub in hubs:
            hubFeasiblePaths = []
            for path in requestFeasiblePaths:
                if hub in path.nodes:
                    hubFeasiblePaths.append(path)
            if len(hubFeasiblePaths) > 0:
```

```
                requestHubFeasiblePaths[hub] = hubFeasiblePaths
        return requestHubFeasiblePaths

    #this creates the set Pra: a dictionary where the key is the arc a and the value is a list of all the paths containing a
    def getArcsFeasiblePaths(requestFeasiblePaths, intraHubsArcs):
        requestArcFeasiblePaths = dict()
        for arc in intraHubsArcs.keys():
            arcFeasiblePaths = []
            for path in requestFeasiblePaths:
                if arc in path.arcs:
                    arcFeasiblePaths.append(path)
            if len(arcFeasiblePaths) > 0:
                requestArcFeasiblePaths[arc] = arcFeasiblePaths
        return requestArcFeasiblePaths
```

A.3 Arc-based SNDHLP

This section presents the creation of the arc-based model.

```
from dataclasses import dataclass
from pyscipopt import Model, quicksum
from collections import defaultdict
from instanceReaderSNDHLP import readSNDHLPfromFile
from SNDHLPmodelFunctions import *

@dataclass
class sndhlpArcBasedInstance:
    customers: set[int] # subset of vertices (simply referred to by an integer id) that can be only origin or destination
    hubs: set[int] # subset of vertices that are hubs
    accessArcs: dict[Tuple[int, int], float] # dictionary representing the subset of arcs whose key is an arc a (identified by a tuple with the two nodes) which connects an origin or a destination to the hubs' network and whose value is its length
    intraHubsArcs: dict[Tuple[int, int], float] # dictonary representing the subset of arcs whose key is an intraHub arc a (identified by a tuple with the two hubs) which links two hubs and whose value is its length
    requests: dict[Tuple[int, int], float] # dictionary mapping customer pairs to demand values
    requestsGraph: dict[Tuple[int, int], nx.DiGraph] # dictionary mapping customer pairs to their request specific graph
    accessCost: float # cost per kilometer and demand unit for the subset of accessArcs
    intraHubsCost: float # cost per kilometer and vehicle for the subset of intraHubsArcs
    vehicleCapacity: float # given capacity of the means of transportation
    numberHubs: int # given number of hubs that have to be used
    numberHops: int # maximal number of transshipments at hubs
    requestsMaxTransportTime: dict[Tuple[int, int], float] # dictionary mapping customer pairs to their maximum time allowed for deliver a request

#this creates an arc-based SNDHLP instance
def createArcBasedSNDHLPinstance(fileName, accessCost, intraHubsCost, vehicleCapacity, numHubs, numHops, maxTransportTime):
#here we create the useful sets by reading a generic instance file
    customers, hubs, accessArcs, intraHubsArcs, requests, numberHubs = readSNDHLPfromFile(fileName)
```

```
    #this is to check if the input is okay with the content of the file and eventually correct it by imposing the file input
    if(numHubs > numberHubs):
        numHubs = numberHubs

    #here we create the graph with hubs and their links, and then two dictionaries mapping customer pairs to their specific request graph and maximum transport time
    hubsGraph = createHubsGraph(hubs, intraHubsArcs)
    requestsGraph = dict()
    requestsMaxTransportTime = dict()
    for r in requests.keys():
        requestsGraph[r] = createRequestGraph(hubsGraph, r[0], r[1], hubs, accessArcs)
        originHub = int(str(r[0]).replace("100","", 1))
        destinationHub = int(str(r[1]).replace("100","", 1))
        requestsMaxTransportTime[r] = maxTransportTime + 1.5*intraHubsArcs.get((originHub, destinationHub))

    #here we create the SNDHLP object, useful for the model creation
    sndhlp = sndhlpArcBasedInstance(customers, hubs, accessArcs, intraHubsArcs, requests, requestsGraph, accessCost, intraHubsCost, vehicleCapacity, numHubs, numHops, requestsMaxTransportTime)
    return sndhlp

@dataclass
#class of the relaxed arc-based model (without the max time and number of hops constraints)
class arcBasedModel:
    master: Model
    h: dict #hub variables
    x: defaultdict(dict) #request arcs variables representing the percentage of the request r transported by arc a (dictionaries that for each request have a dictionary mapping request to an arc)
    v: dict #number of vehicles per transfer arc variables
    numHubsCons: any #constraint for opening a fixed number of hubs
    deliverFullRequestsConss: defaultdict(dict) #constraint for the full delivery of the requests' demand
    numVehiclesConss: dict #constraint for guaranteeing the presence of all the necessary vehicles through an intrahubs arc
    openHubConss: defaultdict(dict) #constraint for opening a specific hub if an arcs' route passes through it

@dataclass
#class of the arc-based model with unsplittable requests (subclass of the class arcBasedModel)
class unsplittableRequestsArcBasedModel(arcBasedModel):
```

```
        numHopsConss: dict #constraint for the maximum number of transshipments
        maxTransportTimeConss: dict #constraint for the maximum transport time

    @dataclass
    #class of the arc-based model with splittable requests (subclass of the class arcBasedModel)
    class splittableRequestsArcBasedModel(arcBasedModel):
        e: defaultdict(dict) #binary request arcs variables indicating wheter or not the request r is transported over arc a
        s: defaultdict(dict) #variables counting number of ships of the part of request r up to the hub i
        w: defaultdict(dict) #variables representing the transport time of the part of request r until the hub i
        startHubsNumHopsConss: defaultdict(dict) #constraint for maximum number of transshipments in the first leg of trip
        transferArcsNumHopsConss: defaultdict(dict) #constraint for maximum number of transshipments in the hubs network
        endHubsNumHopsConss: defaultdict(dict) #constraint for maximum number of transshipments in the last leg of trip (the actual max number of transshipments, but it depends on the previous two)
        startHubsTransportTimeConss: defaultdict(dict) #constraint for maximum transport time in the first leg of trip
        transferArcsTransportTimeConss: defaultdict(dict) #constraint for maximum transport time in the hubs network
        endHubsTransportTimeConss: defaultdict(dict) #constraint for maximum transport time in the last leg of trip (the actual max transport time, but it depends on the previous two)

    #this creates the SNDHLP model of the compact arc-based problem
    def createArcBasedSNDHLPmodel(sndhlp, splittableRequests):
        modelName = "UnsplittableRequestsArcBasedSNDHLP"
        xVarType = "B"
        if splittableRequests:
            modelName = "splittableRequestsArcBasedSNDHLP"
            xVarType = "C"

    #here we create the model
        master = Model(modelName)
    #initialize containers for the master variables
        h = {}
        x = defaultdict(dict)
        v = {}
    #initialize containers for the master constraints
        numHubsCons = None
        deliverFullRequestsConss = defaultdict(dict)
        numVehiclesConss = {}
```

```
        openHubConss = defaultdict(dict)

    #create hub variables
        for i in sndhlp.hubs:
            h[i] = master.addVar(vtype="B", name=f"h({i})")

    #create request arcs variables
        for r in sndhlp.requests.keys():
            for a in list(sndhlp.requestsGraph.get(r).edges):
                x[r][a] = master.addVar(vtype=xVarType, name=f"x({r},{a})")

    #create number of vehicles per arc variables
        for a in sndhlp.intraHubsArcs.keys():
            v[a] = master.addVar(vtype="I", name=f"v({a})")

    #constraint of number of hubs
        numHubsCons = master.addCons(
            quicksum(h[i] for i in sndhlp.hubs) == sndhlp.numberHubs,
            name="numHubsCons")

    #constraint of sum of split requests to guarantee the full delivery of the demand
        for r in sndhlp.requests.keys():
            for node in list(sndhlp.requestsGraph.get(r).nodes):
    #we compute the arcs' flow in the node given by the difference between outgoing arcs and ingoing arcs
                outgoingArcs = quicksum(x[r][(node, succ)] for succ in list(sndhlp.requestsGraph.get(r).successors(node)))
                ingoingArcs = quicksum(x[r][(pred, node)] for pred in list(sndhlp.requestsGraph.get(r).predecessors(node)))
                arcsFlow = outgoingArcs - ingoingArcs
    #this arcs' flow must be equal to 0 if the node is an hub, to 1 if it is the customer origin, and to -1 if it is the customer destination
                arcsFlowValue = 0.0
                if node == r[0]:
                    arcsFlowValue = 1.0
                elif node == r[1]:
                    arcsFlowValue = -1.0
                deliverFullRequestsConss[r][node] = master.addCons(
                    arcsFlow == arcsFlowValue,
                    name=f"deliverFullRequestsCons_{r}_{node}")

    #constraint of number of vehicles per intra-hubs arcs
        for a in sndhlp.intraHubsArcs.keys():
            requestParts = quicksum(sndhlp.requests.get(r)*x[r][a] for r in sndhlp.requests.keys())
            numVehiclesConss[a] = master.addCons(
```

```
                    requestParts <= (sndhlp.vehicleCapacity*v[a]),
                    name=f"numVehiclesCons_{a}")

    #constraint of open hub if an arcs' route passes through it
        for i in sndhlp.hubs:
            for r in sndhlp.requests.keys():
                openHubConss[i][r] = master.addCons(
                    -h[i] + quicksum(x[r][(pred, i)] for pred in list
(sndhlp.requestsGraph.get(r).predecessors(i))) <= 0,
                    name=f"openHubCons_{i}_{r}")

    #objective function:
        transferVehiclesCosts = quicksum(sndhlp.intraHubsCost*sndhlp.
intraHubsArcs.get(a)*v[a] for a in sndhlp.intraHubsArcs.keys())
        externalHubsNetworkCosts = quicksum(sndhlp.accessCost*sndhlp.
accessArcs.get(a)*sndhlp.requests.get(r)*x[r][a] for r in sndhlp.
requests.keys() for a in sndhlp.accessArcs.keys() if (a[0]==r[0]
or a[1]==r[1]))
        objectiveFunction = transferVehiclesCosts +
externalHubsNetworkCosts
        master.setObjective(objectiveFunction)

        arcBasedSNDHLP = arcBasedModel(master, h, x, v, numHubsCons,
deliverFullRequestsConss, numVehiclesConss, openHubConss)

        if splittableRequests:
            splittableRequestsArcBasedSNDHLP =
createSplittableRequestsArcBasedSNDHLPmodel(sndhlp, arcBasedSNDHLP
)
            return splittableRequestsArcBasedSNDHLP
        else:
            unsplittableRequestsArcBasedSNDHLP =
createUnsplittableRequestsArcBasedSNDHLPmodel(sndhlp,
arcBasedSNDHLP)
            return unsplittableRequestsArcBasedSNDHLP

    #this method adds the constraints of the arc-based model for
unsplittable requests
    def createUnsplittableRequestsArcBasedSNDHLPmodel(sndhlp,
arcBasedSNDHLP):
    #initialize containers for unsplittable requests constraints
        numHopsConss = {}
        maxTransportTimeConss = {}

    #constraint of maximum number of transshipments per request
        for r in sndhlp.requests.keys():
            numHopsConss[r] = arcBasedSNDHLP.master.addCons(
                quicksum(arcBasedSNDHLP.x[r][a] for a in sndhlp.
requestsGraph.get(r).edges) <= sndhlp.numberHops + 1,
```

```
                name=f"maxTransshipmentsCons_{r}"
            )

    #constraint of maximum time of transport per request
        for r in sndhlp.requests.keys():
            maxTransportTimeConss[r] = arcBasedSNDHLP.master.addCons(
                quicksum(sndhlp.requestsGraph.get(r)[a[0]][a[1]]["res_cost"][1]*arcBasedSNDHLP.x[r][a] for a in sndhlp.requestsGraph.get(r).edges) <= sndhlp.requestsMaxTransportTime.get(r),
                name=f"maxTransportTimeCons_{r}"
            )

        unsplittableRequestsArcBasedSNDHLP = unsplittableRequestsArcBasedModel(arcBasedSNDHLP, numHopsConss, maxTransportTimeConss)
        return unsplittableRequestsArcBasedSNDHLP

    #this method adds the variables and the constraints of the arc-based model for splittable requests
    def createSplittableRequestsArcBasedSNDHLPmodel(sndhlp, arcBasedSNDHLP):
    #initialize containers for the splittable requests variables
        e = defaultdict(dict)
        s = defaultdict(dict)
        w = defaultdict(dict)
    #initialize containers for the splittable requests constraints
        startHubsNumHopsConss = defaultdict(dict)
        transferArcsNumHopsConss = defaultdict(dict)
        endHubsNumHopsConss = defaultdict(dict)
        startHubsTransportTimeConss = defaultdict(dict)
        transferArcsTransportTimeConss = defaultdict(dict)
        endHubsTransportTimeConss = defaultdict(dict)

    #create corresponding binary request arcs variables e[r][a], with the relative linking constraint x[r][a] <= e[r][a]
        for r in sndhlp.requests.keys():
            for a in list(sndhlp.requestsGraph.get(r).edges):
                e[r][a] = arcBasedSNDHLP.master.addVar(vtype="B", name=f"e({r},{a})")
                arcBasedSNDHLP.master.addCons(arcBasedSNDHLP.x[r][a] <= e[r][a], name=f"arcsVariablesCons_{r}_{a}")

    #create the variables for managing the constraints of number of transshipments and transport time in the hubs
        for r in sndhlp.requests.keys():
            for i in sndhlp.hubs:
                s[r][i] = arcBasedSNDHLP.master.addVar(vtype="I", name=f"s({r},{i})")
```

```
                w[r][i] = arcBasedSNDHLP.master.addVar(vtype="C", name=f"w({r},{i})")

    #constraints of maximum number of transshipments per request
        for r in sndhlp.requests.keys():
            for a in sndhlp.accessArcs.keys():
                if a[0]==r[0]:
                    startHubsNumHopsConss[r][a[1]] = arcBasedSNDHLP.master.addCons(
                        s[r][a[1]] >= e[r][a],
                        name=f"startHubMaxTransshipmentsCons_{r}_{a[1]}"
                    )
                if a[1]==r[1]:
                    endHubsNumHopsConss[r][a[0]] = arcBasedSNDHLP.master.addCons(
                        s[r][a[0]] <= sndhlp.numberHops,
                        name=f"endHubMaxTransshipmentsCons_{r}_{a[0]}"
                    )
            for a in sndhlp.intraHubsArcs.keys():
                transferArcsNumHopsConss[r][a] = arcBasedSNDHLP.master.addCons(
                    s[r][a[0]] + e[r][a] <= s[r][a[1]] + sndhlp.numberHops*(1 - e[r][a]),
                    name=f"transferArcMaxTransshipmentsCons_{r}_{a}"
                )

    #constraints of maximum time of transport per request
        for r in sndhlp.requests.keys():
            for a in sndhlp.accessArcs.keys():
                if a[0]==r[0]:
                    startHubsTransportTimeConss[r][a[1]] = arcBasedSNDHLP.master.addCons(
                        w[r][a[1]] >= e[r][a]*sndhlp.requestsGraph.get(r)[a[0]][a[1]]["res_cost"][1],
                        name=f"startHubMaxTransportTimeCons_{r}_{a[1]}"
                    )
                if a[1]==r[1]:
                    endHubsTransportTimeConss[r][a[0]] = arcBasedSNDHLP.master.addCons(
                        w[r][a[0]] + e[r][a]*sndhlp.requestsGraph.get(r)[a[0]][a[1]]["res_cost"][1] <= sndhlp.requestsMaxTransportTime.get(r),
                        name=f"endHubMaxTransportTimeCons_{r}_{a[0]}"
                    )
            for a in sndhlp.intraHubsArcs.keys():
```

```
                arcTime = sndhlp.requestsGraph.get(r)[a[0]][a[1]]["res_cost"][1]
                transferArcsTransportTimeConss[r][a] = arcBasedSNDHLP.master.addCons(
                    w[r][a[0]] + e[r][a]*arcTime <= w[r][a[1]] + sndhlp.requestsMaxTransportTime.get(r)*(1 - e[r][a]),
                    name=f"transferArcMaxTransportTimeCons_{r}_{a}"
                )

    splittableRequestsArcBasedSNDHLP = splittableRequestsArcBasedModel(arcBasedSNDHLP, e, s, w, startHubsNumHopsConss, transferArcsNumHopsConss, endHubsNumHopsConss, startHubsTransportTimeConss, transferArcsTransportTimeConss, endHubsTransportTimeConss)
    return splittableRequestsArcBasedSNDHLP
```

A.4 Path-based SNDHLP

This section displays the code of the path-based formulation. Note that in the method **createPathBasedSNDHLPinstance** are presented some tools that will be useful also for the following appendix codes.

```
from dataclasses import dataclass
from pyscipopt import Model, quicksum
from collections import defaultdict
from instanceReaderSNDHLP import readSNDHLPfromFile
from SNDHLPmodelFunctions import *

@dataclass
class sndhlpPathBasedInstance:
    customers: set[int] # subset of vertices (simply referred to by an integer id) that can be only origin or destination
    hubs: set[int] # subset of vertices that are hubs
    accessArcs: dict[Tuple[int, int], float] # dictionary representing the subset of arcs whose key is an arc a (identified by a tuple with the two nodes) which connects an origin or a destination to the hubs' network and whose value is its length
    intraHubsArcs: dict[Tuple[int, int], float] # dictonary representing the subset of arcs whose key is an intraHub arc a (identified by a tuple with the two hubs) which links two hubs and whose value is its length
    requests: dict[Tuple[int, int], float] #  dictionary mapping customer pairs to demand values
    requestsFeasiblePaths: dict[Tuple[int, int], list[Path]] # dictionary mapping customer pairs to all possible paths from the customer origin to the destination customer
    requestsHubFeasiblePaths: dict[Tuple[int, int], dict[int, list[Path]]] #  dictionary mapping customer pairs to the dictionary of all possible paths containing the specific hub i (this second dictionary maps each hub i to all possible paths containing it)
    requestsArcFeasiblePaths: dict[Tuple[int, int], dict[Tuple[int, int], list[Path]]] #  dictionary mapping customer pairs to the dictionary of all possible paths containing the specific arc a (this second dictionary maps each arc a to all possible paths containing it)
    accessCost: float # cost per kilometer and demand unit for the subset of accessArcs
    intraHubsCost: float # cost per kilometer and vehicle for the subset of intraHubsArcs
    vehicleCapacity: float # given capacity of the means of transportation
    numberHubs: int # given number of hubs that have to be used
```

```
    numberHops: int # maximal number of transshipments at hubs
    requestsMaxTransportTime: dict[Tuple[int, int], float] # dictionary mapping customer pairs to their maximum time allowed for deliver a request

#this creates a complete path-based SNDHLP instance
def createPathBasedSNDHLPinstance(fileName, accessCost, intraHubsCost, vehicleCapacity, numHubs, numPaths, numHops, maxTransportTime, heuristicApproach, CGmodel):
#here we create the useful sets by reading a generic instance file
    customers, hubs, accessArcs, intraHubsArcs, requests, numberHubs = readSNDHLPfromFile(fileName)
#this is to check if the input is okay with the content of the file and eventually correct it by imposing the file input
    if(numHubs > numberHubs):
        numHubs = numberHubs

#here we create the graph with hubs and their links and the "subgraph" that have only the starting open hubs generated from the auxiliary method or from the heuristic approach
    hubsGraph = createHubsGraph(hubs, intraHubsArcs)
    if CGmodel or heuristicApproach != "":
        heuristicOpenHubs = getHeuristicOpenHubs(fileName, numHubs, heuristicApproach) #to have the starting open hubs of a possible feasible solutions of the model, given the specified heuristic approach
        heuristicOpenHubsGraph = generateOpenHubsGraph(hubsGraph, heuristicOpenHubs)

    requestsMaxTransportTime = dict() #dictionary for the maximum transport time of each request
#then we create for each request the set of feasible paths, and the 2 sets of feasible paths that contain a specific hub or arc
    requestsFeasiblePaths = dict()
    requestsHubFeasiblePaths = dict()
    requestsArcFeasiblePaths = dict()
    for r in requests.keys():
#as customers in the request are of the type "100h" where h is the number of the corresponding hub, we remove the first occurrence of "100" (for avoid problems with cases like 100100)
        originHub = int(str(r[0]).replace("100","", 1)) #this gives the corresponding same location hub of the origin customer of the request
        destinationHub = int(str(r[1]).replace("100","", 1)) #this gives the corresponding same location hub of the destination customer of the request
```

```
            requestMaxTransportTime = maxTransportTime + 1.5*intraHubsArcs.get((originHub, destinationHub)) #maximum transport time of the request
            requestsMaxTransportTime[r] = requestMaxTransportTime

            maxPathLength = numHops + 1
    #if CGmodel is "False" then the 1st input is hubsGraph and the 4th one is hubs, and we are generating all the possible feasible paths with maximum length equal to numHops+1 of a specific request. However, if the input value heuristicApproach is not an empty string, then the 1st and 4th input are the heuristicOpenHubsGraph and heuristicOpenGraph generated by the applied heuristic approach, and we generate all the possible feasible paths with that maximum length containing only those open hubs.
    #else if it is "True" we want to obtain a starting feasible solution for the column generation algorithm, and these input are heuristicOpenHubsGraph and heuristicOpenHubs, so we are generating a subset of the cheapest feasible paths containing only the starting opened hubs
            if CGmodel == False:
                if heuristicApproach == "":
                    rFeasiblePaths = getFeasiblePaths(hubsGraph, r[0], r[1], hubs, accessArcs, accessCost, intraHubsArcs, intraHubsCost, vehicleCapacity, maxPathLength, requestMaxTransportTime)
                else:
                    rFeasiblePaths = getFeasiblePaths(heuristicOpenHubsGraph, r[0], r[1], heuristicOpenHubs, accessArcs, accessCost, intraHubsArcs, intraHubsCost, vehicleCapacity, maxPathLength, requestMaxTransportTime)
            else:
                rFeasiblePaths = getFeasiblePaths(heuristicOpenHubsGraph, r[0], r[1], heuristicOpenHubs, accessArcs, accessCost, intraHubsArcs, intraHubsCost, vehicleCapacity, maxPathLength, requestMaxTransportTime)
                rCheapestPaths = getCheapestPaths(rFeasiblePaths, numPaths) #to obtain the subset of the request cheapest feasible paths
                rFeasiblePaths = rCheapestPaths.copy()

            rHubFeasiblePaths = getHubsFeasiblePaths(rFeasiblePaths, hubs)
            rArcFeasiblePaths = getArcsFeasiblePaths(rFeasiblePaths, intraHubsArcs)
            requestsFeasiblePaths[r] = rFeasiblePaths
            requestsHubFeasiblePaths[r] = rHubFeasiblePaths
            requestsArcFeasiblePaths[r] = rArcFeasiblePaths

    #here we create the SNDHLP object, useful for the model creation
```

```
        sndhlp = sndhlpPathBasedInstance(customers, hubs, accessArcs, intraHubsArcs, requests, requestsFeasiblePaths, requestsHubFeasiblePaths, requestsArcFeasiblePaths, accessCost, intraHubsCost, vehicleCapacity, numHubs, numHops, requestsMaxTransportTime)
        return sndhlp

    @dataclass
    class pathBasedModel:
        master: Model
        y: defaultdict(dict) #request paths variables representing the percentage of the request r transported by path p (dictionaries that for each request have a dictionary mapping request to a path)
        h: dict #hub variables
        v: dict #number of vehicles per transfer arc variables
        numHubsCons: any #constraint for opening a fixed number of hubs
        deliverFullRequestsConss: defaultdict(dict) #constraint for the full delivery of the requests' demand
        numVehiclesConss: dict #constraint for guaranteeing the presence of all the necessary vehicles through an intrahubs arc
        openHubConss: defaultdict(dict) #constraint for opening a specific hub if a path passes through it

    #this creates the SNDHLP model of the path-based master problem
    def createPathBasedSNDHLPmodel(sndhlp, splittableRequests):
        modelName = "UnsplittableRequestsPathBasedSNDHLP"
        yVarType = "I"
        if splittableRequests:
            modelName = "splittableRequestsPathBasedSNDHLP"
            yVarType = "C"

        master = Model(modelName)

    #initialize containers for the master variables
        h = {}
        y = defaultdict(dict)
        v = {}
    #initialize containers for the master constraints
        numHubsCons = None
        deliverFullRequestsConss = {}
        numVehiclesConss = {}
        openHubConss = defaultdict(dict)

    #create hub variables
        for i in sndhlp.hubs:
            h[i] = master.addVar(vtype="B", name=f"h({i})")

```

```python
    #create request paths variables, if the input splittableRequests is True they will be continuous otherwise binary
        for r in sndhlp.requests.keys():
            for p in sndhlp.requestsFeasiblePaths.get(r):
                y[r][p.pathString] = master.addVar(vtype=yVarType, name=f"y({r}, {p.pathString})")

    #create number of vehicles per arc variables
        for a in sndhlp.intraHubsArcs.keys():
            v[a] = master.addVar(vtype="I", name=f"v({a})")

    #constraint of number of hubs
        numHubsCons = master.addCons(
            quicksum(h[i] for i in sndhlp.hubs) == sndhlp.numberHubs,
            name="numHubsCons",
            separate=False,
            modifiable=True)

    #constraint of sum of requests parts equal to 1 to guarantee the full delivery of the demand
        for r in sndhlp.requests.keys():
            deliverFullRequestsConss[r] = master.addCons(
                quicksum(y[r][p.pathString] for p in sndhlp.requestsFeasiblePaths.get(r)) == 1.0,
                name=f"deliverFullRequestsCons_{r}",
                separate=False,
                modifiable=True)

    #constraint of number of vehicles per intra-hubs arcs
        for a in sndhlp.intraHubsArcs.keys():
            requestParts = quicksum(sndhlp.requests.get(r)*y[r][p.pathString] for r in sndhlp.requests.keys() if sndhlp.requestsArcFeasiblePaths.get(r).get(a) is not None for p in sndhlp.requestsArcFeasiblePaths.get(r).get(a))
            numVehiclesConss[a] = master.addCons(
                requestParts <= (sndhlp.vehicleCapacity*v[a]),
                name=f"numVehiclesCons_{a}",
                separate=False,
                modifiable=True)

    #constraint of open hub if a path passes through it
        for i in sndhlp.hubs:
            for r in sndhlp.requests.keys():
                if sndhlp.requestsHubFeasiblePaths.get(r).get(i) is not None:
                    openHubConss[i][r] = master.addCons(
                        -h[i] + quicksum(y[r][p.pathString] for p in sndhlp.requestsHubFeasiblePaths.get(r).get(i)) <= 0,
                        name=f"openHubCons_{i}_{r}",
```

```
                                separate=False,
                                modifiable=True)
                    else:
                        openHubConss[i][r] = master.addCons(
                            -h[i] + quicksum(0 for p in range(0)) <= 0,
                            name=f"openHubCons_{i}_{r}",
                            separate=False,
                            modifiable=True)

    #objective function:
        transferVehiclesCosts = quicksum(sndhlp.intraHubsCost*sndhlp.
    intraHubsArcs.get(a)*v[a] for a in sndhlp.intraHubsArcs.keys())
        externalHubsNetworkCosts = quicksum(sndhlp.accessCost*sndhlp.
    accessArcs.get(a)*sndhlp.requests.get(r)*y[r][p.pathString] for r
    in sndhlp.requests.keys() for p in sndhlp.requestsFeasiblePaths.
    get(r) for a in p.extremeArcs)
        objectiveFunction = transferVehiclesCosts +
    externalHubsNetworkCosts
        master.setObjective(objectiveFunction)

        pathModel = pathBasedModel(master, y, h, v, numHubsCons,
    deliverFullRequestsConss, numVehiclesConss, openHubConss)
        return pathModel
```

Appendix B

Heuristics

Every heuristic method and the auxiliary problem for creating the hubs set of the RMP is based on the following model. In the next sections, we will present the code for obtaining the relative objective function of the specific heuristic approach.

```
from pyscipopt import Model, quicksum
from instanceReaderSNDHLP import readSNDHLPfromFile

#this method creates a model to select the most promising set of hubs, based on the heuristic approach chosen or on the auxiliary problem, and is eventually useful to obtain the starting open hubs of the SNDHLP model for applying the column generation algorithm
def createSNDHLPheuristicModel(fileName, numHubs, heuristicApproach):
#read data from instance file and check the input numHubs value
    customers, hubs, accessArcs, intraHubsArcs, requests, numberHubs = readSNDHLPfromFile(fileName)
    if(numHubs > numberHubs):
        numHubs = numberHubs

    model = Model(heuristicApproach + "SNDHLP") #name based on the input heuristicApproach

#create hub variables
    h = {}
    for i in hubs:
        h[i] = model.addVar(vtype="B", name="h(%i)" % (i))

#constraint of number of hubs
    model.addCons(quicksum(h[i] for i in hubs) == numHubs)

#constraint of open customer allowed hubs for each customer
    for customer in customers:
```

```
            model.addCons(quicksum(h[i] for i in getCustomerAllowedHubs(customer, accessArcs)) >= 1)

    #objective function
        objectiveFunction = quicksum(h[i]*setHubObjValue(i, accessArcs, requests, heuristicApproach) for i in hubs)
        model.setObjective(-objectiveFunction) #the minus is because they are maximization problems (a part from the ShortestAccessArcs one)

        return model

    #this method returns the set of all the hubs linked with a specific customer through an access arc (customer allowed hubs)
    def getCustomerAllowedHubs(customer, accessArcs):
        customerHubs = set()
        for arc in accessArcs.keys():
            if arc[0] == customer:
                customerHubs.add(arc[1])
            elif arc[1] == customer:
                customerHubs.add(arc[0])
        return customerHubs

    #this method sets the corresponding value in the objective function of the hub variable, based on the name of the input heuristicApproach (if the name is wrong, it imposes the value to 0 as if it was solving the auxiliary problem)
    def setHubObjValue(hub, accessArcs, requests, heuristicApproach):
        objValue = 0
        if heuristicApproach == "MostAccessedHubs":
            objValue = getNumIngoingAccessArcs(hub, accessArcs)
        elif heuristicApproach == "GreatestDemandRequests":
            objValue = getHubDemandsPriority(hub, requests, accessArcs)
        elif heuristicApproach == "AdditiveGreatestDemandRequests":
            objValue = getHubAdditiveDemandsPriority(hub, requests, accessArcs)
        elif heuristicApproach == "ShortestAccessArcs":
            objValue = - getHubDistancePriority(hub, accessArcs) #this is the only minimization problem of the four and so the minus
        return objValue
```

B.1 Most Accessed Hubs Heuristic

```
#this method gives the number of ingoing access arcs in the hub
def getNumIngoingAccessArcs(hub, accessArcs):
    numArcs = 0
    for arc in accessArcs.keys():
        if arc[1] == hub:
            numArcs += 1
    return numArcs
```

B.2 Greatest Demand Requests Heuristic

```
#this method returns the objective value of a specific hub derived from the descending number of requests' demand units. The hub priority will be 0 or the number of the last occurence in which that hub is an access hub for the request
def getHubDemandsPriority(hub, requests, accessArcs):
    hubPriority = 0
    sortedRequests = dict(sorted(requests.items(), key=lambda x:x[1])) #sorting requests for ascending number of demand units
    requestPriority = 0
    for r in sortedRequests.keys():
        requestPriority += 1
        for arc in accessArcs.keys():
            if arc[0] == r[0] or arc[1] == r[1]:
                if hub in arc:
                    hubPriority = requestPriority
    return hubPriority
```

B.3 Additive Greatest Demand Requests Heuristic

```
#this method is similar to the previous one, as it returns the objective value of a specific hub derived from the descending number of requests' demand units, but in this case, this value will be 0 or the sum of every number of the corresponding occurence in which that hub is an access hub for the request
def getHubAdditiveDemandsPriority(hub, requests, accessArcs):
    hubAdditivePriority = 0
    sortedRequests = dict(sorted(requests.items(), key=lambda x:x[1])) #sorting requests for ascending number of demand units
    requestPriority = 0
    for r in sortedRequests.keys():
        requestPriority += 1
        for arc in accessArcs.keys():
            if arc[0] == r[0] or arc[1] == r[1]:
                if hub in arc:
                    hubAdditivePriority += requestPriority
    return hubAdditivePriority
```

B.4 Shortest Access Arcs Heuristic

```
#this method gives the objective value of a specific hub derived from the weighted distances from the customer nodes. Indeed, this value will be 10^9 (big M) or the average length of the outgoing access arcs from the hub (equal to the ratio between the sum of distances of each outgoing access arc and the number of outgoing access arcs of the hub)
def getHubDistancePriority(hub, accessArcs):
    hubDistancePriority = 1.0e10 #big integer number as initialization
    numArcs = 0
    hubArcsDistance = 0.0
    for (arc, distance) in accessArcs.items():
        if arc[0] == hub:
            numArcs += 1
            hubArcsDistance += distance
    if numArcs != 0:
        hubDistancePriority = float(hubArcsDistance/numArcs)
    return hubDistancePriority
```

Appendix C

Branch-and-Price algorithm

In this chapter is presented the code useful for the Branch-and-Price algorithm.

C.1 Restricted Master Problem

Here is reported the methods that obtain the restriction on the feasible requests paths sets. They are used in the method **createPathBasedSNDHLPinstance** presented in appendix A.4 when the parameter *CGmodel* is True. After having created the *openHubsGraph* from the method **generateOpenHubsGraph**, that takes as input parameter the *heuristicOpenHubs* from method **getHeuristicOpenHubs**, we create only paths from that graph, and then we restrict the number of these to a maximum of 5.

```
#this method obtains, if the SNDHLP is feasible, one of the possible sets of open hubs by solving the auxiliary SNDHLP model (the eventual heuristic method for the objective function is given in input)
def getHeuristicOpenHubs(fileName, numHubs, heuristicApproach):
    openHubs = set()
    heuristicModel = createSNDHLPheuristicModel(fileName, numHubs, heuristicApproach)
    heuristicModel.optimize()
    if heuristicModel.getStatus() != "infeasible":
        for var in list(heuristicModel.getVars()):
            if heuristicModel.getVal(var) == 1.0:
                strHub = str(var).replace("h(","")
                hub = int(strHub.replace(")",""))
                openHubs.add(hub)
    return openHubs

```

```
    #this method creates a new graph from the original hubs graph that contains only the hubs that are open in a possible feasible solution
    def generateOpenHubsGraph(hubsGraph, openHubs):
        openHubsGraph = hubsGraph.copy()
        closedHubs = []
        for hub in hubsGraph.nodes:
            if hub not in openHubs:
                closedHubs.append(hub)
        openHubsGraph.remove_nodes_from(closedHubs)
        return openHubsGraph

    #this method gives the n cheapest feasible paths of a specific request (where n is the input parameter numPaths)
    def getCheapestPaths(requestFeasiblePaths, numPaths):
        cheapestPaths = []
        requestFeasiblePaths.sort(key=lambda p: p.approximateTotalCost) #sort the list by the approximate total cost of the paths
        j = 0
        for path in requestFeasiblePaths:
            cheapestPaths.append(path)
            j += 1
            if j == numPaths:
                break
        return cheapestPaths
```

C.2 Pricing Problem

The pricing problem is solved, thanks to an auxiliary Python library called *cspy* which automatically solves the resources-constrained shortest path problem. Note that to implement the pricing procedure, it is necessary to include the Pricer class in the master problem before its optimization.

```python
from dataclasses import dataclass
from collections import defaultdict
from pyscipopt import Pricer, SCIP_RESULT
from cspy import BiDirectional
import networkx as nx
from SNDHLPmodelFunctions import Path, sndhlpPathBasedInstance, createRequestGraph

#the following method creates the pricer including it in the model and initializing its variables and constraints to the ones of the model
def pricerInitialization(pathBasedModel, sndhlp, splittableRequests):

    hubsGraph = createHubsGraph(sndhlp.hubs, sndhlp.intraHubsArcs)
#creating and including in the master problem the pricer from the PricerSNDHLP class
    pricer = PricerSNDHLP(sndhlp, hubsGraph, splittableRequests)
    pathBasedModel.master.includePricer(pricer, "PricerSNDHLP", "Pricer to identify new paths", delay=True)

#master variables
    pricer.y = pathBasedModel.y
    pricer.h = pathBasedModel.h
    pricer.v = pathBasedModel.v
#master constraints that can be modified during the pricing iterations
    pricer.numHubsCons = pathBasedModel.numHubsCons
    pricer.deliverFullRequestsConss = pathBasedModel.deliverFullRequestsConss
    pricer.numVehiclesConss = pathBasedModel.numVehiclesConss
    pricer.openHubConss = pathBasedModel.openHubConss

EPS = 1.0e-6 #defined infinitesimal value to compare to the reduced cost of a possible new variable in order to avoid rounding problems if compared with 0

@dataclass
class PricerSNDHLP(Pricer):
```

```
        sndhlp: sndhlpPathBasedInstance
        hubsGraph: nx.DiGraph
        yVarType: str

        def __init__(self, sndhlp, hubsGraph, splittableRequests):
            self.sndhlp = sndhlp
            self.hubsGraph = hubsGraph
            if splittableRequests:
                self.yVarType = "C"
            else:
                self.yVarType = "I"

            self.pricingIterations = 0 #variable counting how many iterations the pricing does

    #master variables
            self.y = defaultdict(dict)
            self.h = {}
            self.v = {}
    #master constraints
            self.numHubsCons = None
            self.deliverFullRequestsConss = {}
            self.numVehiclesConss = {}
            self.openHubConss = defaultdict(dict)

    #branching rules
            self.forbiddenHubs = []
            self.forbiddenIntraHubsArcs = []

    #early branching tools
            self.lowerbound = None #useful to compute the lagrangian gap necessary for the early branching procedure
            self.earlyBranchingNodes = set() #set of the B&B nodes that do not need anymore pricing iterations because their lagrangian gap is lower than the early branching threshold
            self.earlyBranchingThreshold = 0.05 #threshold for the early branching implementation, to compare with the node lagrangian gap

    #method for adding a column with a negative reduced cost to the master problem by adding the new variable and modifying its relative constraints
        def addColumn(self, path, request, reqDemand, requestGraph):
    #value of the new variable in the objective function
            extremeArcsCost = sum(reqDemand*self.sndhlp.accessCost*requestGraph[arc[0]][arc[1]]["distance"] for arc in path.extremeArcs)
    #create the new variable to add to the master problem
```

```python
            self.y[request][path.pathString] = self.model.addVar(name=f"y({request}, {path.pathString})", vtype=self.yVarType, obj=extremeArcsCost, pricedVar=True)

    #adding the new variable created to the master problem constraints
            self.model.addConsCoeff(self.deliverFullRequestsConss[request], self.y[request][path.pathString], 1)
            for a in path.intraHubsArcs:
                self.model.addConsCoeff(self.numVehiclesConss[a], self.y[request][path.pathString], reqDemand)
            for i in path.hubs:
                self.model.addConsCoeff(self.openHubConss[i][request], self.y[request][path.pathString], 1)

            return {"result": SCIP_RESULT.SUCCESS}

    #method for performing a pricing iteration
        def performPricing(self):
            self.pricingIterations += 1

    #early branching exit condition: when the current node is in the set, the pricing is not performed and SCIP skips to the next B&B node to be analyzed
            currentNode = self.model.getCurrentNode().getNumber()
            if currentNode in self.earlyBranchingNodes:
                return {"result": SCIP_RESULT.DIDNOTRUN, "stopearly": True}

    #early branching initialization: we retrieve the objective value of the current B&B node, and then we create the relative node lower bound by initially imposing it equal to the node objective value
            zRMP = self.model.getLPObjVal()
            self.lowerbound = zRMP

    #pricing problem for each request during the same pricing iteration
            for (request, reqDemand) in self.sndhlp.requests.items():
    #these are the 2 resources constraints for minimum and maximum number of hops and time to delivery for the request
                min_res = [0, 0] #minimum resources set both to 0 for simplicity (direct transportation is not possible thanks to the problem construction)
                max_res = [(self.sndhlp.numberHops + 1 + 2), self.sndhlp.requestsMaxTransportTime[request]] #fix the maximum number of arcs (equal to number of hops plus one (+2 for the 2 extra arcs from "Source" and to "Sink")) and transport time

```

```
            origin = request[0]
            destination = request[1]
#the requestGraph is the "original" hubsGraph + origin, destination and their links to their allowed hubs (accessArcs)
            requestGraph = createRequestGraph(self.hubsGraph, origin, destination, self.sndhlp.hubs, self.sndhlp.accessArcs)

#the pricerRequestGraph is a graph derived from the requestGraph but with "dynamic" weights related to dual values of primal constraints, and the two extra nodes "Source" and "Sink" necessary for the cspy search algorithm
            pricerRequestGraph = self.setPricerRequestGraph(origin, destination, reqDemand)

#this cspy algorithm solves the shortest path problem with resources constraints and finds the cheapest path that respects those constraints
            algorithm = BiDirectional(pricerRequestGraph, max_res, min_res)
            algorithm.run()
            if algorithm.path is not None:
                pathLength = int(algorithm.consumed_resources[0])
                pathCost = algorithm.total_cost
                p = algorithm.path[1:pathLength]

                path = Path(p, requestGraph, self.sndhlp.accessCost, self.sndhlp.intraHubsCost, self.sndhlp.vehicleCapacity)
#we compute the reduced cost of the found path to check if it is negative and eventually add the column to the master problem and update the dictionaries of feasible paths, and sum this to the lower bound
                reducedCost = pathCost - self.model.getDualsolLinear(self.deliverFullRequestsConss[request])
                if reducedCost < -EPS:
                    self.addColumn(path, request, reqDemand, requestGraph)
                    self.updateFeasiblePaths(path, request)
                    self.lowerbound += reducedCost

#after all the requests have been analyzed in the pricing iteration, we compute the lagrangian bound equal to the lower bound (which was modified over all the requests), and the corresponding lagrangian gap equal to the ratio between the difference of the current node objective value and the lagrangian bound, and the lagrangian bound itself
        LAGRANGE_BOUND = self.lowerbound
        lagrangeGap = round(((zRMP-LAGRANGE_BOUND)/LAGRANGE_BOUND), 4)
```

```
#then if the lagrangian gap is lower than the earlyBranching threshold, we add the current node to the relative dictionary of early branching nodes
        if (lagrangeGap >= 0.0) and (lagrangeGap < self.earlyBranchingThreshold):
            self.earlyBranchingNodes.add(currentNode)
            return {"result": SCIP_RESULT.SUCCESS, "lowerbound": LAGRANGE_BOUND, "stopearly": True}
        else:
            return {"result": SCIP_RESULT.SUCCESS, "lowerbound": LAGRANGE_BOUND}

#this method sets the request graph for the pricer with its arc that have a current cost based on the current values of variables
    def setPricerRequestGraph(self, origin, destination, requestDemand):
        request = (origin, destination)

#for each intra-hubs arc we compute its current cost equal to the negative sum of the dual cost for using its hub for a specific request and the dual cost of using a vehicle times the request demand
        requestGraph = self.hubsGraph.copy()
        for hub in requestGraph.nodes:
            for pred in list(requestGraph.predecessors(hub)):
                dualIntraHubsArcVehiclesCost = self.model.getDualsolLinear(self.numVehiclesConss[(pred, hub)])
                dualHubUsageCost = self.model.getDualsolLinear(self.openHubConss[hub][request])
                arcWeight = - dualHubUsageCost - dualIntraHubsArcVehiclesCost*requestDemand
                requestGraph[pred][hub]["weight"] = max(arcWeight, 0)

#for each access arc we create the edge in the request graph which has a current cost equal to distance times access arc cost per kilometer times the request demand, and if the arc starts from the origin (and so arrives in a hub) we have to subtract the dual hub usage cost
        for (arc, distance) in self.sndhlp.accessArcs.items():
            if arc[0] == origin or arc[1] == destination:
                arcTime = distance*0.75 #time in minutes to cover the distance between the customer and the hub (assumed equal to 0.75 times the distance)
                arcWeight = distance*self.sndhlp.accessCost*requestDemand
                if origin == arc[0]:
                    arcWeight -= self.model.getDualsolLinear(self.openHubConss[arc[1]][request])
```

```
                requestGraph.add_edge(arc[0], arc[1], res_cost=[1, arcTime], weight=max(arcWeight, 0), distance=distance)

#then we add the 2 extra edges "Source"-origin and destination-"Sink" with no weight, no distance and no resource consumptions (a part from the presence of the arc in the path), because the cspy solving method for the shortest path problem requires the presence of the nodes "Source" and "Sink" to work
        requestGraph.add_edge("Source", origin, res_cost=[1, 0], weight=0, distance=0)
        requestGraph.add_edge(destination, "Sink", res_cost=[1, 0], weight=0, distance=0)

#now we include the branching rules, removing from the request graphs the hubs and the transfer arcs that are forbidden, in order not to generate paths including them
        self.setForbiddenIntraHubsArcs()
        self.setForbiddenHubs()
        for arc in self.forbiddenIntraHubsArcs:
            requestGraph.remove_edge(arc[0], arc[1])
        for hub in self.forbiddenHubs:
            requestGraph.remove_node(hub)
        return requestGraph

#this method updates the dictionaties of feasible paths of the request when a new variable is added to the master problem
    def updateFeasiblePaths(self, path, request):
        self.sndhlp.requestsFeasiblePaths.get(request).append(path)
        for arc in path.intraHubsArcs:
            if self.sndhlp.requestsArcFeasiblePaths.get(request).get(arc) is not None:
                self.sndhlp.requestsArcFeasiblePaths.get(request).get(arc).append(path)
            else:
                self.sndhlp.requestsArcFeasiblePaths.get(request)[arc] = [path]
        for hub in path.hubs:
            if self.sndhlp.requestsHubFeasiblePaths.get(request).get(hub) is not None:
                self.sndhlp.requestsHubFeasiblePaths.get(request).get(hub).append(path)
            else:
                self.sndhlp.requestsHubFeasiblePaths.get(request)[hub] = [path]

#this is the inner methods of the SCIP Pricer to launch the pricing problem
    def pricerredcost(self):
```

```
            return self.performPricing()

    #this is the inner method of the SCIP Pricer to transform the master problem constraints
    def pricerinit(self):
        self.numHubsCons = self.model.getTransformedCons(self.numHubsCons)
        for (req, cons) in self.deliverFullRequestsConss.items():
            self.deliverFullRequestsConss[req] = self.model.getTransformedCons(cons)
        for (arc, cons) in self.numVehiclesConss.items():
            self.numVehiclesConss[arc] = self.model.getTransformedCons(cons)
        for hub in self.openHubConss.keys():
            for (req, cons) in self.openHubConss[hub].items():
                self.openHubConss[hub][req] = self.model.getTransformedCons(cons)

    #this method is necessary to include the branching rules in the column generation algorithm, by forbidding the hubs which upper bound is not 1
    def setForbiddenHubs(self):
        self.forbiddenHubs = []
        for hub in self.sndhlp.hubs:
            if self.model.isLT(self.h[hub].getUbLocal(), 1):
                self.forbiddenHubs.append(hub)

    #this method is necessary to include the branching rules in the column generation algorithm, by forbidding the transfer arcs which upper bound is not 1
    def setForbiddenIntraHubsArcs(self):
        self.forbiddenIntraHubsArcs = []
        for arc in self.sndhlp.intraHubsArcs:
            if self.model.isLT(self.v[arc].getUbLocal(), 1):
                self.forbiddenIntraHubsArcs.append(arc)
```

Appendix D

Matheuristic Approach

In this chapter we present the code of the matheuristic approach. It exploits the B&P algorithm to perturb once or multiple times the solution space.

```
from pyscipopt import SCIP_PARAMSETTING
from SNDHLPgeneration import *
from pricingSNDHLP import *

#this method perturb the previous solution space, applying the matheuristic approach
def newPerturbation(heuristicObjValue, oldObjValue, oldPathBasedModel, oldSNDHLP, numHubs, splittableRequests, timeLimit, gapLimit):

    newPerturbedPathBasedModel = createPathBasedSNDHLPmodel(oldSNDHLP, splittableRequests)
    #new constraint to perturb the current open hubs, imposing that the 75% of them must remain open (rounding down to the previous lower integer)
    perturbedHubsCons = newPerturbedPathBasedModel.master.addCons(
        quicksum(newPerturbedPathBasedModel.h[i] for i in oldSNDHLP.hubs if oldPathBasedModel.master.getVal(oldPathBasedModel.h[i])==1) >= int(0.75*numHubs),
        name="perturbedHubsCons",
        separate=False,
        modifiable=True)

    #new constraints to perturb the current number of vehicles per intra-hubs arcs, imposing a maximum number of vehicles
    maximumNumVehiclesConss = {}
    for a in oldSNDHLP.intraHubsArcs.keys():
```

```
            arcCurrentVehicles = oldPathBasedModel.master.getVal(oldPathBasedModel.v[a])
            if arcCurrentVehicles > 0:
                maximumNumVehiclesConss[a] = newPerturbedPathBasedModel.master.addCons(
                    newPerturbedPathBasedModel.v[a] <= 1.5*arcCurrentVehicles,
                    name=f"maximumVehiclesCons_{a}",
                    separate=False,
                    modifiable=True)

    newPerturbedPathBasedModel.master.setObjlimit(oldObjValue)

#set solver parameters
    newPerturbedPathBasedModel.master.setPresolve(SCIP_PARAMSETTING.OFF)
    newPerturbedPathBasedModel.master.setIntParam("presolving/maxrestarts", 0)
    newPerturbedPathBasedModel.master.setSeparating(SCIP_PARAMSETTING.OFF)
    setTimeAndGapLimits(newPerturbedPathBasedModel.master, timeLimit, gapLimit)

#call the method for the initialization of the pricer variables and constraints, and consequent inclusion in the master problem
    pricerInitialization(newPerturbedPathBasedModel, oldSNDHLP, splittableRequests)
    newPerturbedPathBasedModel.master.optimize()

    newObjValue = round(newPerturbedPathBasedModel.master.getPrimalbound(), 2)
    newSolvingTime = round(newPerturbedPathBasedModel.master.getSolvingTime(), 2)
    newGap = 100*newPerturbedPathBasedModel.master.getGap()
    newHeuristicSolutionImprovement = 100*round(((heuristicObjValue - newObjValue)/heuristicObjValue), 4)
    newPerturbedSolutionImprovement = 100*round(((oldObjValue - newObjValue)/oldObjValue), 4)

    return newObjValue, newSolvingTime, newGap, newHeuristicSolutionImprovement, newPerturbedSolutionImprovement, newPerturbedPathBasedModel, oldSNDHLP

#to try to perturb once or multiple times the optimal solution obtained from an heuristic method by imposing in the path-based model new constraints based on the values in the heuristic solution
```

```
def testMultiplePerturbationsHeuristicSNDHLP(fileName, accessCost, intraHubsCost, vehicleCapacity, numHubs, numPaths, numHops, maxTransportTime, splittableRequests, timeLimit, gapLimit, heuristicApproach, numPerturbations):

    heuristicObjValue, heuristicSolvingTime, heuristicGap, heuristicPathBasedModel, sndhlp = testHeuristicSNDHLP(fileName, accessCost, intraHubsCost, vehicleCapacity, numHubs, numPaths, numHops, maxTransportTime, splittableRequests, timeLimit/50, gapLimit, heuristicApproach)
    totalSolvingTime = heuristicSolvingTime
    perturbations = dict()
    perturbations[0] = [heuristicObjValue, heuristicSolvingTime, 0.0, 0.0, heuristicPathBasedModel, sndhlp]

#launch the perturbations loop for a specific number of times
    for i in range(1,numPerturbations+1):
        j = i-1
#retrieve the old perturbation elements
        oldObjValue, oldPathBasedModel, oldSNDHLP = perturbations[j][0], perturbations[j][4], perturbations[j][5]
#compute the new perturbation elements and save them in the dictionary of perturbations
        newObjValue, newSolvingTime, newGap, newHeuristicSolutionImprovement, newPerturbedSolutionImprovement, newPerturbedPathBasedModel, newSNDHLP = newPerturbation(heuristicObjValue, oldObjValue, oldPathBasedModel, oldSNDHLP, numHubs, splittableRequests, timeLimit, gapLimit)
        perturbations[i] = [newObjValue, newSolvingTime, newGap, newHeuristicSolutionImprovement, newPerturbedSolutionImprovement, newPerturbedPathBasedModel, newSNDHLP]
        totalSolvingTime += newSolvingTime
#exit condition in case of no more improvements possible
        if newObjValue == oldObjValue:
            numPerturbations = i
            break

    firstPerturbedObjValue, firstHeuristicSolutionImprovement, firstPerturbedPathBasedModel = perturbations[1][0], perturbations[1][3], perturbations[1][5]
    finalPerturbedObjValue, finalGap, finalHeuristicSolutionImprovement, finalPerturbedPathBasedModel = perturbations[numPerturbations][0], perturbations[numPerturbations][2], perturbations[numPerturbations][3], perturbations[numPerturbations][5]

```

```
    return heuristicObjValue, firstPerturbedObjValue,
finalPerturbedObjValue, firstHeuristicSolutionImprovement,
finalHeuristicSolutionImprovement, totalSolvingTime, finalGap,
finalPerturbedPathBasedModel, firstPerturbedPathBasedModel,
perturbations
```

Bibliography

AG, Logistische Informations Systeme (2023). *Definition of Intermodal Transport*. URL: https://www.lis.eu/en/lexikon/intermodal-transport/ (visited on 01/31/2023) (cit. on p. 18).

Alumur, Sibel A., Bahar Y. Kara, and Oya E. Karasan (2012). «Multimodal hub location and hub network design». In: *Omega* 40(6), pp. 927–939. ISSN: 0305-0483. DOI: 10.1016/j.omega.2012.02.005 (cit. on pp. 17, 24).

Bang-Jensen, J. and G. Gutin (2001). *Digraphs: Theory, Algorithms, and Applications*. Monographs in Mathematics. Springer. ISBN: 9781852332686 (cit. on p. 27).

Barnhart, Cynthia, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance (1998). «Branch-and-Price: Column Generation for Solving Huge Integer Programs». In: *Operations Research* 46(3), pp. 316–329. ISSN: 0030364X, 15265463 (cit. on p. 13).

Benders, Jacobus F. (1962). «Partitioning procedures for solving mixed-variables programming problems». In: *Numerische Mathematik* 4, pp. 238–252. DOI: 10.1007/BF01386316 (cit. on p. 12).

Boschetti, Marco Antonio and Vittorio Maniezzo (2022). «Matheuristics: using mathematics for heuristic design». In: *4OR* 20(2), pp. 173–208. DOI: 10.1007/s10288-022-00510-8 (cit. on p. 15).

Campbell, James F. (2009). «Hub location for time definite transportation». In: *Computers And Operations Research* 36(12), pp. 3107–3116. ISSN: 0305-0548. DOI: 10.1016/j.cor.2009.01.009 (cit. on p. 24).

Çetiner, Selim, Canan Sepil, and Haldun Süral (Dec. 2010). «Hubbing and routing in postal delivery systems». In: *Annals of Operations Research* 181, pp. 109–124. DOI: 10.1007/s10479-010-0705-2 (cit. on p. 51).

Contreras, Ivan (2021). «Hub Network Design». In: *Network Design with Applications to Transportation and Logistics*. Ed. by Teodor Gabriel Crainic, Michel Gendreau, and Bernard Gendron. Springer Books. Springer. Chap. 18, pp. 567–598. DOI: 10.1007/978-3-030-64018-7 (cit. on p. 17).

Crainic, Teodor Gabriel (2000). «Service network design in freight transportation». In: *European Journal of Operational Research* 122(2), pp. 272–288. ISSN: 0377-2217. DOI: 10.1016/S0377-2217(99)00233-7 (cit. on p. 17).

Dantzig, George B. and Mukund N. Thapa (1997). «Linear Programming 1». In: *The Simplex Method*. Springer Series in Operations Research and Financial Engineering. Springer, pp. 63–111. DOI: 10.1007/b97672 (cit. on p. 5).

de Camargo, Ricardo S., Gilberto de Miranda, Morton E. O'Kelly, and James F. Campbell (2017). «Formulations and decomposition methods for the incomplete hub location network design problem with and without hop-constraints». In: *Applied Mathematical Modelling* 51, pp. 274–301. ISSN: 0307-904X. DOI: 10.1016/j.apm.2017.06.035 (cit. on p. 25).

Desrosiers, Jacques and Marco E. Lübbecke (2005). «A Primer in Column Generation». In: *Column Generation*. Ed. by Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon. Springer US, pp. 1–32. ISBN: 978-0-387-25486-9. DOI: 10.1007/0-387-25486-2_1 (cit. on pp. 8, 9).

Dictionary, Cambridge English (2023). *Definition of Service*. URL: https://dictionary.cambridge.org/dictionary/english/service (visited on 01/28/2023) (cit. on p. 16).

Eiselt, Horst A. and Carl-Louis Sandblom (2000). «Integer Programming and Network Models». In: Springer Berlin, Heidelberg. DOI: 10.1007/978-3-662-04197-0 (cit. on p. 14).

Fischetti, Martina and Matteo Fischetti (2016). «Matheuristics». In: *Handbook of Heuristics*. Ed. by Rafael Martí, Pardalos Panos, and Mauricio G.C. Resende. Springer International Publishing, pp. 1–33. DOI: 10.1007/978-3-319-07153-4_14-1 (cit. on p. 15).

Fischetti, Matteo and Andrea Lodi (2011). «Heuristics in Mixed Integer Programming». In: *Wiley Encyclopedia of Operations Research and Management Science*. Ed. by J.J. Cochran. John Wiley & Sons, Ltd, pp. 738–747. DOI: 10.1002/9780470400531.eorms0376 (cit. on p. 15).

Gilmore, Paul C. and Ralph E. Gomory (1961). «A Linear Programming Approach to the Cutting Stock Problem». In: *Operations Research* 9, pp. 849–859. DOI: 10.1287/opre.9.6.849 (cit. on p. 13).

Irnich, Stefan, Ann-Kathrin Rothenbächer, and Michael Drexl (2016). «Branch-and-price-and-cut for a service network design and hub location problem». In: *European Journal of Operational Research* 255(3), pp. 935–947. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2016.05.058 (cit. on pp. 24, 25).

Krishnamoorthy, Mohan, Jamie Ebery, Andreas Ernst, and Natashia Boland (2000). «The capacitated multiple allocation hub location problem: Formulations and algorithms». In: *European Journal of Operational Research* 120(3), pp. 614–631. ISSN: 0377-2217. DOI: 10.1016/S0377-2217(98)00395-6 (cit. on p. 51).

Lodi, Andrea and Matteo Fischetti (2003). «Local Branching». In: *Mathematical Programming* 98, pp. 23–47. DOI: 10.1007/s10107-003-0395-5 (cit. on p. 15).

McLeod, Sam and Carey Curtis (2020). «Understanding and Planning for Freight Movement in Cities: Practices and Challenges». In: *Planning Practice & Research* 35(2), pp. 201–219. DOI: 10.1080/02697459.2020.1732660 (cit. on p. 16).

Nemhauser, George (2012). «Column generation for linear and integer programming». In: *Optimization Stories*. Martin Grotschel, pp. 65–73 (cit. on p. 8).

O'Kelly, M.E., J.F. Campbell, G. Miranda, and Y. Park (2023). *Studies in Hub Location and Network Design*. URL: https://www.researchgate.net/project/Studies-in-Hub-Location-and-Network-Design (visited on 02/03/2023) (cit. on p. 51).

O'Kelly, Morton (1986). «The Location of Interacting Hub Facilities». In: *Transportation Science* 20, pp. 92–106. DOI: 10.1287/trsc.20.2.92 (cit. on pp. 17, 24, 25).

Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-05594-8 (cit. on p. 14).

Pióro, Michał and Deepankar Medhi (2004). «Application of Optimization Techniques for Protection and Restoration Design». In: *Routing, Flow, and Capacity Design in Communication and Computer Networks*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann: San Francisco, pp. 403–454. DOI: 10.1016/B978-0-12-557189-0.70006-1 (cit. on p. 2).

Savelsbergh, Martin W. P. (2001). «Branch and Price: Integer Programming with Column Generation». In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M. Pardalos. Springer US, pp. 218–221. ISBN: 978-0-306-48332-5. DOI: 10.1007/0-306-48332-7_47 (cit. on p. 12).

Schrijver, Alexander (1986). «Fundamental concepts and results on polyhedra, linear inequalities, and linear programming». In: *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, pp. 85–98 (cit. on p. 11).

Serper, Elif Zeynep and Sibel A. Alumur (2016). «The design of capacitated intermodal hub networks with different vehicle types». In: *Transportation Research Part B: Methodological* 86, pp. 51–65. ISSN: 0191-2615. DOI: 10.1016/j.trb.2016.01.011 (cit. on p. 25).

Yoon, M-G and John Current (2008). «The Hub Location and Network Design Problem with Fixed and Variable Arc Costs: Formulation and Dual-Based Solution Heuristic». In: *Journal of the Operational Research Society* 59, pp. 80–89. DOI: 10.1057/palgrave.jors.2602307 (cit. on p. 24).

Zhang, Mo, Bart Wiegmans, and Lori Tavasszy (2013). «Optimization of multimodal networks including environmental costs: A model and findings for transport policy». In: *Computers in Industry* 64(2), pp. 136–145. ISSN: 0166-3615. DOI: 10.1016/j.compind.2012.11.008 (cit. on p. 24).