Models and Algorithms for Problems of Scheduling/Routing with Synchronizations

José Luis Figueroa Gonzalez

THÈSE

Présentée par

José Luis FIGUEROA GONZÁLEZ

pour obtenir le diplôme de

Docteur d'université

Spécialité : Informatique

Models and Algorithms for Problems of Scheduling/Routing with Synchronizations

---

Modèles et Algorithmes pour Problèmes de Scheduling/Routing avec Synchronizations

---

Soutenue publiquement le 4 décembre 2023 devant le jury

M.	Maurício	CARDOSO DE SOUZA	Rapporteur et examinateur
M.	Pierre	FOUILHOUX	Rapporteur et examinateur
M.	Alain	QUILLIOT	Directeur de Thèse
M.	Eric	SANLAVILLE	Président du jury, examinateur
Mme.	Hélène	TOUSSAINT	Codirectrice de Thèse
Mme.	Annegret	WAGLER	Codirectrice de Thèse

École Doctorale Sciences Pour l'Ingénieur
Université Clermont Auvergne

Acknowledgements

Doing a PhD is a task that constantly demands intense work rhythms and there is always the uncertainty of not being able to finish it or not being able to do it within the established deadlines. It is common to find PhD students who complain about the impact that their studies have on their social relationships and on their health (physical and mental). For all these reasons, I must first thank my wife, Dulce Rocío López Acosta, for having supported me unconditionally when I decided to start a second PhD.

I want to thank Dr. Alain Quilliot for his kind reception and his continuous guidance and expertise throughout all the research work. I thank Dr. Hélène Toussaint for all her technical support (especially during the implementation of the Branch-and-Cut callbacks in Cplex) and for her suggestions during the preparation of the "comités de thèse". I also want to thank Dr. Annegret Wagler for her comments and suggestions to improve my work (especially for the suggestion to consider the distance from the depot to define the extended subtour constraints). In general, I want to thank my PhD advisors for all the freedom and confidence they gave me to carry out the research work.

Moving more than 10,000 km from the country of origin entails many sacrifices as one has to distance oneself from family and friends. In many aspects one has to start from scratch and all the help one receives is particularly valuable. So I have to thank Matthieu Gondran and Rafael Colares, the first friends I found at the LIMOS. I would also like to thank all the colleagues with whom I had the opportunity to interact: Axel Delsol, Iván Peña Arenas, Mateus Vilela Souza, Eloïse Yollande Molé Kamga, Saeedeh Dehghani Firouzabadi, Thi Thu Trang NGO, Chijia Liu, Benoit Albert, Caroline Brosse, Amal El Kaid, Thị Quỳnh Trang Võ, Aurélien Mombelli, Mari Chaivkovskaia, and Tam Tien Tran.

I also would like to thank the LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes) for hosting me and providing me with most of the physical conditions for the research development.

I thank the staff of the Université Clermont-Auvergne, LIMOS, and ISIMA, particularly Bastien Doreau (for all the times he helped me solve problems with my computer and e-mail accounts) and Raquel Melara (for all her help during the procedures related to the renewal of my "titre de séjour"). I also thank "Pôle emploi" for the financial support throughout all the many months during which I was waiting for the date of my thesis defense.

I also thank my family and all the people who have supported me from Mexico, particularly Dr. Gilberto Calvillo Vives for his constant encouragement.

I dedicate this work to the memory of Dr. David Romero Vargas, who introduced me to metaheuristics and first told me about the LIMOS. I would also like to dedicate this work to the memory of my grandmother María Gaytán Ocampo.

Abstract

This work studies two Pickup-and-Delivery Problems with transfers and time horizon, and provides models, theoretical results, and methods for handling them.

The first problem is a Relocation Problem which can be seen as a one-commodity many-to-many Capacitated Pickup-and-Delivery Problem with unpaired pickups and deliveries, transfers, and time horizon. We introduce a 2-commodity flow model on a Time-Expanded network and we propose a "Project-and-Lift" approach for handling that model. First, we derive a "Projected Model" that manages the time constraints in an implicit way and that provide us with a lower bound for the optimal solution cost of a Relocation Problem instance. We strengthen the Projected Model by adding specific "Extended-Subtour" constraints (related to the time horizon) and "Feasible-Path" constraints (related to a path decomposition property that must be verified by the item flow), and we solve it efficiently by branch-and-cut. Finally we deal with the "Lift" problem, which consists in computing good quality solutions starting from solutions of the Projected Model. We manage several levels of flexibility for constructing those solutions, and propose mixed integer linear programming models and a decomposition approach for handling the problem in a flexible way.

The second problem is a one-to-one Capacitated Pickup-and-Delivery Problem with paired demands, transfers, and time horizon. We start by introducing the Virtual Path Problem which consists in the modification of an acyclic digraph which involves an underlying constraint system. The aim is to construct a directed path connecting two given vertices, while minimizing a cost function and maintaining a feasible constraint system. We propose an A*-like algorithm (called virtual A*) for solving this problem in an exact way. Next we study the problem of the exact insertion of a single request into a Pickup-and-Delivery Problem with Transfers schedule. We show this problem can be seen as a particular case of the Virtual Path Problem and so we can use the virtual A* algorithm to solve this problem in an exact way. We also propose a fast heuristic based on Dijkstra's algorithm. Finally we combine those single request algorithms with some classical metaheuristics for handling the insertion of multiple requests into a Pickup-and-Delivery Problem with Transfers schedule.

Key words: Combinatorial Optimization, Integer Programming, Vehicle Routing Problem, Pickup-and-Delivery Problem with transfers, Constrained Shortest Path Problem, A* Algorithm, Relocation Problem, Multicommodity Flows, Branch-and-Cut Algorithm, Column Generation Algorithm, Time-Expanded Networks.

Contents

Part I

Introduction

Context

According to a recent report [210] of the World Meteorological Organization, a specialized agency of United Nations (UN), the years from 2015 to 2022 are likely to be the hottest ever reported, and despite La Niña conditions, which have contributed to keep global temperature low during the last two years, the global mean temperature in 2022 is expected to be the fifth or the sixth warmest ever recorded.

Independent global surface temperature datasets (see [121, 141, 147, 162, 185, 223] and Figure 1) agree that the average global temperature on Earth has increased by at least 1.1° Celsius since 1880, and that the majority of the warming has occurred since 1975, at a rate of roughly 0.15 to 0.20°C per decade [165].

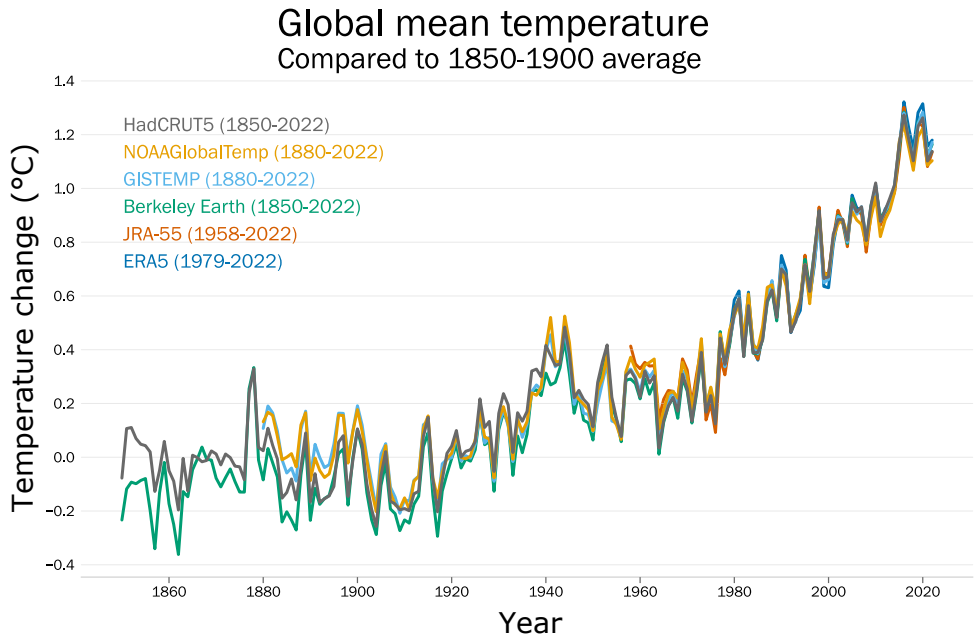


Figure 1: Global annual mean temperature difference from the pre-industrial reference period (1850–1900) for six global temperature datasets (1850–2022, 2022 based on an average to September)[210].

It has been estimated that more than 50% of the observed increase in global mean surface temperatures from 1950 to 2010 is very likely due to anthropogenic increase in greenhouse gases (GHG) concentrations in the atmosphere [9]. GHGs cause a positive radiative imbalance at the top of the atmosphere, and lead to an accumulation of heat in the oceans. In 2021, the atmosphere concentrations of carbon dioxide, methane, and nitrous oxide reached record highs [210].

The main source of those anthropogenic GHG emissions is the combustion of fossil fuels (like coal, petroleum, and natural gas), and is highly influenced by some demographic phenomena like the accelerated rates of population growth and urbanization.

The most ancient fossils of Homo sapiens allow to estimate that modern humans appeared on earth around 300,000-200,000 years ago [120, 125] and since then, their population have tended to grow in the long term [76]. Recent demographic estimations show that world population has been growing year by year in an uninterrupted way since the end of the Black Death around 1350 [126]. The world population began growing more rapidly since the Industrial Revolution due to the reduction in mortality achieved with sanitation and technological advances that improved medical treatments and agricultural productivity.

The growth rate of the world's population peaked in 1965-1970 [207] but the population is still growing. For the year 2019, the United Nations estimated the global population at 7.7 billion people and this number is expected to reach the 8.0 billion mark by the end of 2022 or within 2023. Figure 2 shows estimates of the total population from 1950 to 2019. It also shows the probabilistic median, the 95 per cent prediction interval of the probabilistic population projections, and the high and low variants (+/- 0.5 child) for the period from 2020 to 2100 (source data were retrieved from [209] and plotted with the R programming language [202]).

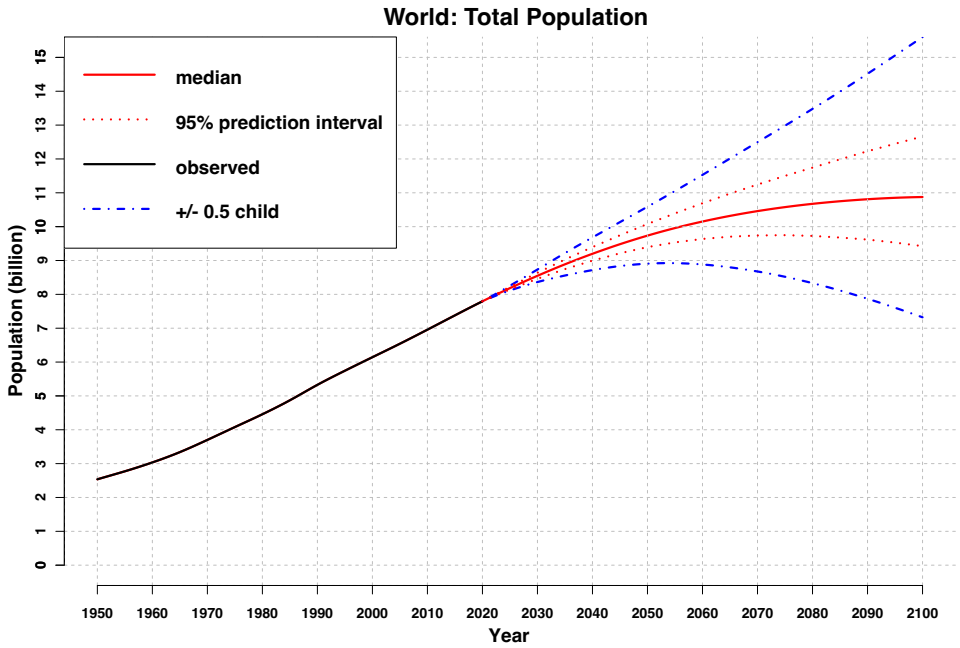


Figure 2: Estimates and probabilistic projections of the total population.

In the last decades, there has been a migratory tendency from rural to urban areas. According to the UN reports [206] and [208], in 1950, about 30% of world's population was living in urban areas, whilst in 2018, this percentage was increased until 55%. The urbanization process still continues and by 2030, urban areas are projected to house 60% of people globally and one third of humans will live in cities with at least half a million inhabitants. By 2050, 68% of the world's population is projected to be urban.

Here, it is worth to note that there is no global homogeneity in population distribution or urbanization rates. Figure 3 was extracted from [206] and shows estimations for the percentages of population living in rural and urban areas in 2018 and the corresponding projections for 2030, classified by population size of human settlements and geographical region.

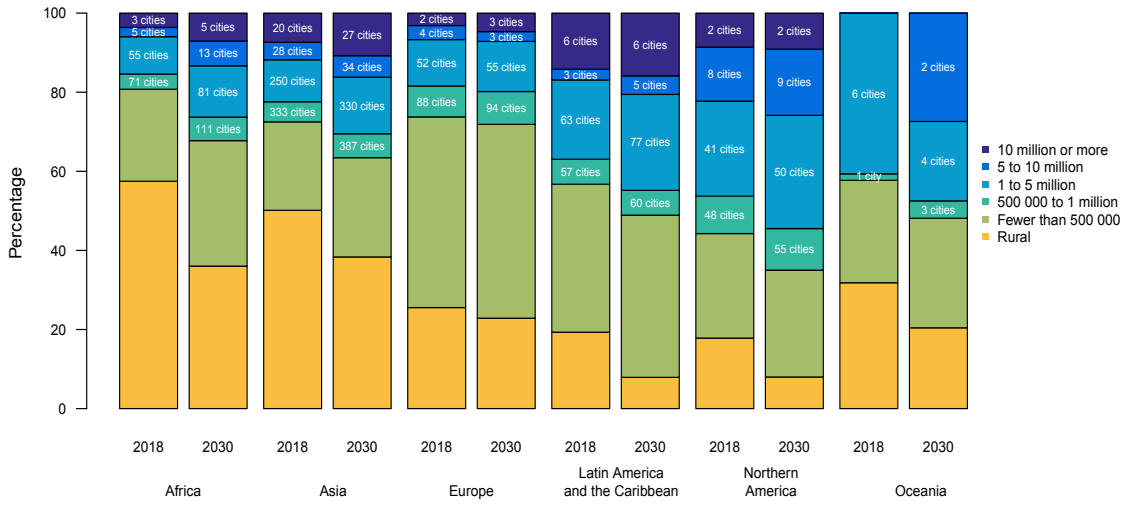


Figure 3: Percentages of population distribution estimated in 2018 and projections for 2030, classified by population size of human settlements and geographical region. The number of cities with 500,000 inhabitants or more are also displayed.

The phenomenon of urbanization has also led to a gradual emergence of megacities. A megacity is an urban agglomeration with more than ten million inhabitants. Globally, in 2021, there were 36 megacities in the world and a total of 90 urban areas with five million or more population [63], the number of megacities is projected to rise to 43 in 2030. Figure 4 was taken from [206] and shows the geographical distribution of cities projected to have one million inhabitants or more in 2030.

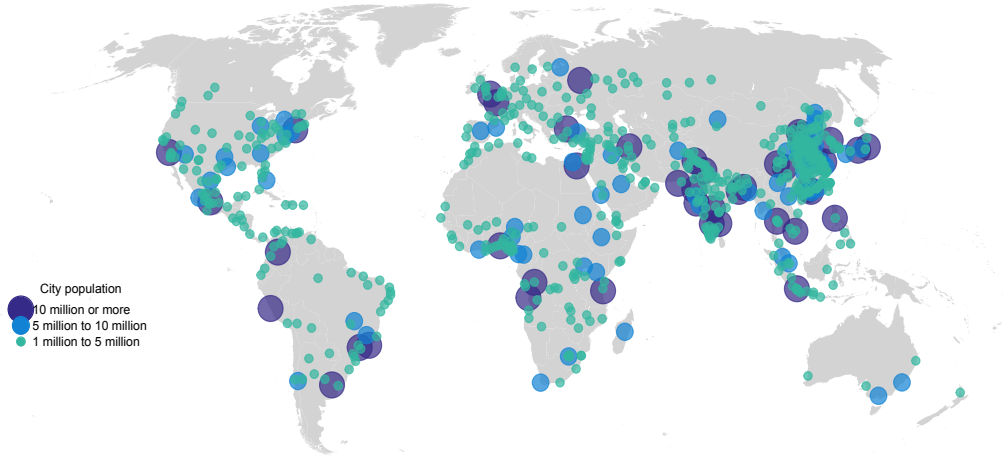


Figure 4: Map of cities with one million inhabitants or more projected for 2030.

The excessive growth of urban agglomerations may potentiate some environmental, economical, and social issues. These problematics are usually very complex and keep some relation with other human activities like, for example, the transport.

Urban transport problems are not particularly recent, in 1977 Michael Thomson [203] published a study about traffic in big cities and coined the phrase "the larger the city, the greater the problems and the higher the costs of providing transport". In that pioneer work, Thomson also identified the seven subproblems of the urban transport problem that are shown in Figure 5. One year later, Banister [20] published another study about transport problems in global cities, and revisited the work of Thomson.

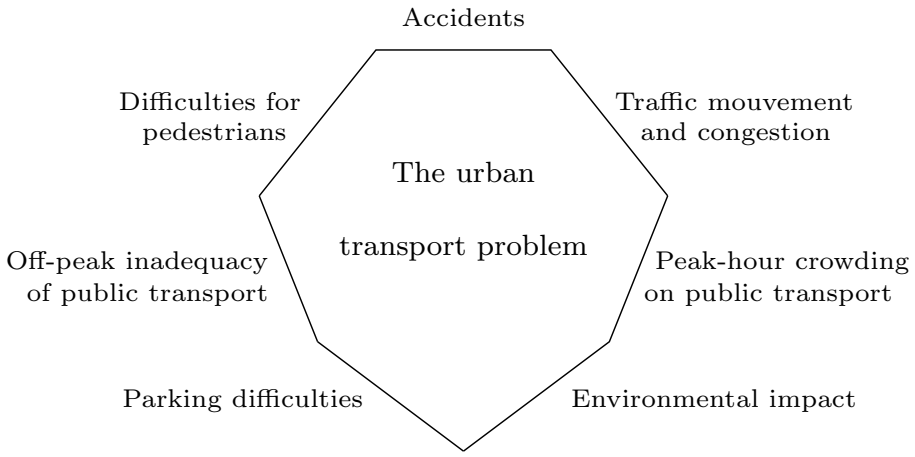


Figure 5: The seven subproblems of the urban transport problem according to Thomson [203].

Nowadays, those subproblems still persists and some of them have become of greater concern. For example, the "2019 Traffic Scorecard" published by INRIX, Inc. [127] points out that traffic congestion continues to grow across the world. It also contains estimations of the average number of hours lost by each driver during the year, and one ranking of the most congested cities in the World. The ten first cities of this ranking are presented in Table 1.

Traffic congestion has also a negative impact on the economy due to fuel consumption and delays. The American Transportation Research Institute estimates that between 2015 and 2016, the additional operational costs incurred by the trucking industry due to traffic congestion were $74.5 billion, with $66.1 billion of it occurring in urban areas [123].

Countermeasures to diminish congestion include improving road infrastructure, urban planning and design, increasing road capacity, limiting the number of vehicles in circulation (within a particular area or during a period of time), use of intelligent transportation systems, etc.

Urban Area	Impact Rank	Hours Lost in Congestion
Bogota	1	191
Rio de Janeiro	2	190
Mexico City	3	158
Istanbul	4	153
São Paulo	5	152
Rome	6	166
Paris	7	165
London	8	149
Boston	9	149
Chicago	10	145

Table 1: The ten most congested cities in the world in 2019 according to [127]. Impact rank captures the aggregate impact of congestion's impact relative to population, whereas hours lost in congestion captures exclusively the intensity of traffic in a given city.

Initiatives that involve changes in road infrastructure are usually expensive, and its viability depends mainly on the particular characteristics of a city. Also, it has been observed that adding road capacity sometimes results in attracting more traffic, and making congestion worse [72]. For these reasons, several large cities in the world prefer to limit the number of vehicles in circulation [213], and encourage the use of alternative ways of transport (like bike-sharing or carpooling).

In a bike-sharing system, users are able to access bicycles for use when required. Those bicycles can be retrieved from a set of bike-sharing stations that are strategically scattered over an urban area. Bike-sharing stations are typically unattended, and the majority of bike-sharing operators cover the costs of maintenance, storage, and parking [152]. Users generally join bike-sharing organizations on an annual, monthly, daily, or per-trip basis, and trips of less than 30 minutes are included within the membership fees [196]. According to [221], in 2021 there were approximatively 2,000 active bike-sharing systems distributed in 85 countries.

Carpooling is the sharing of car journeys for allowing that more than one person travels in a car, and prevents the need for others to have to drive to a location themselves. Carpooling has some economic benefits because by having more people using one vehicle, the average travel cost per capita is reduced. Also, carpooling is environmentally more beneficial than driving alone, because it diminishes the number of vehicles in circulation, and this in turn reduces air pollution, and carbon emissions.

Despite the above-mentioned benefits, carpooling in the US reduced from 19.7% in 1980 to 7.8% in 2021, and the percentage of workers in the US who drove alone during their commute increased from 64.4% in 1980 to 67.8% in 2021 [84, 211].

In the last decades, the maintained tendency of people to continue driving alone, the aggravation of environmental issues, and the increase of oil prices have turned the development of cleaner alternative fuels into a high priority for many governments and vehicle manufacturers around the world.

In France, the "Contrats de plan État-Région" (CPER) are regional plans for programming and financing the major development projects, such as the creation of infrastructure and the support of key sectors for the future. The CPERs of the Auvergne-Rhône-Alpes region [175, 176, 177] including the finance of operations related to ecology and energy transition.

With regard to renewable energies, one of the objectives of the CPER-Auvergne-Rhône-Alpes 2021-2027 is to increase the use of alternatives energies from 19% to 36% by 2030. Also, this CPER considers that hydrogen is a key technology for environmental transition and industrial growth prospects because meet three major challenges: energy and environmental transition, air quality (thanks to zero-emission mobility solutions), and job creation (the Region aims to deploy an "Auvergne-Rhône-Alpes Hydrogen HUB" for hydrogen production). To complete successfully the development of hydrogen technologies, the CPER sets the following challenges.

1. The industry's transition to clean hydrogen.
2. Mobility, distribution infrastructure, and vehicles (specially for heavy vehicles).
3. Hydrogen energy, production from electrolysis at a competitive price, storage and transport infrastructure, stationary applications, network and market services.

The use of clean hydrogen (and other alternative fuels or energies) brings some new difficulties related to transport because the infrastructure for recharging or refueling vehicles is still scarce, the availability of those alternative fuels or energies may fluctuate, and the vehicles' efficiency vary a lot from one technology to another.

This thesis contributes to face the third above challenge by providing models to solve vehicle routing problems with transfers and time horizon, and strengthening informed decision-making on transport network management.

Transfers may contribute to diminish the number of vehicles in circulation because sometimes it is possible to satisfy more transportation requests with the same number vehicles; they also may contribute to save fuel by reducing the total travel distance. However, a careful handling of time constraints is required to guarantee that a transfer will take place.

In contrast, by imposing a time horizon we can conceive transportation schedules to be executed within a particular period of time where resources may be available (e.g., a sunny or windy day, surplus of hydrogen or other alternative fuels).

Main Contributions

The main topic of this work are the Vehicle Routing Problems (VRP), which are mathematical abstractions of real-world transportation problems. VRPs began to be studied formally around 1950 for optimizing truck routes to deliver gasoline to service stations, and since then the number of applications and mathematical models has been increasing gradually. Nowadays, many variants of vehicle routing problems can be distinguished according to the type of constraints involved (e.g., capacity, time windows, transfers of loads, synchronization of visits), their objective functions (e.g., minimize riding costs of vehicles or loads, waiting time of passengers, number of vehicles used, maximize total profit or number of requests satisfied), or even their applications (e.g., transportation of goods or people, pickup and delivery, relocation of objects).

Given the computational complexity of many of those problems, the use of computers has been necessary since the early studies and this situation has determined indirectly the type of addressed problems. For example, due to the existing limitations of computing power, most of the early papers were focused on single vehicle problems and small instances. Then, the development of the transportation industry brought soon more challenging problems and the use of increasingly sophisticated electronic computers, mathematical models, and algorithms became necessary.

Some real-world transportation problems such as congestion traffic and air pollution have made it desirable to reduce both the number of vehicles in circulation and the total traveled distance. For some transportation problems, these factors can be meet by allowing transfers of loads between vehicles or by limiting the circulation of vehicles to occur within a given time horizon that avoids rush hours traffic.

The content of this thesis may contribute to optimize transportation systems that allow transfers between vehicles and that are constrained by a time horizon. In particular, two Pickup-and-Delivery Problems (PDP) are examined: a Relocation Problem which can be seen as a one-commodity many-to-many Capacitated PDP with unpaired pickups and deliveries, transfers, and time horizon; and a one-to-one Capacitated PDP with paired demands, transfers, and time horizon.

In Part I (Chapter 1) we survey the related literature and we briefly describe some of the methods that will be used later.

Part II (Chapters 2 and 3) describes a Relocation Problem with transfers and time horizon and proposes a "Project-and-Lift" approach for handling it.

In Chapter 2 we propose a 2-commodity flow model on a Time-Expanded network, and we derive a "Projected Model" that manages the time constraints in an implicit way. We strengthen that Projected Model by adding specific "Extended-Subtour" constraints related to the time horizon and "Feasible-Path" constraints related to a path decomposition property that must be verified by the item flow. The main result of this chapter is an efficient branch-and-cut algorithm for computing a lower bound for the optimal cost of a Relocation Problem instance. Such a bound can be used as a reference point for evaluating the quality of solutions obtained with heuristic methods.

Chapter 3 deals with the problem of computing "good" quality solutions starting from a "Projected Model" solution. We propose several levels of flexibility for constructing those solutions and propose several mixed integer linear programming (MILP) models. We also propose a novel decomposition approach for handling the problem in a more flexible way.

The results of this part were presented at the *7th International Symposium on Combinatorial Optimization* (ISCO 2022), the *2023 congress of the French Operations Research and Decision Support Society* (ROADEF 2023), the *19th Cologne-Twente Workshop on Graphs and Combinatorial Optimization* (CTW 2023), and the *XII Latin-American Algorithms, Graphs and Optimization Symposium* (LAGOS 2023). Parts of this work were published in *Lecture Notes in Computer Science* [87] (Springer), *Graphs and Combinatorial Optimization: from Theory to Applications* [89] (AIRO Springer Series 13), and *Procedia Computer Science* [88] (Elsevier).

Part III (Chapter 4) introduces the Virtual Path Problem and provides an application to the one.to-one Capacitated Pickup-and-Delivery Problem with paired demands, transfers, and time horizon.

We start the by describing the "Virtual Path Problem" and we present the "Virtual A*" algorithm for solving this problem in an exact way. Next, we show that the problem of inserting a single request into a Pickup-and-Delivery Problem with Transfers schedule is a particular case of the Virtual Path Problem. So, we propose the Virtual A* algorithm to solve it in an exact way, and a Dijkstra-like algorithm for handling the problem in a heuristic way. Then we combine those single request algorithms with some metaheuristics, for handling the insertion of multiple requests into a Pickup-and-Delivery Problem with Transfers schedule.

The results of this part were presented for the first time in the *2021 congress of the French Operations Research and Decision Support Society* (ROADEF 2021), and in the *11th International Conference on Operations Research and Enterprise Systems* (ICORES 2022). Two works were published: one in the *ICORES 2022 proceedings* [85] and the other in *Operations Research and Enterprise Systems* (CCIS Springer Series 1985)[86].

Part IV is the last part of this work. It contains the conclusions and provides future lines of research.

A summary of the basic theory and notation used in this work can be found in the Appendix A.

A remark about writing style. This document follows some of the mathematical writing conventions suggested by Knuth et al. (1989) [140]. So, the word "we" is used instead of "I" and should be understood as a kind of "joint reading" between author and reader.

Chapter 1

State of the Art

In this chapter we survey the scientific literature about Vehicle Routing Problems and some of the methods for handling those problems. In Section 1.1 we discuss two seminal works about Vehicle Routing Problems and then we examine in more detail the Pickup-and-Delivery variants. We synthesize the main ideas of the most influential papers (according to reference surveys) and provide a classification of the published works related to Pickup-and-Delivery Problems involving transfers. In Section 1.2 we discuss briefly some of the methods related to the topics that will be used in subsequent chapters.

1.1 Vehicle Routing Problems

We start this section with the discussion of two pioneer works about Vehicle Routing Problems and then we provide a classification perspective of these problems.

The Vehicle Routing Problem (VRP) has roots in the work of Dantzig and Ramser (1959) [59] for solving a problem concerned with the optimum routing of a fleet of gasoline delivery trucks to serve a set of stations. In this paper, the authors introduced the VRP as a generalization of the Traveling Salesperson Problem (TSP) obtained by adding new types of constraints. Some of these constraints are related to the capacity of the trucks, which is supposed to be small in comparison with the total amount of demands, and as a result, every truck is constrained to make a limited number of deliveries on each trip and then return to the depot to replenish; furthermore, each demand must be satisfied in a single trip.

Mathematically, this situation can be posed as a graph problem. The graph has one vertex for each station to be served and one distinguished vertex, the depot, which is linked to every station by two edges; also, every pair of stations is linked by

a single edge. The graph has vertices costs corresponding to the station demands, and edge costs corresponding to the distance between the stations. The aim is to find a vertex cover of cycles, they all sharing only the depot vertex, and such that, the sum of vertices costs in every cycle does not exceed the truck's capacity, and the sum of the costs in the edges of the cycles is minimum. Due to the shape of solutions, the authors suggested initially the name "Clover Leaf Problem", but this name did not catch on.

For solving the above mathematical problem, Dantzig and Ramser gave a linear programming formulation; note that, in those years there was no general method for solving discrete variable linear programming problems, and so the authors allowed variables to take fractional values. Then, the solutions with fractional values were converted into feasible integer solutions by using a set of ad hoc heuristics. The general process presented by Dantzig and Ramser is designed in two stages. In the first stage, the authors solve a linear programming problem whose solution is heuristically converted into an integer solution. From this solution, we can obtain a partition of the non-depot vertices which is composed of singletons and pairs of vertices that can be served in a single trip. Then, these sets are successively combined and merged in pairs in a heuristic way to build a maximal-coarse partition of the non-depot vertices. Each member of the partition (together with the depot) is solved as an instance of the Traveling Salesperson Problem to obtain a final solution which is a cover of cycles. The authors remark that this solution process does not guarantee to find the true optimal value.

Some years later Clarke and Wright (1964) [98] proposed a greedy heuristic for improving the work of Dantzig and Ramser on more realistic contexts. In their work, the authors suppose that there is a fleet of vehicles with distinct capacities $C_1, C_2, \ldots, C_N$, with $C_1 < C_2 < \ldots < C_N$, and to avoid infeasible instances, they consider that there is available an unlimited number of vehicles with capacity C_1. Furthermore, the load required by each customer is considered to be less than C_1. In the first step of the heuristic, Clarke and Wright construct an initial solution by assigning one vehicle to each single customer. Then, they compute the "savings" that can be obtained by assigning some customer to the route of another customer. These computed savings are used to exchange some customers from one route to another while using a greedy strategy for maximizing the savings. By following this strategy, the number of customers on a single route tends to increase, and on the opposite, the mileage covered by the fleet and the number of assigned vehicles tends to decrease. In the subsequent steps, the heuristic continues computing savings and performing exchanges of customers until arriving to a situation where is not possible to obtain more savings. During the algorithm, the sum of the loads required by the

customers on a same route does not have to exceed C_N, and also, if there are only k vehicles of capacity C_i, $i > 1$, it must avoided to assign more than k vehicles of capacity C_i to satisfy the resulting routes. The authors conclude that the solutions obtained with this heuristic can have good quality but they are not guaranteed to be optimal.

After these two seminal papers, the VRP and its variants were gradually attracting more interest, and due to their difficulty and economic importance, hundreds of algorithms (exact and heuristics) have been developed. Nowadays, the VRP constitute still an active field of research. The book by Toth and Vigo (2014) [205] contains an overview of the most important problems, techniques, and algorithms related to these optimization problems.

It is difficult to give an exhaustive classification of all the VRP variants because the terminology is not completely standardized and some problems can be classified in more than one variant. Also, from time to time, the use of new technologies give rise to applications of VRP with new characteristics that make necessary the introduction of new variants (e.g., the Vehicle Routing Problem with Drones [215]).

Hence, following Irnich et al. (2014) [130], we give a classification perspective and we only mention that the VRP can be classified according to:

- the network characteristics (e.g., *dynamic VRP*: some data become available during operation, *stochastic VRP*: unknown data can be modeled by random variables, *deterministic static VRP*: all data is completely known in advance),
- the type of transportation requests (e.g., a pickup request, a delivery request, or a simple visit to a customer),
- the intra-route constraints (e.g., capacity constraints, time windows, route duration, use of multiple vehicles),
- the fleet composition and location (e.g., homogeneous fleet, heterogeneous fleet, multiple depots, vehicle depending routing costs),
- the inter-route constraints (e.g., synchronizations, difference between maximum and minimum route duration, transshipments, transfers), and
- the optimization objectives (e.g., single objective, hierarchical objective, multi-objective).

In this work we study a variant of VRP which is known by the name of Pickup-and-Delivery Problem (PDP).

1.1.1 Pickup-and-Delivery Problem

The Pickup-and-Delivery Problem (PDP) is a variant of VRP that was introduced around 1980; two early references are Psaraftis (1980) [178] and Kalantari et al. (1985) [15]. In the PDP, a transportation request usually consists in picking up people or objects from their origin locations and deliver them in their corresponding destination locations. Battarra et al. (2014) [23] classify the PDP into the following three categories (see Figure 1.1).

- Many-to-many (M-M): each commodity may have multiple origins and destinations, and any location may be the origin or destination of multiple commodities. (e.g., repositioning of inventory, bicycle or car sharing systems),
- One-to-many-to-one (1-M-1): some commodities have to be delivered from a depot to many customers, and other commodities have to be collected at the customers and transported back to the depot (e.g., distribution of beverages and collection of empty cans and bottles).
- One-to-one (1-1): each commodity has a single origin and a single destination between which it must be transported (e.g., urban courier operations)

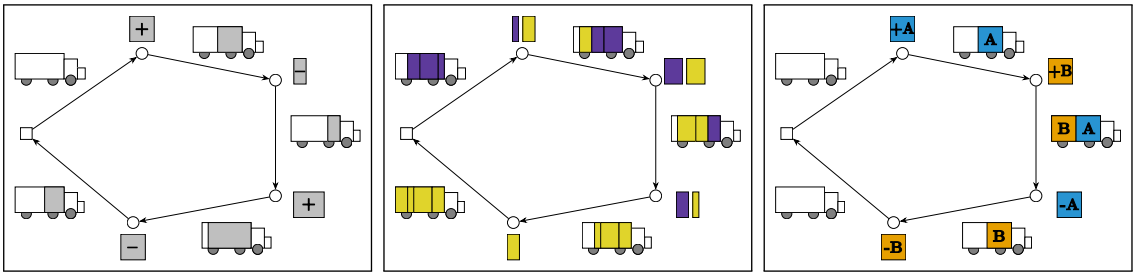


Figure 1.1: Three types of PDP according to Battarra et al. (2014) [23].

In the literature, the PDP for people transportation may include some constraints to ensure a good quality of service to users, and is also known as Dial-A-Ride Problem (DARP) (c.f. Section 1.1.2). Surveys for general Pickup-and-Delivery Problems can be found, for example, in [27, 28, 170, 205]; a more recent survey about DARP is Ho et al. (2018) [122].

Depending on context, two main types of PDP are distinguished. A PDP is *static* (or *offline*) if all the requests are completely known before the beginning of the transportation process. On the other hand, a PDP is *dynamic* (or *online*) if the set of transportation requests is gradually revealed during the transportation process.

We say that a PDP is *vehicle preemptive* if requests can change of vehicle on their way to their destination (e.g., passengers taking flights with connections). Similarly, a PDP is *load preemptive* if vehicles can serve other requests before the current one is delivered (e.g., shared taxis).

The Pickup-and-Delivery Problem with Time Windows (PDPTW) is a PDP variant where each request has to be picked up and delivered within a given time interval specified by a release time and a deadline. In Dumas et al. (1991) [75] is presented an exact model, based on integer programming, for solving some Pickup-and-Delivery problems with time windows. The model contains three types of variables: continuous time variables, continuous load variables, and triple-indexed binary decision variables $x^v_{i,j}$, indicating whether or not the vehicle v goes from vertex i to vertex j in its route. The model also contains 17 types of constraints for modeling the flows of vehicles and requests, while respecting the time and capacity requirements. The authors solve some instances of this problem ranged from 19 to 55 requests, by using a column generation scheme that contains a Constrained Shortest Path Problem as a subproblem, and in turn, the subproblem is handled with a dynamic programming algorithm.

A more recent work that surveys the PDPTW putting the focus on metaheuristics is, for example, Buriol and Sartori (2020) [42].

PDP Involving Transfers

Most of the PDP models consider that the pickup and delivery of an object or person must be performed by the same vehicle, if it is not the case, we talk about problems with transfers. In this section we provide a review of some works related to the PDP with transfers. Then in Table 1.1 we classify those works into exact, heuristic, and metaheuristic approaches.

A *transfer* represents the change of a load from one vehicle to another at some specific vertex called *transfer point*. Depending on the nature of the transfer points, we may distinguish the following three types of transfers.

- *Transshipment*: vehicles can drop loads for other vehicles only at a specific set of vertices called *transshipment points*.
- *Transfer without detours*: vehicles can drop loads for other vehicles at any vertex of the network.

- *Transfer with detours*: it is possible to modify the transportation network by introducing new vertices and arcs. Vehicles can drop loads for other vehicles at those new vertices. In this context, some authors speak about *dynamic transfer points* when the new transfer points are generated as they are needed (see Deleplanque and Quilliot (2013) [61]).

Also, we may distinguish the following two types of vehicle synchronizations to perform a transfer.

- *Weak synchronization*: Vehicles are not required to be at the same time in a transfer point for performing a transfer, but if the receiving vehicle arrives first to the transfer point, it has to wait for the arriving of the emitting vehicle with the load.
- *Strong synchronization*: Vehicles have to meet at the same place and time to perform a transfer.

The notion of transfer in the PDP was introduced by Mitrović-Minić and Laporte (2006) [160] in the so-called Pickup-and-Delivery Problem with Time Windows and Transshipment, which is a PDP characterized for the presence of transshipment points, where the vehicles can drop some objects or split their loads to allow other vehicles to pick them up later. The motivation for this problem was a courier company, based in San Francisco (US), that allowed transshipment of loads between vehicles to keep the drivers in their home areas. The authors present an empirical study to identify circumstances under which transshipment may be beneficial, they consider applications where the requests are completely known in advance and their loads are relatively small in comparison with the vehicle's capacity. The main contributions of this paper are a construction-improvement heuristic to solve some random clustered instances, and an analysis of the impact in costs and fleet size of several policies to locate the transshipment points.

One year later, Thangiah et al. (2007) [201] addressed the Split-Delivery Pickup-and-Delivery Time Window Problem with transfers (SDPDTWP), which is a PDP where split-delivery is allowed (i.e., the same origin or destination of a request can be served by more than one vehicle), each request has a service time window that must be respected (i.e., hard time windows), and requests can be transferred between vehicles. The authors developed insertion heuristics and solved both static and real-time instances. Because the heuristics were focused on low running times, they can be used in real-time applications although without any guarantee of quality.

Nakao and Nagamochi (2008) [164] is a theoretical paper that analyzes the maximum travel cost that can be saved by introducing a single transshipment point to the Pickup-and-Delivery Problem. The authors obtained bounds that are proportional to the square root of the number of cycles in an optimal solution and also to the square root of the number of requests. They also constructed an instance that achieves tightly one of the bounds.

An exact model for the Pickup-and-Delivery Problem with Time Windows and Transshipment was proposed by Deschamps et al. (2012) [64]. This exact model is a complex mixed integer linear programming formulation, and it was validated on a set of ten particular instances whose optimal solution was alternatively computed by an ad hoc method. The authors confirm that the introduction of transshipment points can improve cost savings and give more flexibility for solving the problem because in some instances there is no solution at all if we do not include transshipment points. The price to pay for these attractive features is an even larger solution space that makes the exact problem harder to solve; the authors bear out some prohibitive computation times from just five transportation requests.

The first exact method for solving some instances of the PDPT was presented by Cortés et al. (2010) [50]. In their paper, the authors construct a complex mixed integer linear programming formulation, and they confirm that works correctly by solving one example instance with a method based on Benders decomposition.

In Bouros et al. (2011) [36], the authors addressed a dynamic Pickup-and-Delivery Problem with Transfers (that they abbreviate as dPDPT). They proposed a graph-based formulation that treats each request independently as a Constrained Shortest Path Problem. One of the main difficulties of the problem considered is that the cost of the paths does not exhibit the so-called subpath optimality, this is, subpaths of an optimal path are not necessarily optimal subpaths. Due to this limitation, it is not possible to use some fast shortest paths algorithms. Hence, the authors build a graph that they call the "dynamic plan graph" and propose a label-setting algorithm to explore the solutions space. They compare this approach against a relatively conventional local search algorithm based on insertion heuristics and tabu search, and conclude that their method is significantly faster with the inconvenience that solutions quality is marginally lower.

Masson et al. (2013) [157] developed an adaptative large neighborhood search (ALNS) algorithm for the Pickup-and-Delivery Problem with Transfers. The ALNS uses destruction heuristics, insertion heuristics, and a simulated annealing accepting criterion. The insertion heuristics were developed with the objective of efficiently inserting requests through transfer points. The adaptative aspect of the ALNS algo-

rithm consists in associating scores with neighborhoods, and use those scores within a roulette wheel selection procedure for choosing neighborhoods: at the beginning all the neighborhoods start with the same weight, and each time a neighborhood is called, its score is updated depending on its performance. The score of a neighborhood is incremented: in a quantity σ_1 if it leads to a new best solution, in a quantity σ_2 if the generated solution is better than the current accepted solution, and in a quantity σ_3 if the new solution is accepted and appears for the first time. The authors evaluated the algorithm on a set of ten real-life instances and on the set of instances proposed by Mitrović-Minić and Laporte [160]. The use of transfers improved up to 9% the solutions of real-life instances, and the ALNS algorithm improved the results of Mitrović-Minić and Laporte [160].

The computational complexity of checking the feasibility for the insertion of one request in the PDPT was studied by Masson et al. (2013) [156]. In this work, the authors determined that if we perform some preprocessing of the current state of a PDPT network, we can test the feasibility of insertions in constant time, and that the complexity to update the preprocessed information after insertion/deletion of one request is quadratic in the size of the network.

Coltin and Veloso (2014)[48] proposed a very large neighborhood search with transfers (VLNS-T) algorithm for solving the PDPT. The algorithm is based on the VLNS algorithm for the PDP introduced by Ropke and Pisinger [186], but allows multiple transfers for items at arbitrary locations, and is not restricted to a set of predefined transfer points. The VLNS-T uses simulated annealing in its main loop. It starts by choosing a random current schedule, and uses deletion/ insertion heuristics to generate "neighboring" schedules, and search for a better current schedule. The best times for those schedules are computed by using the second step of the algorithm proposed by Cordeau and Laporte in [56], and its time-feasibility is checked by constraint propagation. Coltin and Veloso applied the VLNS-T algorithm over some benchmark PDP problems to verify how the use of transfers improves the best known solution costs, and tested the algorithm over instances constructed from real world taxi data in New York City.

The Selective Pickup-and-Delivery Problem with Time Windows and Paired Demands (SPDPTWPD)[6] is a version of the classical PDP with time windows with a selective aspect that arises when it is not possible to honor all the requests (e.g., a company with a small fleet of vehicles). Each request is then associated with some profit, and the aim of the problem is to maximize the total profit with a minimal routing cost.

Peng et al. (2019)[8] studied the Selective Pickup-and-Delivery Problem with Transfers (SPDPT) which is a PDP that results from a combination of the Selective Pickup-and-Delivery Problem with Time Windows and Paired Demands (SPDPTWPD) and the Pickup-and-Delivery with Transfers (PDPT). For dealing with the SPDPT the authors elaborated a strategy for encoding solutions and developed a particle swarm optimization (PSO)♣ algorithm with local search and insertion heuristics. The authors generated a set of 25 new instances, and modified the instances proposed by Al Chami et al. (2017) [7] by adding transfer points. The proposed algorithm obtained good quality solutions (in comparison with the solutions without transfers) in a relatively small solving time.

Applications of transfers have been mainly related to transportation of people with limited mobility (see [154, 158, 184, 191]). But there are also some works related to the impact of transport on the environment, for example, Andini et al. (2019) [10] analyzed empirically the utility of introducing a single transfer point to reduce traffic congestion.

A more recent application of transfers to reduce carbon emissions in a Pickup-and-Delivery network can be found in Xue (2022) [220]. This paper proposes a two-stage heuristic framework to model the problematic and generate high-quality delivery routes. It also provides a case study from Dalian (China).

♣The PSO is an algorithm introduced by Eberhart and Kennedy [77] that improves a candidate solution by having a population of candidate solutions (particles), and moving these particles around in the space of variables according to simple mathematical formula over the particle's position and velocity. It was first intended for simulating the movement of organisms in a bird flock or fish school.

Table 1.1: Overview of some papers about PDP with transshipment or transfers. Abbreviations: CAP=finite capacity, ST=static, DY=dynamic, SEL=selective, TRSH= transhipment, TRSF=transfers, SPL=split-load, FS = fleet size, TPR=total profit (requests), SR=satisfied requests, RC=routing cost, RTV=ride time vehicles, RTR=ride time (requests), WTR=waiting time (requests), TWV=time windows violations, STW=soft time windows, and HTW=hard time windows.

Reference	Type	Objective	Constraints	Algorithm	Size
Theoretical papers					
Nakao and Nagamochi (2008) [164]	TRSH, ST	RC	1 transhipment point	Upper bounds on travel cost that can be saved by introducing a single transshipment point	–
Masson et al. (2013)[156]	TRSF, ST	–	HTW, 1 transfer at most	Computational complexity of checking the feasibility for the insertion of one request	–
Exact methods					
Contardo et al. (2010)[50]	TRSF, ST	FS, RTV+ RTR+ WTR, TWV	STW	A MILP formulation, a branch-and-cut + Benders decomposition algorithm	≤6 req.
Deschamps et al. (2012)[64]	TRSH, SPL, ST	RC	HTW	A MILP formulation	≤7 req.
Heuristics					
Mitrović-Minić and Laporte (2006) [160]	TRSH, SPL, ST	RC	HTW	Construction-improvement heuristic	≤100 req.
Thangiah et al. (2007)[201]	TRSF, SPL, ST and DY	FS, RC	HTW	Insertion and reactive heuristics	–
Bouros et al. (2011)[36]	TRSF, DY	RTV+ RTR+ WTR	–	BFS with label-setting over an auxiliary graph	–
Xue (2022)[220]	TRSF, ST	RTV+ RTR	–	Two-stage heuristic (clustering + local search)	≤100 req.
Metaheuristics					
Masson et al. (2013) [157]	TRSF, ST	–	CAP, HTW	An ALNS algorithm with insertion/remotion heuristics and a simulated annealing accepting criterion	≤193 req.
Coltin and Veloso (2014)[48]	TRSF, ST	RC+ RTV+ SR	CAP, HTW	A VLNS algorithm (simulated annealing + remotion/insertion heuristics + time constraint propagation)	≤144 req.
Peng et al (2019)[8]	TRSF, ST, SEL	max TPR and min RC	HTW	A hybrid particle swarm optimization (PSO + local search and insertion heuristics)	≤57 req.

1.1.2 Dial-A-Ride Problem

In previous section, we have seen Pickup-and-Delivery Problems that are usually related with goods transportation; for people transportation the name *Dial-A-Ride Problem* (DARP) is preferred. The main difference between PDP and DARP is that the later usually takes user inconvenience into consideration. There exists several ways of modeling the user inconvenience in a DARP, but the most common ones consist in the introduction of new terms in the objective function or the incorporation of additional sets of constraints. This type of problems arises commonly from situations related to the transportation of handicapped or elderly persons; but there are also other applications like, e.g., transportation of perishable goods, that require maximum ride time limits.

Historically, the Dial-A-Ride Problems received more attention long before Pickup-and-Delivery Problems for goods transportation. The first references about DARP in the literature date back to the late 1960s and the 1970s (cf. Wilson and Weissberg (1967) [219], Wilson et al. (1971) Wilson et al. [218], Rebibo (1974) [183], and Wilson and Colvin (1977).

Since the early literature, PDP and DARP have used different terminology and notation (e.g., in DARPs we usually speak of clients or customers instead of transportation requests). A first attempt to generalize Pickup-and-Delivery Problems in unified notation was proposed in Savelsbergh and Sol [189]. In this paper, the authors introduce a more general problem called the General Pickup-and-Delivery Problem (GPDP) and survey the related problems and the solution methods published until 1995.

The Single Vehicle Case

In the literature the single vehicle DARP is sometimes referred as SDARP (see Parragh et al. [170]), and it can be seen as an extension of the classical Traveling Salesperson Problem (TSP). An early reference related to the incorporation of precedence constraints into the TSP is Lokin (1979) [150]. In this paper, the author analyzes some situations that occur in real-life transportation problems, and that impose some precedence relations between the visits of some cities (e.g., a cluster of cities must be visited contiguously, some cities must be visited first than others, etc). The author proposes several sets of constraints that can be included into MILP formulations of the TSP to impose those precedence constraints, and propose some branch-and-bound methods for solving the resulting formulations.

Around 1980, Psaraftis [178] introduced the first exact algorithm for solving a SDARP. The considered problem involves a generalized objective function that consists of a weighted combination of the time to service all customers, and the wait and ride times of each customer (to model the degree of "dissatisfaction" experienced by each customer). The author proposes a dynamic programming algorithm based in a backward recursion, and uses that algorithm to solve SDARP instances with up to nine requests. Three years later, in Psaraftis [179], the same author proposed a modified version of the above algorithm to solve a SDARP with time windows and with the objective of minimizing the time needed to service all customers. This algorithm is based in a forward recursion and requires the same computational effort as the previous one (namely $O(n^2 3^n)$ for n customers). Furthermore, it is capable to detect infeasible problem instances.

Another forward dynamic programming algorithm for a SDARP was proposed by Desrosiers et al. (1986) [67]. The considered problem includes time windows to model user inconvenience, and aims to minimize the total distance traversed by the vehicle. The authors use the two dimensional (time, cost) labeling that they had previously used in Desrosiers et al. (1983) [65], and introduce some criteria to eliminate infeasible states (i.e., states which are incompatible with vehicle capacity, precedence, or time window constraints). The proposed algorithm achieve running times that in practice increase linearly with the number of considered requests. The largest instance solved by the authors comprises 40 requests.

Sexton and Bodin (1985) [193, 194] proposed several heuristics for the SDARP in a two parts paper. In the first part [193], the authors presented a mathematical programming formulation of the problem, and proposed a Benders decomposition procedure to attack the problem through an alternation between a routing component and a scheduling component. In the second part [194], the authors proposed two heuristics: one for building an initial route, and one for improving the route sequence. They also describe results of some computational experiments on problem instances with up to 20 requests.

Kubo and Kasugai (1990) [145] provide a comparative survey about early local search heuristics for the SDARP. They also proposed three heuristic algorithms: a sequential insertion algorithm based on the algorithm proposed by Jaw et al. in [131], an adaptation of the classical nearest neighbor method with a randomized routine, and a space-filling curve heuristic which is a simplified version of an asymptotically optimal heuristic developed by Stein in [200].

Now we proceed to discuss the literature related to multi-vehicle DARP.

The Multi-Vehicle Case

In 1984, Kikuchi [138] addressed a DARP related to a demand-responsive transportation system with the objective of minimizing empty vehicle travel and idle time. The author proposes a methodology that uses a linear programming Transportation Problem whose supply and demand vectors contain both location and time elements. This Transportation Problem is used to minimize the empty vehicle travel and idle time. The methodology also yields the minimum fleet size to serve the passenger trip requests.

In the early 2000s, Cordeau [54] proposed a complex 3-index mixed integer linear programming formulation for a static DARP with a homogeneous fleet. The author introduced some sets of valid constraints and used them to develop a branch-and-cut algorithm for solving the proposed formulation. That branch-and-cut algorithm was used for solving to optimality problem instances with up to 36 requests. Later, the formulation was extended to handle a DARP with heterogeneous fleet and passengers.

Around 2007, Ropke et al. [187] proposed two 2-index integer formulations and additional valid inequalities (called the Strengthened Capacity constraints and the Fork constraints) for a static DARP. Those formulations can be adapted to handle problems with either a homogeneous fleet or a heterogeneous fleet. The authors designed two branch-and-cut algorithms for solving the proposed formulations, and solved to optimality some problem instances with up to 96 requests.

One of the first heuristics developed for static multi-vehicle DARP was published around 1981 by Cullen et al. [58]. The authors propose a cluster-first-route-second approach: they first obtain a collection of "good" clusters which represent promising segments of a vehicle route, and then those clusters are "chained" into complete vehicles routes. The authors proposed two partitioning models (one for generating improving clusters, and one for identifying better chains), and used column generation approaches for dealing with them.

Around 1986, Jaw et al. [131] proposed a heuristic based on a sequential insertion procedure to assign customers to vehicles and to determine a time schedule of pickups and deliveries for each vehicle. The authors describe also the results of computational experiences with the algorithm. This can be considered one of the first works dealing with large-scale DARP instances, because the authors worked on an instance with 20 vehicles and 2617 customers.

Ioachim et al. [129] developed a two phase method involving a clustering heuristic and a column generation algorithm for solving a DARP related to a door-to-door

handicapped transportation problem. In the first phase of the method, the authors build an auxiliary network and set a Pickup-and-Delivery Problem with Time Windows with multiple vehicles (m-PDPTW). That m-PDPTW is solved by adapting the column generation exact algorithm from Dumas et al. (1991) [75]. As a result, they obtain a set of mini-clusters. In the second phase, those mini-clusters are used to set a Traveling Salesperson Problem with Time Windows with multiple vehicles (m-TSPTW). The m-TSPTW is also solved by using the column generation algorithm from [75]. The solutions of this last problem yield the itineraries for the vehicles. The authors compared the resulting mini-clustering algorithm of the first phase against a relatively standard parallel insertion heuristic for mini-clustering, and concluded the column generation based method outperforms the parallel insertion heuristic on the internal traveling time (i.e., the travel time necessary to cover the internal distance within the mini-clusters) by an average of 9.7%. The authors used the two phase method for handling instances with up to 2,545 requests.

Potvin and Rousseau [173] proposed a parallel route building heuristic for a DARP. In this problem, the number of vehicle is not predetermined, and the overall objective is to service the customers while minimizing total travel distance and waiting time. To determine an initial number of routes, the authors used a sequential insertion algorithm from Solomon (1987) [198]. Then, the parallel routes are initialized by selecting the farthest customer from the depot in each route created by the Solomon insertion algorithm, and the remaining customers are inserted once at time by computing its best feasible insertion place. The next customer to be inserted is selected by using a "generalized regret measure" (see Tillman et al. (1972) [204]) over all routes. The regret measure chosen by the authors is based on the gap between the best insertion place for a customer and its best insertion places on other routes. The main idea of the heuristic is that the unrouted customers with large regrets must be considered first (because the number of interesting alternative routes for inserting them is small) and the customers with small regret measures can be considered later for insertion (because they can be easily inserted into alternative routes without loosing much). The authors tested the heuristic over the Solomon's instances [198] and concluded the parallel insertion performs better on not purely clustered instances.

Madsen et al. [153] proposed an insertion based algorithm for solving a DARP with a multiobjective function involving fleet size, total driving time, total waiting time, and penalties related to soft time windows and to vehicles with unutilized capacity. The proposed algorithm is called REBUS, and was focused on an efficient insertion for dynamic environments (e.g., requests are treated in less than one second).

Healy and Moll [116] proposed an local search algorithm for Dial-A-Ride Problem. The authors developed a "sacrificing" algorithm which is an extension of the classical local search algorithm. The main idea is to use an additional cost function to obtain a broader view of solution quality. The proposed algorithm alternates between both cost functions. When the algorithm uses the secondary cost function, it may visit solutions which are better with respect to the secondary cost, but that may be poorer with respect to the original cost function (the authors call this a "sacrifice move").

Bondörfer et al. (1999) [35] proposed a cluster-first-route-second approach for solving a DARP. The clustering problem is modeled and solved optimally as a Set Partitioning Problem (see Balas and Padberg (1976) [18]). The routing subproblems are also modeled as Set Partition Problems, but they are solved approximatively by a branch-and-bound algorithm that uses only a subset of all possible tours.

Surveys about DARP can be found, for example, in Cordeau and Laporte (2003) [55], Cordeau and Laporte (2007) [57], Agatz et al. (2012) [1], Doerner and Salazar-González (2014) [71], Molenbruch et al. (2017) [161], and Ho et al. (2018) [122].

Now we provide a review of some works related to the DARP with transfers.

DARP with Transfers

Transfers were mentioned for the first time in the DARP literature around 1978 by Stein [199, 200]. These pioneer articles contain only theoretical results. In [199], the author develops an asymptotic probabilistic analysis of some vehicle routing problems to estimate the length of an optimal solution that picks up and delivers n passengers from random locations to random destinations within a planar bounded region R with area a, and when n goes to infinity. The study considers a uniform probability distribution over R to generate the random points. The author denotes by Y_n the length of an optimal tour and concludes that $\lim_{n\to\infty} \frac{Y_n}{\sqrt{n}} = c\sqrt{a}$ *almost everywhere*[◇], with $\sqrt{2b} \leq c \leq 2b$, taking b as the *Traveling Salesperson constant*[▽] (cf. Beardwood et al. [24]). In the case which there are k vehicles that travel at unit speed, the author proves that the minimal total distance travelled by all k vehicles converges *almost everywhere* to $\frac{c\sqrt{a}}{k}$. If transfers are allowed, then the minimal time-to-delivery the final passenger also converges almost everywhere to $\frac{c\sqrt{a}}{k}$. In [200], Stein describes a class of heuristic algorithms for solving some DARPs, and proves those algorithms are asymptotically optimal in a probabilistic sense.

[◇]The convergence *almost everywhere* is a concept from Measure Theory. It means pointwise convergence on a subset of the domain whose complement has measure zero.

[▽]The precise value of b has not been determined mathematically yet. Some experimental results point toward an ultimate value around 0.712... (see [171, 132]).

Exact methods. Hou et al. (2016) [124] presented a MILP formulation for a DARP with transfers related to an application for electric taxis. The authors called such a problem the Transfer-Allowed Shared eTaxis (TASeT). The objective of the TASeT problem is to maximize the number of passengers that can be served by a given taxi fleet during a given period of time. Each passenger is allowed to make at most one transfer between taxis per trip at the charging stations that are scattered throughout the city. The need for recharging eTaxis is also considered: an eTaxi will be send to charge its battery while waiting for a transfer passenger or when its battery level is too low to serve the next passenger. The MILP model was used to solve a set of small instances with up to four vehicles and nine requests.

Pierotti and Theresia van Essen (2021) [172] proposed two very complex MILP formulations of a Dial-A-Ride Problem with transfers. One formulation is based on continuous time, whilst the other one uses a discrete time. Both formulations rely on a 2-commodity flow (one for requests and one for vehicles), and rely on the idea of tracking request flow and forcing vehicle flow to be compatible with that request flow. The objective function aims at minimizing a generalized costs involving eight terms: travel cost, penalty for every passenger time travel, penalty for early and late departure, penalty for early and late arrival, penalty related to loss of quality every time there is a transfer, penalty of passenger waiting time at transfer nodes, parking costs for vehicles, and penalty for every unserved request. The MILP formulations are very general, but they contain 13 types of variables and 37 types of constraints. The authors tested the formulations for solving real-life instances involving twenty cities (from the Netherlands), up to four vehicles, and up to eight requests. Due to the difficulty of the proposed formulations some of the small instances were not solved to optimality.

Heuristics. Shang and Cuff (1996) [197] developed a heuristic for a Pickup-and-Delivery Problem♠. The problem involves time windows and a multiobjective function that aims to minimize routing cost, tardiness and travel time. A particularity of this problem is that it does not consider a predetermined fleet size. The proposed heuristic either receives the number of vehicles as a parameter or starts by estimating an appropriate number of vehicles necessary to satisfy the requests. Items are clustered according to their due times and then ordered in increasing order of their ready times. Then, during the main loop, a new item is selected and the algorithm tries to incorporate the current item into existing schedules. The algorithm tries first to incorporate the item into a single vehicle route and if it is not possible, then seeks to incorporate it using a transfer between vehicles. If those incorporations are

♠The problem studied in this article involves service time windows and a multiobjective function for reducing the tardiness of the system. For those reasons, the authors mention that the problem can be classified as a DARP.

not possible, then the procedure selects the next k items in the current cluster (k is a parameter) and proceed to construct a miniroute. For constructing a miniroute, the algorithm integrates the current item with the next k items (one at time), while using three different integration strategies. After the integration, a list of alternative miniroutes is constructed, those routes are evaluated and the best cost miniroute is assigned to any vacant vehicle for creating a new schedule. The algorithm continues until no more integrations are possible. The authors analyzed the behaviour of the heuristic (evaluating the decrease of tardiness, travel time, and number of vehicles needed) over instances with up to 300 requests

Deleplanque and Quilliot (2013) [61] studied the Dial-A-Ride Problem with time windows, transshipments, and dynamic transfer points, which is a DARP including the possibility of one transfer from a dynamic transfer point by request (such a transfer point is computed at the same time as the request is included in a vehicle planning). The authors proposed an algorithm based on techniques of insertion and constraint propagation.

Hou et al. (2016) Hou et al. [124] presented a greedy heuristic for the previously defined TASeT problem, and used the heuristic to perform a large-scale evaluation over a set of instances with up to 350 requests. They also presented a case study in the city of Shanghai, China. The study concluded that TASeT solutions can significantly improve the number of passengers served by eTaxis by 118% in comparison with non-shared taxis, and by 35% in comparison with a no-transfers system. Additionally, the number of taxis can be reduced by up to 41%.

Andini et al (2019)[10] performed an empirical analysis about the importance of introducing a single transfer point to reduce traffic congestion. They consider a dynamic PDP (which can also be classified as a DARP) where customer location and service time are random variables that are realized dynamically during the plan execution, and use an insertion heuristic for solving some problem instances. The numerical results they obtained show the transfer point allows to fulfill on average 16.7% more requests.

Metaheuristics. Masson et al. (2011) [154] proposed a tabu search metaheuristic for solving the Dial-A-Ride Problem with Transfers (DARPT) that was motivated by a practical case of school bus routing for handicapped children in France. Later, the same authors proposed in [155, 158] an adaptive large neighborhood search (ALNS) metaheuristic and explained how to check the feasibility for the insertion of one request. They also evaluated the proposed method on a set of real-life instances, and on a modified set of DARP instances from Cordeau and Laporte (2003) [56].

Shönberger (2017) [191] studied a Dial-A-Ride Problem involving two fleets of vehicles operating over two distinct regions and with a central transfer hub where all the passenger have to pass when they travel from a pickup location on one region to a destination location on the other region. This paper introduced a set of "transfer scheduling" constraints for maintaining changing times on an acceptable level. The author proposed a memetic algorithm (a kind of genetic algorithm) that uses a procedure for sampling the search space, and then combine the sampled individuals to obtain a set of routes. The genetic search is then enhanced by a schedule building procedure that postpones waiting times at selected locations (if necessary) in order to meet the transfer scheduling constraints.

Table 1.2: Overview of papers about DARP with transfers. Abbreviations: CAP=finite capacity, ST=static, DY=dynamic, SEL=selective, TRSH= transshipment, TRSF=transfers, SPL=split-load, FS = fleet size, TPR=total profit (requests), SR=satisfied requests, RC=routing cost, RTV=ride time vehicles, RTR=ride time (requests), WTR=waiting time (requests), TWV=time windows violations, STW=soft time windows, and HTW=hard time windows.

Reference	Type	Objective	Constraints	Algorithm	Size
Exact methods					
Hou et al. (2016) [124]	TRSF, ST	SR	HTW	A MILP formulation	≤9 req.
Pierotti and Theresia van Essen (2020) [172]	TRSF, ST, SEL	TPR+ RC+ RTV+ RTR+ WTR	CAP, HTW	Two MILP fomulations	≤8 req.
Heuristics					
Shang and Cuff (1996) [197]	ST, TRSF	FS, RC + RTR + WTR	HTW	A heuristic (item clustering and integration of items in miniroutes)	≤300 req.
Deleplanque and Quilliot (2013) [61]	TRSF, static	SR, RTV + RTR	HTW, 1 transfer at most	Insertion heuristics with constraint propagation	≤96 req.
Hou et al. (2016) [124]	TRSF, ST	SR	HTW	Greedy insertion heuristic	≤350 req.
Metaheuristics					
Masson, Lehuedé, and Péton (2011)[154]	TRSF, ST	RC	HTW	A tabu search algorithm	–
Masson, Lehuedé, and Péton (2012)[158]	TRSF, ST	RC	HTW	An ALNS algorithm	≤144 req.
Schönberger (2017)[191]	TRSH, ST	RC	CAP, HTW	A memetic algorithm	≤200 req.
Hamouda et al. (2020) [114]	TRSH, ST	RC	CAP, HTW	A simulated annealing algorithm	≤200 req.

More recently, Hamouda et al. (2020) [114] proposed a simulated annealing algorithm combined with four neighborhood search methods for handling the same DARP introduced in Shönberger (2017) [191]. The authors tested the algorithm on the set of instances proposed by Shang et al. (1996) [197] and concluded their algorithm performs slightly better than the memetic algorithm proposed by Shönberger in [191].

1.1.3 Relocation Problem

The Relocation Problem is a VRP involving the transportation of identical objects on a transit network. Typical applications of this problem occur in the context of vehicle rental (e.g., bike/car sharing networks) or inventory repositioning. However, the problem arises naturally in any situation involving the redeployment of non-consumable resources from places where they are not being used to places where they are needed. It can be set in the following way.

In the initial state of the problem, a graph is given, and each vertex of the graph may contain a multiset of objects of a unique type. A final state, specifying the multisets of objects desired at each vertex, is also given. A fleet of vehicles is available for shipping objects among the vertices. The Relocation Problem consists in computing an optimal set of routes for the vehicles in the fleet (with respect to certain objective function that may involve the number of used vehicles, the length of vehicle/objects routes, etc). The computed routes must allow the fleet vehicles to accomplish the rearrangement of the objects while following the assigned routes.

Note that, because this problem may involve the transportation of a single type of identical objects between multiple stations, it can be seen as a one-commodity many-to-many Pickup-and-Delivery Problem with unpaired pickups and deliveries. For this reason, in the literature this problem is also known by the name Pickup-and-Delivery Vehicle Routing Problem (PDVRP) with unpaired pickup and delivery points (see Parragh et al. [170]). In the case of a single vehicle, the problem is usually referred as the Capacitated Traveling Salesperson Problem with Pickups and Deliveries (CTSP-PD) (see Anily and Bramel [11]), the One-Commodity Pickup and Delivery Traveling Salesperson Problem (1-PDTSP) (see Hernández-Pérez and Salazar-González [117]), the Traveling Salesperson Problem with Pickup and Delivery (see Hernández-Pérez and Salazar-González [118]), or the Pickup and Delivery Traveling Salesperson Problem (PDTSP) (see Parragh et al. [170]).

We describe next some works related to Relocation Problems, and we survey the solution methods that have been used to tackle those problems. We classify those solution methods into exact, heuristic, and metaheuristic approaches. Because some

articles contain several of those approaches, we also provide a comparative overview in Table 1.3 (single vehicle case) and Table 1.4 (multi-vehicle case).

Exact methods. The first exact method proposed for a single vehicle Relocation Problem was introduced by Hernández-Pérez and Salazar-González (2003, 2004) [117, 118]. It is a branch-and-cut algorithm that uses a construction-improvement heuristic to obtain feasible solutions from the LP relaxations at each node of the branch-and-bound tree. The construction phase is an adaptation of the nearest insertion algorithm for the TSP, and the improvement phase consists in applications of 2-opt and 3-opt exchanges. The authors tested their algorithms on adaptations of the instances used in Mosheiov (1994) [163] and Gendreau et al. (1999) [102], which contain up to 75 customers.

Raviv et al. (2013) [182] proposed MILP formulations for a static multi-vehicle Relocation Problem (vehicle non-preemptive and load preemptive) with the objective of scheduling vehicle routes for balancing the system while minimizing routing costs and user dissatisfaction.

Contardo et al. (2012) [51] addressed a load preemptive/vehicle non-preemptive Relocation Problem for balancing a dynamic public bike-sharing system during peak-hours. The authors proposed an arc-flow formulation on a Time-Expanded network, and applied Dantzig-Wolfe decomposition for deriving two MILP formulations. The authors solved one of the formulations using a column generation algorithm and used the other formulation in a primal heuristic to find good solutions. As a result, the authors provided a methodology for computing lower and upper bounds in short computing times. The authors tested their methodology on a set of random/clustered instances with up to 100 stations distributed in a plane, with integral coordinates between 0 and 60 (inclusive). They also considered a time horizon of two hours, discretized with two different granularities (24 periods of five minutes, and 60 periods of two minutes). The experiments confirmed that the algorithm based on column generation can find better bounds in shorter computing times in comparison to solve the arc-flow formulation directly with a MILP solver. They also showed that the optimality gaps are usually bigger for random instances than for clustered instances.

Chemla et al. (2013) [45] studied the dynamic C-delivery TSP, which is a static Relocation Problem involving a single vehicle of capacity C. They proposed an exact algorithm based on column generation where columns represent feasible vehicle routes together with sequences of pickup/delivery actions. The proposed algorithm involves a pricing problem that is solved by dynamic programming.

Heuristics. One of the early works involving a Relocation Problem is Anily and Hassin [12]†. This paper introduces a Vehicle Routing Problem called "The Swapping Problem" which is stated over a weighted graph and consists in the computation of a shortest route for a single vehicle with unitary capacity. In the initial state, each vertex of the graph may contain an object of a known type, and it is also given a final state specifying the type of object desired at each vertex. The vehicle should attain the rearrangement of the objects from the initial state to the final state, shipping and dropping objects while following the computed route, and without exceeding its capacity. Anily and Hassin proved the Swapping Problem is $\mathcal{NP}$-hard by providing a polynomial reduction to the Taveling Salesperson Problem (TSP), exhibited some structural properties of the shortest routes, and developed polynomial approximation algorithms with a guarantee ratio of 2.5 (established by using the heuristic of Christofides (1976) [47] for the TSP). Those approximation algorithms are based on the "patching algorithm" of Gilmore and Gomory (1964) [104] for the TSP, which is a two-step heuristic where we first solve a set of Assignment Problems to construct a collection of subpaths, and then we "patch" those paths to produce a single route.

The first work that considered a Relocation Problem for multiple vehicles is Dror et al. (1998) [73]. The authors studied a vehicle/load preemptive problem (involving a weak synchronization mechanism) in the context of self-service electric cars. They proposed a MILP formulation of the problem (with a pseudo-polynomial size) and used both constraint programming and Lagrangian relaxation methods for solving small instances in an exact way. For solving practical size applications the authors developed a heuristic based on the A* algorithm of Hart et al. (1968) [115].

Chalasani and Motwani (1999) [43] studied Relocation Problems that can be seen as instances of the following k-delivery TSP: given n source points and n sink points in a metric space, with exactly one item at each source, find a tour by a vehicle with finite capacity k to pickup the items from the source points and deliver exactly one item to each sink point. The authors start by studying the unit capacity case (i.e., 1-delivery TSP) and show that the problem of "finding a tour with at most twice the cost of an optimal tour" is equivalent to the problem of "finding a minimum-weight bipartite spanning tree such that the vertices of one part have degree at most two" and in turn, this last problem can be seen as the problem of "finding a minimum-weight maximum-cardinality independent set that is common to the matroid‡ of all bipartite forests and to the matroid of all bipartite subgraphs having a part that consists of vertices with degree at most two". This problem, in turn, is a particular case of the matroid intersection problem (see Edmonds (1970) [79]). Because

†Unfortunately the official digital sources of this article contain an incomplete document. Page 432 appears duplicated in place of page 423.

‡Oxley [169] provides a gentle introduction to matroids.

Brezovec et al. (1988) [38] proved that there exist polynomial-time independence oracles© for the above-mentioned matroids, a 2-approximation algorithm is deduced. The authors use the algorithm for the unit capacity case together with lower bound arguments to obtain a 9.5-approximation algorithm for the general case.

Anily and Bramel (1999) [11] also studied the k-delivery TSP and improved the work of Chalasani and Motwani [43] to obtain a (7-$\frac{3}{k}$)-Approximation algorithm. They also developed a heuristic called MATCH^k that provides a better worst-case bound for many practical values of k (for example, for $k \leq 385$, they obtain approximation algorithms with worst-case bounds that are less than or equal to seven).

Hernández-Pérez and Salazar-González (2004) [119] studied the One-Commodity Pickup-and-Delivery Traveling Salesperson Problem (1-PDTSP). They developed a construction-improvement heuristic that uses a greedy construction procedure, that is followed by applications of 2-opt, and 3-opt exchanges. The authors also proposed a second heuristic which is based on incomplete optimization: they apply the algorithm proposed in [119] over a reduced set of variables associated with "promising" edges.

Lim et al. (2005) [149] studied the Capacitated Traveling Salesman Problem with Pickups and Deliveries (CTSP-PD) over trees and proved that is $\mathcal{NP}$-hard. They also developed a 2-approximation algorithm with a worst time complexity of $O[n^2/\min(n,k)]$. The algorithm consists of a recurrent construction process that builds a series of route sets for all vertices, from the leaves to the root of the tree instance.

Benchimol et al. (2011) [26] is a study motivated by the installation of the Vélib system (a "bike hire" system) in Paris with 1,500 stations, almost 20,000 bicycles, and more than 70,000 travels each day. The authors addressed a static Relocation Problem, called the C-delivery TSP, which involves a single vehicle of finite capacity C. They proved the problem is $\mathcal{NP}$-hard by providing a reduction to the Partition Problem (see Karp (1972) [136]), and adapted the 9.5-approximation algorithm of Chalasani and Motwani (1999) [43] to obtain also a 9.5-approximation algorithm for the C-delivery TSP. They also proposed a 2-approximation algorithm for the particular case of a complete graph with unit costs, and a polynomial algorithm for solving the problem in the case of a tree graph.

Dell'Amico et al. (2014) [62] studied a bike-sharing rebalancing problem (load preemptive/vehicle non-preemptive) where a fleet of capacitated vehicles is employed to relocate the bikes while minimizing routing cost. They proposed four MILP

© An independence oracle of a matroid is any subroutine that takes as its input a subset of the ground set of the matroid, and returns as output a Boolean value: `true` if the given set is an independent set of the matroid, and `false` otherwise.

formulations (with an exponential number of constraints) of the problem. In order to handle those formulations, the authors proposed several branch-and-cut algorithms involving the separation of two new types of clique inequalities, a directed version of the infeasible path constraints proposed by Hernández-Pérez and Salazar-González (2004) [118], and an extension of the tournament constraints proposed by Ascheuer et al. (2000) [16]. The authors tested the proposed algorithms in real data sets of twenty-two bike sharing systems from Italy, Ireland, the USA, Canada, Mexico, Argentina, and Brazil; and they also tested a set of random instances The largest instances involved up to 116 stations.

Krumke et al. (2014) [144] studied a Relocation Problem in the context of a carsharing system, where the cars are partly autonomous. The authors considered an online problem (vehicle and load preemptive) and proposed a 2-commodity flow MILP model based on a Time-Expanded network (TEN). The model is solved approximatively with a two-phase heuristic. In the first phase the authors compute the arcs of the Time-Expanded network which are likely to be used in an optimal solution (this is done by solving two linear programs: one for carrier flow and the other for car flow). Then, in the second phase, they construct a reduced Time-Expanded network (keeping only the arcs computed in the phase one), and compute an optimal solution on this reduced network. Note that the considered problem has a selective aspect because the objective is to decide which customers requests can be satisfied without spending more costs in the relocation process than gaining profit by satisfying them. The authors analyzed 1,080 randomly generated instances with 50-99 stations; 550-990 cars; 5-15 carriers with capacity 20; time horizons between 60 and 240 time units; and 500-8,000 customers requests.

Zhang et al. (2017) [222] examined a dynamic bicycle repositioning problem (load preemptive and vehicle non-preemptive) that considers inventory level forecasting, user arrival forecasting, bicycle repositioning, and vehicle routing using a multicommodity Time-Expanded network flow model with a non-linear objective function. They also proposed a methodology for linearizing that model and obtain an equivalent MILP model. The resulting MILP is solved with a heuristic algorithm: the authors solve the LP relaxation of the MILP to detect a subset of "potential" arcs (arcs carrying a positive amount of flow in the LP solution), then they enumerate all the potential routes involving those potential arcs and assign loading sequences to each route. Some of those routes are fixed and assigned to vehicles, and then the LP relaxation is solved iteratively while discarding the routes already fixed. The authors tested the proposed methodology on three data sets with up to 200 stations.

Bsaybes et al. (2018, 2019) [40, 41] analyzed a Relocation Problem motivated by the managing of a fleet of individual public autonomous vehicles that operate in a closed site for supplying an internal transportation service. They addressed the offline/online and preemptive/non-preemptive versions of the problem. For the offline problem they proposed an exact MILP formulation involving multicomodity coupled flows in a Time-Expanded network. Because such a formulation is difficult to solve in the practice, the authors reduced the Time-Expanded network to contain only promising arcs and then solved the formulations resulting from such reduced networks. For solving the online version of the problem, the authors propose three heuristics. The first one constructs tours in an incremental way, and the other two use one of two strategies while solving a sequence of offline problems for certain time intervals and on adapted Time-Expanded networks. They also provided a competitive analysis of the online problem, and tested the offline algorithms over two sets of 180 random instances each, and with up to 290 requests.

Metaheuristics. Chemla et al. (2012) [44] addressed the C-delivery TSP. They proposed a branch-and-cut algorithm for solving a relaxation of the problem and obtain a lower bound for the optimal cost of the problem. They also proposed a tabu search algorithm for obtaining an upper bound for the optimal cost of the problem. The authors tested the algorithms on a set of instances with up to 100 stations with demands between -10 and 10.

Rainer-Harbach et al. (2013) [181] studied a static Relocation Problem (vehicle and load preemptive) related to a bike sharing system. They proposed a general variable neighborhood search (VNS) algorithm that generates candidate routes for vehicles to visit unbalanced stations. Then, the number of bikes to be loaded or unloaded at each station are derived by using one of three alternative methods: a greedy heuristic, a maximum flow calculation, and a linear programming model. The authors tested the proposed algorithms over a set of real world instances from the Vienna Citybike system (involving up to five vehicles and 90 stations).

Di Gaspero et al. (2013) [68] studied a static Relocation Problem (without transfers) in the context of bike sharing systems. They proposed a constraint programming formulation and a hybrid approach which combines constraint programming techniques with an ant colony optimization algorithm. The authors validated the proposed approach over a set of real world instances from the Vienna Citybike system (involving up to five vehicles and 90 stations). The authors concluded that the performance of the proposed approach is not as good as those achieved by other metaheuristic approaches.

Table 1.3: A comparative overview of papers about Relocation Problems involving a single vehicle.

Reference	Type	Objective	Constraints	Algorithm	Size
Anily and Hassin (1994)[12]	–	routing cost	capacity 1	A 2.5-approximation algorithm	–
Chalasani and Motwani (1999)[43]	–	routing cost	capacity κ, N items, n stations	A 9.5-approximation algorithm (with respect to $O(\kappa + n \log N)$)	-
Anily and Bramel (1999)[11]	–	routing cost	capacity κ	Two approximation algorithms	-
Hernández-Pérez and Salazar-Gómez (2003, 2004)[117, 118]	–	routing cost	capacity κ	An exact branch-and-cut algorithm	$\leq$75 req.
Hernández-Pérez and Salazar-Gómez (2004)[119]	–	routing cost	capacity κ	Two heuristics: construction-improvement, and branch-and-cut over a set of promising arcs	$\leq$75 req.
Lim, Wang, and Zu (2005)[149]	for tree graphs	routing cost	capacity κ	A 2-approximation algorithm.	–
Benchimol et al. (2011) [26]	-	routing cost	capacity κ, N items, n stations	A 9.5-approximation algorithm (with respect to $O(\log \kappa + n \log N)$)	–
Chemla et al. (2012) [44]	-	routing cost	capacity κ, n stations	A branch-and-cut heuristic and a tabu search algorithm	$n \leq 100$
Chemla et al. (2013) [45]	dynamic	routing cost	capacity κ, n stations	A column generation algorithm	$n \leq 250$

Table 1.4: A comparative overview of papers about Relocation Problems involving multiple vehicles.

Reference	Type	Objective	Constraints	Algorithm	Size
Dror, Fortin, and Roucairol (1998)[73]	static, vehicle and load preemptive	routing cost	capacity κ	A pseudopolynomial MILP formulation, an A* algorithm	≤25 req
Contardo et al. (2012) [51]	dynamic, load preemptive, vehicle non-preemptive	minimize unmet demands	heterogeneous fleet, n stations	An arc-flow and two MILP formulations; a branch-and-cut and a heuristic algorithm	$n \leq 100$
Raviv et al. (2013)[182]	static, load preemptive, vehicle non-preemptive	routing cost + users' dissatisfaction	heterogeneous fleet, n stations	Two MILP formulations	$n \leq 104$
Rainer-Harbach et al. (2013) [181]	static, vehicle and load preemptive	min imbalance + number of loads/ unloads + routing cost	capacity κ, n stations	A variable neighborhood search algorithm	$n \leq 90$
Di Gaspero et al. (2013) [68]	static, load preemptive, vehicle non-preemptive	max balance with min routing cost	capacity κ, n stations	A CP formulation and a hybrid algorithm (CP + ant colony optimization)	$n \leq 90$
Dell'Amico et al. (2014) [62]	static, load preemptive, vehicle non-preemptive	routing cost	capacity κ, n stations	Four MILP formulations and a branch-and-cut algorithm	$n \leq 116$
Krumke et al. (2014) [144]	online, selective, vehicle and load preemptive	total profit	capacity κ, n stations	A two-phase heuristic (a 2-commodity flow MILP over a reduced TEN)	$n \leq 99$
Zhang et al. (2017) [222]	dynamic, load preemptive, vehicle non-preemptive	routing cost + user dissatisfaction	capacity κ, n stations	A Time-Expanded network flow model, a MILP formulation, and a heuristic	$n \leq 200$
Bsaybes et al. (2018) [40]	offline/ online, vehicle preemptive/ load non-preemptive	maximize (profit - cost)	capacity k	A 2-commodity flow MILP in a Time-Expanded network, three online heuristics	≤290 req.
Bsaybes et al. (2019) [41]	dynamic, vehicle and load non-preemptive	minimize unmet demands	capacity 1	A MILP involving a flow in a Time-Expanded network	$n \leq 100$

1.1.4 Combined Vehicle Routing and Scheduling Problem

In the Combined Vehicle Routing and Scheduling Problem with Time Windows (VRSP-TW) there is a given fleet of vehicles in a depot, and a set of customers to be serviced. Each customer has associated a time window for the arrival time, and a duration for the visit. The objective is usually to optimize a function which evaluates service quality and economical cost.

Typical applications occur when two vehicles must meet at a point at the same time (strong synchronization) or when a vehicle cannot pick up a load before another vehicle has delivered the same load.

For example, Fagerholt and Christiansen (2000) [82] used a single vehicle variant of the VRSP-TW to solve a ship scheduling application. The solution method consisted of two phases. First, the authors generate a collection of candidate schedules for each ship in the fleet. Then, those candidates schedules are brought into a set partitioning model and solved by commercial optimization software.

In 2008, Bredström and Rönnqvist [37] presented a mixed integer linear programming model for the Combined Vehicle Routing and Scheduling Problem with Time Windows. The model contains two types of variables: binary routing variables and rational scheduling variables. It also contains eight types of constraints: six of them correspond to a multiple Traveling Salesperson Problem and the two remaining were introduced by the authors to model pairwise temporal precedences and pairwise synchronization between customers visits. The authors presented also a heuristic which uses the linear relaxation of their mathematical programming model to iteratively choose/discard subsets of promising arcs of the network. The explicit introduction of the proposed temporal and precedence constraints allowed the optimization library to solve to optimality 33 of the 75 smaller proposed instances, within a time limit of one hour. In contrast, the proposed heuristic found 29 optimal solutions within a time limit of two minutes. It is important to note that, although those instances involve twenty customers, four vehicles, and two synchronizations, they give rise to MILP models with 1,900 variables and 2,100 constraints. For the remaining bigger instances (which give rise to MILP models with more than 27,000 variables and more than 28,000 constraints), the MILP library was not able to find any feasible solution within the one hour time limit, and the authors only analyzed the behavior of the heuristic on some of those instances with an overall time limit of ten minutes. Based on those results, the authors argue that, if we are interested only in obtaining good feasible solutions, then it does not seem more difficult to solve the problem with synchronization constraints that solve the problem with the relaxed constrains.

1.2 Methods

In this section, we present a brief discussion of some topics, techniques, and algorithms that will be used later in this work. We only present a brief discussion of general topics, and provide some concise examples of models and algorithms.

1.2.1 The A* Search Algorithm

The A* search algorithm is a graph traversal algorithm that was introduced by Peter Hart, Nil Nilsson, and Bertranm Raphael in [115]. It can be considered as an extension of Dijkstra's algorithm (see Appendix A.5.3 or Dijkstra (1959) [70]) to search for the minimum cost path in very large graphs. The main idea behind this algorithm is to incorporate some heuristic information to guide the search. For some problems, this results in an efficient search procedure that guarantees the optimality of the solution found.

Let us recall that an *arc progression* W from x_1 to x_{k+1} in a digraph $G' = (X', A')$ is a finite sequence $(x'_1, a'_1, x'_2, \ldots, x'_k, a'_k, x'_{k+1})$ such that $k \geq 0$, and $a'_i = (x'_i, x'_{i+1}) \in A'$ for $i = 1, \ldots, k$.

Let us consider a simple digraph $G' = (X', A')$ with a distinguished vertex x'_0. We define a digraph $G = (X, A)$ by taking X as the set of all the finite arc progressions in G' starting from x'_0. We add one arc (x, y) to G if and only if there exists an arc $a' = (x', y') \in A'$ such that y can be obtained by extending x with the arc a'. We call every $x \in X$ a *state* and we say that G is a *graph of states*.

Suppose that we are given an initial state x_0 and a final state x_ν, and that we aim to find a minimum cost path from x_0 to x_ν. Suppose also that we have two functions $g, h : X \to \mathbb{R}$, such that, for all $x \in X$, $g(x)$ is the *cost* of a path from x_0 to x, and $h(x)$ is the *estimated cost* of a minimum cost path from x to the x_ν. The A* search consists of a best-first search that uses the evaluation function $f(x) = g(x) + h(x)$ to guide the search.

The value $g(x)$ corresponds to the exact cost of moving from x_0 to state x, and the value $h(x)$ is an approximation of the cost increment that would result from moving from x to x_ν. So, the function g is defined in a unique way, but we can use any heuristic as the function h. These functions g and h are usually called the *current cost function* and the *estimated completion cost function*, respectively.

Given a state x, let us denote by $g^*(x)$ the minimum cost of a path from x_0 to x, by $h^*(x)$ the minimum cost of a path from x to x_ν, and define $f^*(x) = g^*(x) + h^*(x)$. We have the following.

Lemma 1.1 (Subpath optimality)**.** *Let c^* be the cost of a minimum cost path P^* from x_0 to x_ν, and let x be any vertex of P^*, then $g(x)$ is the smallest possible cost of a path from x_0 to x. In other words, we have that $g(x) = g^*(x)$.*

Proof. By definition of $g^*(x)$, we have that $g^*(x) \leq g(x)$. We will show that $g^*(x) = g(x)$. Suppose, to the contrary, that $g^*(x) < g(x)$, then there exists a path P from x_0 to x with cost $g^*(x)$. The path $P + P^*_{[x,x_\nu]}$ (i.e., the path P extended by the subpath of P^* from x to x_ν) has cost $g^*(x) + h^*(x) < g(x) + h^*(x) = c^*$ contradicting the optimality of P^*. ■

Now, the optimality of an A* algorithm depends on certain properties of the heuristic function h. For example, we say that a heuristic is *admissible* if never overestimates the cost to reach the final state. That is, a function h is admissible if, for every state x, it provide us with a lower bound for the cost of a minimum cost path from x to the final state.

Note that, if h is an admissible heuristic then, for every state x, the cost $f(x)$ is an optimistic estimation of the cost of a minimum cost path that continues from state x to the final state x_ν.

We also have the following result.

Theorem 1.1 - Optimality of the A* algorithm
Let suppose that we have a finite system of states represented as a digraph $G = (X, A)$, with a given final state x_ν that is reachable from a given initial state x_0. Consider an A algorithm that uses a state evaluation function $f = g + h > 0$, with g and h defined as in the previous paragraphs. Then, if the heuristic function h is admissible, the solution found by the A* algorithm is optimal.*

Proof. We proceed by contradiction. Suppose that the optimal path P^* from x_0 to x_ν has cost c^*, but the algorithm has ended with a x_0-x_ν-path P of cost c with $c > c^*$. Because the algorithm has not found the optimal state, we have that there exists some vertex $x \neq x_\nu$ that is in the optimal path P^*, and that was not expanded. Let $g^*(x)$ be the cost of a minimum cost path from x_0 to x, and $h^*(x)$ be the cost of a minimum cost path from x to the final state x_ν. We have the following.

- $c \leq f(x)$, because otherwise, we would have $c > f(x)$ and the algorithm would have not finished yet;
- $f(x) = g(x) + h(x)$ by definition of f;
- $g(x) + h(x) = g^*(x) + h(x)$ by Lemma 1.1, because x is on the optimal path P^*;

- $g^*(x) + h(x) \leq g^*(x) + h^*(x)$ because h is admissible (i.e., the value $h(x)$ may underestimate the true cost $h^*(x)$);

- $g^*(x) + h^*(x) \leq c^*$, because in fact $g^*(x) + h^*(x) = c^*$.

Putting all together, we have that $c \leq f(x) = g(x) + h(x) = g^*(x) + h(x) \leq g^*(x) + h^*(x) \leq c^*$. But we initially have supposed that $c > c^*$. This contradiction completes the proof.

■

Example 1.1 - Computation of a shortest path with an A* algorithm
The map of France in Figure 1.2 shows the approximated location of twelve French cities. The numbers correspond to the approximate straight-line distances in kilometers from each city to the City of Paris.

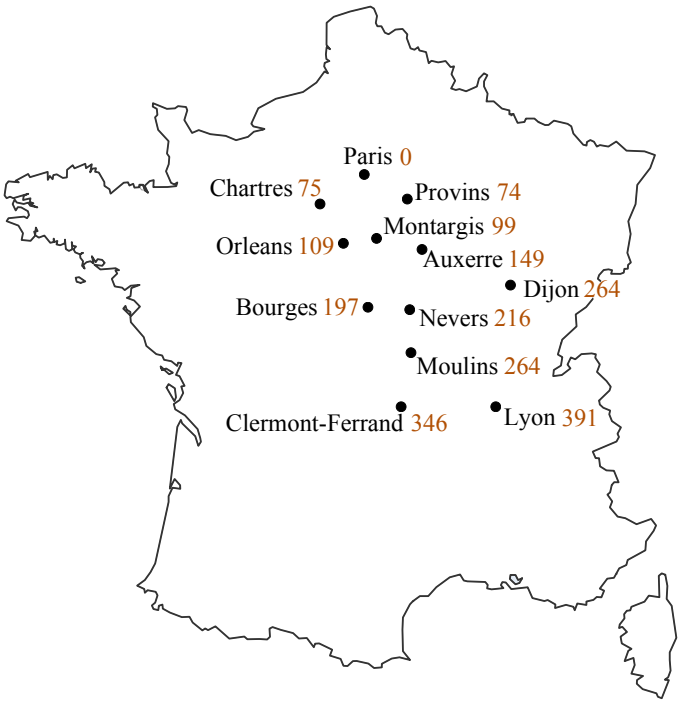


Figure 1.2: Location of twelve French cities and their straight-line distances (in kilometers) to the City of Paris.

Table 1.5 provides us with the shortest highway distance in kilometers between some of those cities.

Table 1.5: The shortest distances in kilometers of the highway trajects between some pairs of French cities (values were taken from Google Maps [108]).

from \ to	Auxerre	Bourges	Chartres	Clermont-Fd	Dijon	Lyon	Montargis	Moulins	Nevers	Orleans	Paris	Provins
Auxerre	-	-	-	-	-	-	-	-	-	-	-	106
Bourges	138	-	-	-	-	-	118	-	-	122	-	-
Chartres	-	-	-	-	-	-	-	-	-	-	96	-
Clermont-Fd	-	190	-	-	-	167	-	105	-	-	-	-
Dijon	150	-	-	-	-	-	-	-	-	-	-	-
Lyon	-	-	-	-	197	-	-	204	-	-	-	-
Montargis							-					
Moulins	-	100	-	-	182	-	-	-	56	-	-	-
Nevers	119	-	-	-	-	-	137	-	-	166	-	-
Orleans			78	-	-	-	-	-	-	-	-	-
Provins											89	-

The straight-line distance between two cities is clearly an optimistic lower bound for the shortest highway distance between those cities, because in real-life, due to geographical accidents, roads almost always have curves.

Figure 1.3 represents the progress of an A* algorithm to compute a shortest highway path from Clermont-Ferrand to Paris, considering only the highway trajects that are indicated in Table 1.5. Each rectangle represents a state of the system and corresponds to a path starting at Clermont-Ferrand and ending at the city indicated by the label inside of the rectangle.

The cost $f(s)$ of a state s corresponding to a city x is calculated by adding the costs $g(s)$ and $h(s)$; where $g(s)$ is the shortest highway distance from Clermont-Ferrand to city x, and $h(s)$ is the straigth-line distance from city x to Paris. For example, in Figure 1.3 (b), the state with the label "Lyon" has an estimated cost of **558**, and this value was computed as the sum of 167 (the shortest highway distance from Clermont-Ferrand to Lyon) and 391 (the straight-line distance from Lyon to Paris).

Note that, at each stage, we choose a state s with the minimum estimated cost $f(s)$ and then we expand s to create new states. For example, in Figure 1.3 (b), the state with the smallest estimated cost is Moulins, so it is the next state to expand. Figure 1.3 (c) shows the system after expanding Moulins. Note that, we have three new states with the labels "Bourges", "Dijon", and "Nevers". The estimated cost of the state with the label "Nevers" is **377 km**, and it is computed as the sum of the shortest highway distance from Clermont-Ferrand to Moulins (105 km), plus the shortest highway distance from Moulins to Nevers (56 km), plus the straight-line distance from Nevers to Paris (216 km).

Finally, we observe the A* algorithm finishes when it reaches the final state "Paris" which corresponds to a highway path of length **424 km**. Because the straight-line distance is an admissible heuristic, we can assert that the solution found by the algorithm is optimal. □

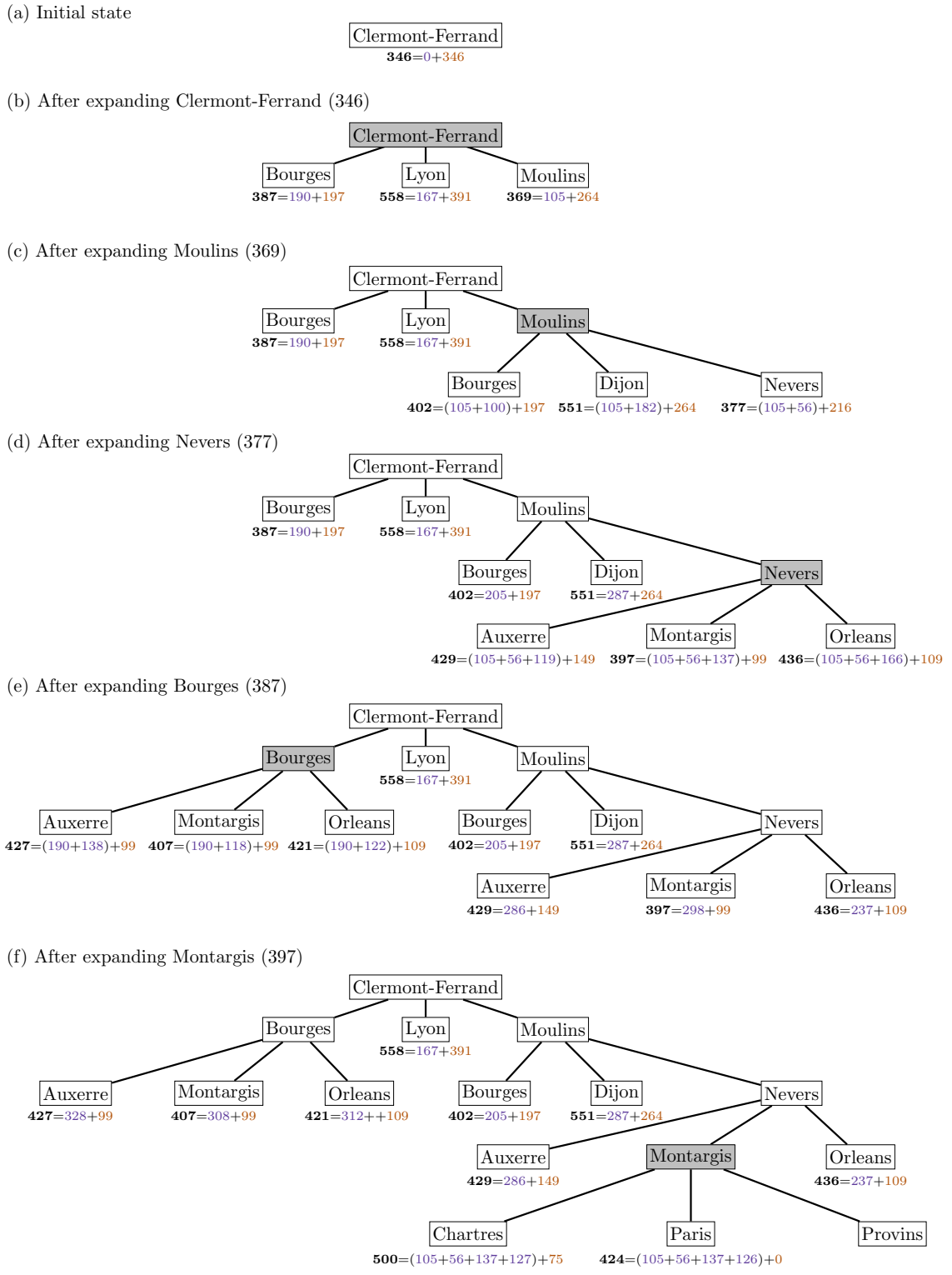


Figure 1.3: Stages in a A* search for Paris, starting at Clermont-Ferrand. States are labelled with the values $f = g + h$. The g values were taken as the shortest highway distances (see Table 1.5) and the h values were taken as the straight-line distances from each city to Paris (see Figure 1.2).

Dechter and Pearl (1985) [60] examined some properties of heuristic best-first search strategies whose evaluation functions depend on all the information available from each candidate path, and not merely on the current cost function g and the estimated completion cost function h. They show that under certain assumptions several properties of A* can be preserved. The authors also introduced an interesting notion of optimality for comparing best-first search algorithms: an algorithm A *dominates* an algorithm B with respect to a set of instances $\mathcal{I}$, if and only if for every $I \in \mathcal{I}$, the set of nodes expanded by A is a subset of the set of nodes expanded by B. Using this definition of domination, the authors examined the class of algorithms that, like A*, return optimal solutions when all cost estimates are optimistic. They proved that no optimal algorithm exists for that class, but if we consider only the class of best-first search algorithms that are guided by path-dependent evaluation functions, then A* is indeed optimal (i.e., A* dominates all the algorithms in that class).

We close this section by mentioning the A* has been used extensively for tackling a wide range of problems. As a matter of fact, the original paper of Hart, Nilsson, and Bertram [115] has more than 12,800 citations.

1.2.2 Methods Based in Branch-and-Bound

Branch-and-bound is an algorithmic general framework for solving discrete and combinatorial optimization problems, as well as mathematical optimization problems. It was first proposed by Ailsa Land and Alison Doig from the London School of Economics in [146], while they were carrying out a research sponsored by British Petroleum in 1960 to develop discrete programming models for the transport and storage of petroleum.

In this work we are going to use some methods based in branch-and-bound for solving mixed integer linear programs. So we start this section by providing some terminology and notation about linear programs.

A *mixed integer linear program* (MILP) is a problem of the form

$$\begin{array}{lrll} \text{minimize} & \mathbf{cx} + \mathbf{hy} & & \\ \text{subject to} & \mathbf{Ax} + \mathbf{By} & \leq \mathbf{b} & \\ & \mathbf{x} & \geq 0 \text{ integral} & \\ & \mathbf{y} & \geq 0. & \end{array} \tag{P1}$$

where $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ and $\mathbf{h} = (h_1, \ldots, h_p)$ are two row vectors, $\mathbf{A} = (a_{ij})$ is an $m \times n$ matrix, $\mathbf{B} = (b_{ij})$ is an $m \times p$ matrix, and $\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$ is a column vector. We assume that all entries of $\mathbf{c}$, $\mathbf{h}$, $\mathbf{A}$, and $\mathbf{B}$ are rational numbers. The columns vectors $\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ and $\mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix}$ contain the variables to be optimized. The variables x_j are constrained to be nonnegative integers whilst the variables y_j are allowed to take nonnegative real values.

The expression $\mathbf{cx} + \mathbf{hy}$ is called the *objective function* of (P1) and the set $S = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : \mathbf{Ax} + \mathbf{By} \leq \mathbf{b}\}$ is called the set of *feasible solutions* to (P1). Given a feasible solution $s = (\mathbf{x}, \mathbf{y}) \in S$ we define the *cost* of s as the value $\mathbf{cx} + \mathbf{hy}$ and we denote it by $cost(s)$. If $(\mathbf{x}, \mathbf{y})$ is a feasible solution of (P1) that minimizes the expression $\mathbf{cx} + \mathbf{hy}$, then $(\mathbf{x}, \mathbf{y})$ is also a feasible solution of (P1) that maximizes the expression $-(\mathbf{cx} + \mathbf{hy})$, so it is possible to transform any maximization problem into a minimization one.

If we take the problem (P1) without considering $\mathbf{h}$, $\mathbf{B}$, and $\mathbf{y}$ we obtain a *pure integer linear program*. In contrast, if we take (P1) without considering $\mathbf{c}$, $\mathbf{A}$, and $\mathbf{x}$ we obtain a *linear program*. Linear programs are important because there exist algorithms to solve them efficiently[☼] in theory and practice.

Now we define the *natural linear programming relaxation* of (P1) as the problem

$$\begin{array}{lrl} \text{minimize} & \mathbf{cx} + \mathbf{hy} & \\ \text{subject to} & \mathbf{Ax} + \mathbf{By} & \leq \mathbf{b} \\ & \mathbf{x} & \geq 0 \\ & \mathbf{y} & \geq 0. \end{array} \tag{P2}$$

Note that in this case, vector $\mathbf{x}$ is not constrained to be integral. So, the optimal cost of a feasible solution for (P2) is a lower bound for the optimal cost of a feasible solution for (P1). Let us denote by $S' = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{R}_+^n \times \mathbb{R}_+^p : \mathbf{Ax} + \mathbf{By} \leq \mathbf{b}\}$ the set of feasible solutions of (P2).

An inequality $\alpha\mathbf{u} \leq \beta$ is *valid* for a set $K \subseteq \mathbb{R}^m$ if it is satisfied by every $\mathbf{u}' \in K$.

Let $s = (\mathbf{x}^*, \mathbf{y}^*)$ be a feasible optimal solution of (P2). We may assume that $s = (\mathbf{x}^*, \mathbf{y}^*)$ is a basic[☽] optimal solution of (P2). If s is not a feasible solution of

[☼] In the sense of computational complexity, see Appendix A.4

[☽] This is a very technical concept from linear programming theory. For our present discussion, it is sufficient to know that given an optimal feasible solution of a linear program, there exist algorithms that allow us to find a basic feasible solution with the same cost.

(P1), then we can find an inequality $\alpha\mathbf{x} + \gamma\mathbf{y} \leq \beta$ that is valid for S but such that $\alpha\mathbf{x}^* + \gamma\mathbf{y}^* > \beta$.✠ Such an inequality $\alpha\mathbf{x} + \gamma\mathbf{y} \leq \beta$ is called a *cutting plane* separating $s = (\mathbf{x}^*, \mathbf{y}^*)$ from S.

Given a cutting plane $\alpha\mathbf{x} + \gamma\mathbf{y} \leq \beta$ separating $s = (\mathbf{x}^*, \mathbf{y}^*)$ from S, we define $S_1 = S' \cap \{(\mathbf{x}, \mathbf{y}) : \alpha\mathbf{x} + \gamma\mathbf{y} \leq \beta\}$. The linear programming relaxation of (P1) over S_1 is

$$\begin{array}{lrll} \text{minimize} & \mathbf{cx} + \mathbf{hy} & & \\ \text{subject to} & \mathbf{Ax} + \mathbf{By} & \leq \mathbf{b} & \\ & \alpha\mathbf{x} + \gamma\mathbf{y} & \leq \beta & \\ & \mathbf{x} & \geq 0 & \\ & \mathbf{y} & \geq 0. & \end{array} \tag{P3}$$

Since every solution in S satisfies $\alpha\mathbf{x} + \gamma\mathbf{y} \leq \beta$, we have that $S \subseteq S_1 \subset S'$ so we say that (P3) is *stronger* than (P2).

Let us return to the branch-and-bound method. For applying the branch-and-bound method to a minimization MILP problem, we need to perform iteratively the following two kinds of steps:

- "branch" step: given a subset of possible solutions, partition the subset into at least two nonempty subsets.
- "bound" step: for every subset obtained by branching iteratively, compute a lower bound on the cost of any solution within the subset.

At the beginning of the branch-and-bound algorithm, we have a set S consisting of all feasible solutions. Note that such a set is usually given in a implicit way (e.g., $S = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : \mathbf{Ax} + \mathbf{By} \leq \mathbf{b}\}$ is the set of feasible solutions of (P1) but is not an explicit set of solutions). Also, we initialize an upper bound variable `U` as $+\infty$ or as any other valid value (e.g., a heuristic solution of the problem).

The branch-and-bound algorithm constructs a search tree T rooted at S by performing iteratively a branch step followed by a sequence of bound steps. During the branch-and-bound process, we call the *incumbent* to the best feasible solution found by so far at a given moment.

At each step, the branch-and-bound algorithm chooses an active (i.e., non-explored) node of T corresponding to a (implicit) set S of feasible solutions, and partitions such a set into a collection of (implicit) subsets of feasible solutions, i.e.,

✠The existence of such an inequality is guaranteed when $(\mathbf{x}^*, \mathbf{y}^*)$ is a basic feasible solution of (P2).

$S = S_1 \sqcup \ldots \sqcup S_t$. Then, for each $i = 1, \ldots, t$ the algorithm computes a lower bound $\mathtt{L}_i$ on the cost of any solution in S_i. Such a bound is usually computed by solving the linear relaxation of the MILP in S_i and we have one of the following cases.

1. Pruning by optimality: If while computing the lower bound $\mathtt{L}_i$ we find a feasible solution $s \in S_i$ with cost $\mathtt{L}_i$, then we deduce that s is a best cost solution in S_i. In this case, we can redefine $S_i = \{s_i\}$, and mark resulting node as non-active. In case $cost(s_i) < \mathtt{U}$, then we update the incumbent by setting $s^* = s_i$ and we update the best known cost by setting $\mathtt{U} = cost(s_i)$.

2. Pruning by bound: If we have $\mathtt{L}_i \geq \mathtt{U}$ then we deduce that S_i does not contain feasible solutions with a better cost than the incumbent. In this case, we mark node S_i as non-active.

3. Pruning by infeasibility: If we find that $S_i = \varnothing$, then we discard S_i.

4. New node: If we are not in any of the previous cases, then we have that $|S_i| > 1$ and $\mathtt{L}_i < \mathtt{U}$. In this case, we add a new active node S_i and a new arc (S, S_i) to the directed tree T.

The algorithm stops when there are no more active nodes to explore. A pseudocode of the whole above process is shown in Algorithm 1.

Algorithm 1: Branch-and-bound algorithm

Input : An instance of a minimization MILP problem.
Output: An optimal solution s^* (if any).

1 Set $T \leftarrow (V(T) = \{S\}, E(T) = \varnothing)$, where S is the set of all feasible solutions.
Mark S as active and set the incumbent $s^* \leftarrow$ `Nil`.
Set the upper bound $\mathtt{U} \leftarrow +\infty$ (or to any other finite upper bound available).
2 Choose an active vertex S' of the tree T (if there is none, **stop** and **return** s^*).
Mark S' as non-active and find a partition $S' = S_1 \sqcup \ldots \sqcup S_t$.
Set $T \leftarrow \big(V(T) \cup \{S_1, \ldots, S_t\}, A(T) \cup \{(S', S_1), \ldots, (S', S_t)\}\big)$ `[Branch step]`
3 **for** $i = 1, \ldots, t$ **do**
  Find a lower bound $\mathtt{L}_i$ on the cost of any solution in S_i. `[Bound step]`
  if $S_i = \{s\}$, $cost(s) = \mathtt{L}_i$, and $\mathtt{L}_i < \mathtt{U}$ **then**
    $\mathtt{U} \leftarrow cost(s)$, $s^* \leftarrow s$, and mark S_i as non-active. `[Pruning by optimality]`
  if $\mathtt{L}_i \geq \mathtt{U}$ **then** mark S_i as non-active. `[Pruning by bound]`
  if $S_i = \varnothing$ **then** mark S_i as non-active. `[Pruning by infeasibility]`
  if $|S_i| > 1$ and $\mathtt{L}_i < \mathtt{U}$ **then** mark S_i as active. `[New node]`
4 Go to Step 2.

According to Williams [217], although branch-and-bound based methods have proved to be the most successful, in general, on practical MILPs, the basic branch-and-bound algorithm usually receives little attention in theoretical mathematical programming books, possibly because of its lack of mathematical sophistication compared with other methods like the Gomory cutting-plane algorithm.

The Gomory cutting-plane algorithm [107] was the first algorithm for solving general MILPs. It can be described as an iterative process that approximates the set of feasible solutions of a MILP as the set of feasible solutions of a linear program. Such a linear program is constructed initially as the linear relaxation of the MILP and then the approximation is improved by generating and introducing additional specific cutting planes called *Gomory cuts*.

Ralph Gomory invented this method around 1957 for solving pure integer programming problems and provided a way for obtaining a finite algorithm. Later, Gomory extended the algorithm for solving MILPs. However, this and other cutting-plane algorithms resulted to be impractical due to numerical instability and because they usually need an exponential number of cuts to make progress towards the solution.

As a result, the branch-and-bound method of Land and Doig dominated the practical world of MILP computation until mid-1990s, when Balas et al. [19] showed that cutting-plane methods can be very effective when combined with branch-and-bound methods. Resulting algorithms were hence called branch-and-cut methods.

Branch-and-Cut

The first branch-and-cut implementation for solving general MILPs was proposed by Balas et al. [19]. It adds several Gomory cuts at a time, reoptimizes the resulting linear program, performs a few rounds of cut generation, and incorporates this procedure in a branch-and-bound framework. Incorporated in this way, Balas et al. (1998) [31] performed numerical experiments on a set of 106 instances (taken from a library of customer and academic models) and they found that Gomory cuts speed up the branch-and-bound search by a factor of 2.5. Commercial integer programming solvers started to incorporate Gomory cuts and other types of cuts in 1999 [49].

Note the branch-and-cut methods implemented in the commercial solvers are of general purpose. In practice, we usually face MILPs with a combinatorial structure that admits other types of ad hoc cuts. In those cases, we can incorporate the generation of those cuts into the cutting component of a branch-and-but algorithm of general purpose.

A pseudocode of the whole above process is shown in Algorithm 2.

Algorithm 2: Branch-and-cut algorithm

Input : An instance of a minimization MILP problem.
Output: An optimal solution s^* (if any).

1 Set $T \leftarrow (V(T) = \{S\}, E(T) = \varnothing)$, where S is the set of all feasible solutions.
Mark S as active and set the incumbent $s^* \leftarrow$ `Nil`.
Set the upper bound $\mathtt{U} \leftarrow +\infty$ (or to any other finite upper bound available).
2 Choose an active vertex S' of the tree T (if there is none, **stop** and **return** s^*).
Mark S' as non-active and find a partition $S' = S_1 \sqcup \ldots \sqcup S_t$.
Set $T \leftarrow \big(V(T) \cup \{S_i\}, A(T) \cup \{(S', S_i)\}\big)$ `[Branch step]`
3 **for** $i = 1, \ldots, t$ **do**
  Find a lower bound $\mathtt{L}_i$ on the cost of any solution in S_i. `[Bound step]`
  Try to add some cutting planes for improving the value of $\mathtt{L}_i$. `[Cut step]`
  Update the value of $\mathtt{L}_i$. `[Bound step]`
  if $S_i = \{s\}$, $cost(s) = \mathtt{L}_i$, and $\mathtt{L}_i < \mathtt{U}$ **then**
    $\mathtt{U} \leftarrow cost(s)$, $s^* \leftarrow s$, and mark S_i as non-active. `[Pruning by optimality]`
  if $\mathtt{L}_i \geq \mathtt{U}$ **then** mark S_i as non-active. `[Pruning by bound]`
  if $S_i = \varnothing$ **then** mark S_i as non-active. `[Pruning by infeasibility]`
  if $|S_i| > 1$ and $\mathtt{L}_i < \mathtt{U}$ **then** mark S_i as active. `[New node]`
4 Go to 2.

As we have mentioned previously, the branch-and-bound algorithms for solving MILPs usually solve a natural linear programming relaxation for obtaining a bound on the optimal cost at each node. Because the quality of those bounds may have an important impact on the performance of a branch-and-bound algorithm, researchers began to consider alternative MILP formulations with stronger natural linear programming relaxations. This gave rise to the branch-and-price method.

Branch-and-Price

Branch-and-price is a method of combinatorial optimization for solving some mixed integer programming problems that typically contain a large number of variables. An introductory description of the method with some examples can be found in [21].

We start this section by describing the column generation algorithm which is one of the main components of the branch-and-price method.

Consider the LP problem

$$\begin{array}{lrl} \text{minimize} & \mathbf{cx} & \\ \text{subject to} & \mathbf{Ax} & = \mathbf{b} \\ & \mathbf{x} & \geq 0 \end{array} \tag{P4}$$

with $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^n$, and $\mathbf{A} \in \mathbb{R}^{m \times n}$ with linearly independet rows. Suppose that $\mathbf{A}$ is

given in an implicit way because its number of columns is so large that is impossible to store it in a computer.

There exist some iterative linear programming algorithms (like the revised simplex method) which, for generating a new basic feasible solution $\mathbf{x}'$ at any given iteration, only require the n columns related to the current basic feasible solution $\mathbf{x}$ and a column which is to enter the basis. So, if we find an efficient way for discovering a column x_i that yields a new basic feasible solution $\mathbf{x}'$ with improved cost, then it is possible to solve (P4) with a column generation algorithm.

Such an algorithm considers two problems: the master problem and the subproblem. The master problem is the original problem with only a subset of columns being considered. The subproblem is a new problem created to identify an improving column (i.e., a column that yields a new basic feasible solution with improved cost). A pseudocode of the column generation method is shown in Algorithm 3.

Algorithm 3: Column generation algorithm

Input : A feasible LP problem $P = \min\{\mathbf{cx} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}$.
Output: An optimal solution $\mathbf{x}^*$.

```
1 Initialize the master problem and the subproblem.
2 Find a solution x* of the master problem.
3 Solve the subproblem for finding an improving column.
4 if an improving column x_i is found then
5 |   Add x_i to the master problem and then go to Step 2
6 else
7 |   Return x*.
```

One of the most influential works about the column generation method was the paper of Gilmore and Gomory (1961) [103] on the cutting-stock problem. Another interesting example is Desrosiers et al. (1984) [66] where the authors used the column generation method for solving a routing problem with time windows.

Let us return to the branch-and-price method.

The branch-and-price method is based on a branch-and-bound schema in which, at each node of the search tree, we have a natural linear programming relaxation whose solution provide us with a bound for the cost of any MILP feasible solution on the node, but whose explicit formulation may be impossible to write due to time/ memory requirements. For this reason, some sets of columns are left out and we proceed to obtain/improve the linear programming relaxation solutions by using a column generation method.

At first glance we can roughly conceive the method as a hybrid of branch-and-bound with column generation methods. However, according to Johnson [133], the following drawbacks may arise.

- The structure of the pricing problem is delicate and it can be destroyed if we use the conventional integer programming branching on variables.
- Solving to optimality the natural linear relaxation problems at each node may be computationally expensive. So we may need to apply specific rules for managing the branch-and-bound tree and avoid the computation of the solution of many of those relaxations.

To finish this brief section, we mention an illustrative example of a branch-and-price algorithm for the Generalized Assignment Problem (GAP).

Example 1.2 - A branch-and-price model for the GAP
The Generalized Assignment Problem is the following 0, 1 pure integer linear program, defined by coefficients c_{ij} and t_{ij}, and capacities T_j, $i = 1, \ldots, m$, $j = 1, \ldots, n$.

$$\begin{array}{lrll} \text{maximize} & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} & & \\ \text{subject to} & \sum_{j=1}^{n} x_{ij} & = 1 & i = 1, \ldots, m \\ & \sum_{i=1}^{m} t_{ij} x_{ij} & \leq T_j & j = 1, \ldots, n \\ & \mathbf{x} & \in \{0,1\}^{m \times n}. & \end{array} \tag{GAP}$$

One interpretation of this model is as follows. Suppose that a hospital has n operating rooms and that there are m surgeries that must be scheduled during a given time period T. Let t_{ij} be the estimated time of operating on patient i in room j, for $i = 1, \ldots, m$, $j = 1, \ldots, n$. The aim is to maximize the utilization time of the operating rooms during the time period (note that this is equivalent to minimizing wasted capacity).

The first set of constraints are called the assignment constraints and mean that each patient i is operated exactly once. The second constraints are the capacity constraints on each of the operating rooms.

If we apply the Dantzig-Wolfe decomposition to previous formulation with the assignment constraints defining the master problem, and the operating room capacity constraints defining the subproblems, we obtain the following master problem.

$$\begin{array}{lrll} \text{maximize} & \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq K_j} \left(\sum_{1 \leq i \leq m} c_{ij} y_{ik}^j \lambda_{ik}^j \right) & & \\ \text{subject to} & \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq K_j} y_{ik}^j \lambda_k^j & = 1 & i = 1, \ldots, m \\ & \sum_{1 \leq k \leq K_j} \lambda_k^j & = 1 & j = 1, \ldots, n \\ & \lambda_k^j \in \{0,1\} & & j = 1, \ldots, n, \\ & & & k = 1, \ldots, K_j. \end{array}$$

where the first m entries of a column, given by $y_k^j = (y_{1k}^j, y_{2k}^j, \ldots, y_{mk}^j)$, form a feasible solution to the j$^{\text{th}}$ knapsack constraint

$$\sum_{1 \le i \le m} t_{ij} y_i^j \le T_j,\ y_i^j \in \{0,1\},\ i = 1, \ldots, m,$$

and where K_j denotes the number of feasible solutions of that constraint.

The reason for such a reformulation is that the natural linear programming relaxation of the Dantzing-Wolfe decomposition is stronger than the one obtained from the standard formulation. However, note that such a decomposition it is usually too big for writing the complete LP formulation in an explicit way. So, for solving the LP relaxation, we use a column generation algorithm that will be based on the solution of n knapsack problems (the j^{th} knapsack problem searches for a new assignment of patients to operating room j).

Once we have obtained an optimal solution for the cost of the natural linear programming relaxation of the master problem on some node, if the solution is not integral it would be necessary to branch over some variables in order to find an integral solution. Yet, we observe that a standard branching over the λ_k^j variable would create a problem along a branch where $\lambda_k^j = 0$, and because y_k^j represents a particular solution to the jth knapsack subproblem, setting $\lambda_k^j = 0$ means that this solution would be excluded. That usually would result in a bad branching strategy because it is possible that the next time that the jth knapsack problem is solved the optimal solution is precisely y_k^j, and in that case, it would be necessary to forbid that solution and search for an alternative solution of that knapsack problem. If we continue branching in that way, at depth ℓ in the branch-and-bound tree we would need to find an optimal solution of the jth knapsack problem that is different from the ℓ previous optimal solutions that we have found at each branching. As a result, the jth knapsack subproblem may become harder with every new branching.

An interesting idea to remediate that difficulty is to consider the current fractional solution with respect to the original variables and select a fractional variable x_{ij}. Then we use a branching rule that would correspond to branching on x_{ij}. That is, if $x_{ij} = 1$, then all the existing columns in the master problem that do not assign patient i to operating room j are deleted and patient i is permanently assigned to operating room j (i.e., variable y_i^j is fixed to 1 in the jth knapsack); in contrast, if $x_{ij} = 0$, all existing columns in the master problem that assign patient i to operating room j are deleted and patient i cannot be assigned to operating room j (i.e., variable y_i^j is removed from the jth knapsack problem). The application of such a branching rule implies that each of the knapsack problems contains one fewer variable after performing the branching, and so they become easier. □

1.2.3 Layered Graphs

Layered graphs are a modelization tool to (re)state optimization problems that allow an indexation over resources state values during the different steps of a process. Gouveia et al. (2019) [109] contains a relatively recent survey about layered graphs and several classifications of the literature.

Although it is difficult to point out which were the papers that used layered graphs for the first time, it seems clear that most of the early developments occurred in the context of Time-Expanded networks (see, e.g., [30, 90, 94, 106, 110, 212]). A Time-Expanded network (TEN) is a particular type of layered graph that is obtained by considering indexations of vertices over time values. We start this section by describing Time-Expanded networks.

To illustrate the construction of a Time-Expanded network, we are going to consider an example based on the Vehicle Routing Problem (VRP). In the VRP we have a digraph $G = (X, A)$ whose vertex set X contains a distinguished depot vertex d and customer vertices $\{x_2, \ldots, x_{\nu(G)}\}$. For every arc $a \in A$ we have a nonnegative travel time denoted by $time_G(a)$. Also, we associate with every vertex x in $X \setminus \{d\}$ a nonnegative demand b_x. Finally, we have a fleet of vehicles with capacity κ and a time horizon $[0, \Omega]$. We consider that a customer vertex x is served when some vehicle arrives to x and incorporates the demand b_x to its load. The aim is to determine a collection of tours for the vehicles in the fleet, in such a way that, all the customers are served, the capacity of the vehicles is respected at any moment, and the sum of the travel times of the arcs in the tours is minimized.

For the VRP, the vertex set of the related Time-Expanded network is obtained by replicating each vertex x for each admissible time value t, creating a vertex x_t. Every vertex x_t is associated with a particular visiting time t of customer x. Also, the Time-Expanded network includes arcs $(x_t, y_{t'})$ between pairs of replicated vertices x_t and $y_{t'}$ if the travel time from x to y is equal to $t' - t$ and the arc (x, y) is an arc of the original digraph. Example 1.3 illustrates a VRP instance and its related Time-Expanded network.

Example 1.3 - A Time-Expanded network built from a VRP instance
Figure 1.4 (a) depicts the transit network of a VRP instance with a depot d and five customers $\{u, w, x, y, z\}$. We consider a time horizon $\Omega = 7$ (i.e., seven time units) and a fleet of vehicles with capacity $\kappa = 6$. Figure 1.4 (b) depicts the corresponding Time-Expanded network. It is not difficult to show that the optimal solution for the VRP instance is given by the tours $\Gamma_1 = (d, u, w, x, d)$ and $\Gamma_2 = (d, z, y, d)$, which correspond, respectively, to the paths $(d_0, u_1, w_3, x_4, d_6)$ and (d_0, z_3, y_4, d_7) in the Time-Expanded network. □

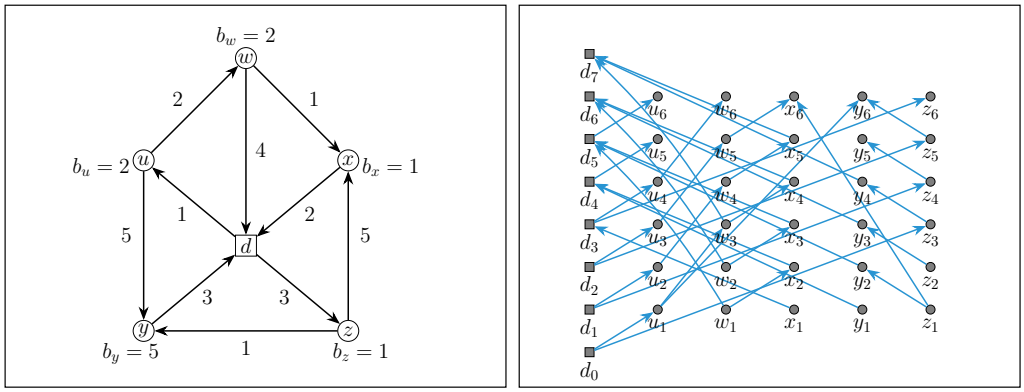


Figure 1.4: An example of Time-Expanded network. (a) The transit network of a VRP instance with a depot vertex d and customers $\{u, w, x, y, z\}$, a time horizon $\Omega = 7$, and a vehicle capacity $\kappa = 6$. Travel times are provided next to the arcs. (b) The Time-Expanded network constructed from the VRP instance in (a).

As we can see from Example 1.3, the size of a Time-Expanded network may become huge even for small VRP instances, yet we should not give up too easily. A closely look of the Time-Expanded network in Figure 1.4 (b) allows us to see that many arcs in the Time-Expanded network cannot be used by any feasible solution. This is because a valid tour for a vehicle corresponds to a path in the Time-Expanded network starting at d_0 and ending at an element of $\{d_1, d_2, \ldots, d_6, d_7\}$. So, by performing a breadth-first search (BFS) starting at vertex d_0, we can discard all the arcs of the Time-Expanded network that are not contained in any of those paths, and proceeding in this way, we obtain the simplified Time-Expanded network that is shown in Figure 1.5.

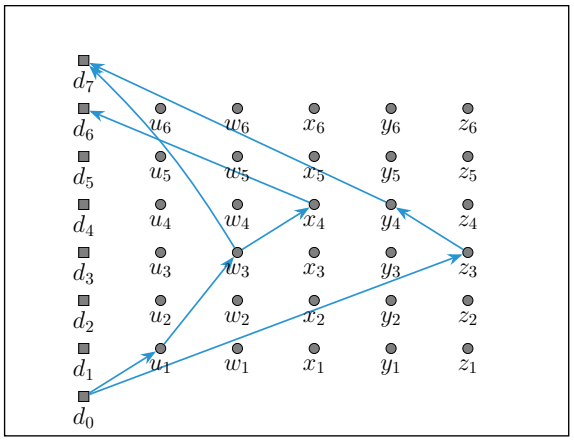


Figure 1.5: A simplification of the Time-Expanded network in Figure 1.4 (b).

The above kind of simplifications may have a drastic impact in our capability to solve formulations over Time-Expanded networks, for this reason, this is an important and active field of research (see for example [25], [40], or [144]).

Simplifications are often performed with one of the following objectives in mind.

- Preserve optimality: in this case, we only discard the incompatible or redundant arcs, yet the resulting network may be still too big and it could be necessary the use of ad hoc methods for solving the underlying formulations.
- Obtain a feasible solution: in this case, we not only discard the incompatible or redundant arcs, but we also discard (in a heuristic way) collections of arcs that seem less promising. Proceeding in this way, we obtain networks that are easier to handle, but we obtain approximated solutions.

A more recent approach that works in the reverse sense is the dynamic discretization discovery (DDD) algorithm. It was proposed by Boland et al. (2017) [32] in the context of a Service Network Design Problem and it was adapted later for solving the Traveling Salesman Problem with Time Windows (see [33]) and the Time-Dependent Traveling Salesman Problem with Time Windows (see [214]). The algorithm repeatedly solves a MILP formulation on a "partially" Time-Expanded network and refines that network based on an analysis of the solutions obtained. At the general step, we solve an auxiliary MILP which is based on a partially Time-Expanded network $\mathcal{D}_{\mathcal{T}}$ and whose solution provide us with a lower bound for the optimal cost of the original problem. If such auxiliary solution can be seen as a feasible solution of the original problem, then the solution is optimal and we are done; otherwise we create one or more variations of $\mathcal{D}_{\mathcal{T}}$ and we solve some related MILP problems to search for "converted" solutions that are feasible for the original problem. If one of those converted solutions has the same cost than the current lower bound, then we can assert that it is an optimal solution, otherwise we use the solution on $\mathcal{D}_{\mathcal{T}}$ to identify time points that can be added (together with some arcs) to $\mathcal{D}_{\mathcal{T}}$, and ensure an improved solution in the next iteration. Note that the nodes and arcs added at each iteration are chosen in such a way that they guarantee to preserve optimality. Gnegel et al. (2023) [105] provided a more general approach by embedding the DDD algorithm in a branch-and-bound framework and called branch-and-refine the resulting method.

So far in this section, we have only considered examples of networks that involve indexations over time values. Although time values give us an intuitive representation of the evolution of the states in a process, there is nothing special in this type of resource, and we can consider time as an abstract resource among others. Such an interpretation leads us directly to the concept of layered graph, which generalizes the concept of Time-Expanded network. Example 1.4 shows a way in which we can use the load values to obtain a load-indexed layered graph for the VRP.

Example 1.4 - A load-indexed layered graph built from a VRP instance Figure 1.6 depicts a load-indexed layered graph constructed from the VRP instance in Example 1.3. In order to avoid a cumbersome drawing, we have removed some incompatible or redundant arcs. Once more, it is not difficult to show that the optimal solution for the VRP instance is given by the tours $\Gamma_1 = (d, u, w, x, d)$ and $\Gamma_2 = (d, z, y, d)$ which correspond, respectively, to the paths $(d_0, u_2, w_4, x_5, d_5)$ and (d_0, z_1, y_6, d_6) in the load-indexed layered graph. □

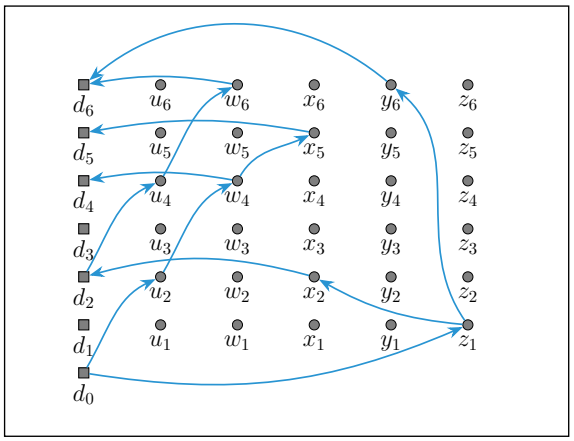


Figure 1.6: A load-indexed layered graph constructed from the VRP instance in Example 1.3. Note that we have removed some incompatible or redundant arcs.

We close this section by summarizing the following observations that we have extracted from the article of Gouveia et al. [109].

- Layered graphs allow to obtain easily integer programming formulations of some graph problems involving complex aspects or relations, but the price to pay for this flexibility is a formulation with a big number of variables.
- Integer programming formulations based on layered graphs usually give tight dual bounds that can be obtained from the associated linear programming relaxations. However the big size of the formulations may turn difficult to solve even those linear programming relaxations and it could be necessary the development of special methods and techniques for handling those models.
- The use of layered graphs can be interesting if the number of achievable resource values is not so big, and also for difficult problems for which are not known alternative models, or acceptable algorithms based on the original decision variables.

- Using a layered graph approach does not seem very promising in problems for which there are ad hoc models with evidence of certain performance, especially if those models can be effectively solved by algorithms based on natural space formulations.

1.2.4 Flows and Multicommodity Flows

In this section, we start by defining two important problems of the network flow theory: the Maximum Flow Problem and the Minimim Cut Problem. Then we introduce the Multicommodity Flow Problem as a generalization of the Maximum Flow Problem.

Let G be a digraph, we denote by $V(G)$ and $A(G)$ the set of vertices and arcs of G, respectively. We also use the notations $\partial_G^-(U) = \{(x, y) \in A(G) : x \notin U, y \in U\}$ and $\partial_G^+(U) = \{(x, y) \in A(G) : x \in U, y \notin U\}$. If U is a singleton $\{x\}$, then we write $\partial_G^-(x)$ and $\partial_G^+(x)$ instead of $\partial_G^-(\{x\})$ and $\partial_G^+(\{x\})$, respectively.

Given a digraph G with capacities $cap : A(G) \to \mathbb{R}_+$, a *flow* is a function $f : A(G) \to \mathbb{R}_+$ with $f(a) \leq cap(a)$ for all $a \in A(G)$. The *excess* of a flow f at $v \in V(G)$ is defined by

$$ex_f(v) = \sum_{a \in \partial_G^-(v)} f(a) - \sum_{a \in \partial_G^+(v)} f(a).$$

We say that f satisfies the *flow conservation rule* at vertex v if $ex_f(v) = 0$. A flow satisfying the flow conservation rule at each vertex is called a *circulation*. An $\hat{s}$-$\hat{p}$-flow is a flow satisfying $ex_f(s) \leq 0$ and $ex_f(v) = 0$ for all $v \in V(G) \setminus \{\hat{s}, \hat{p}\}$. We also define the *value* $\mathrm{val}(f)$ of an $\hat{s}$-$\hat{p}$-flow f by $\mathrm{val}(f) := -ex_f(s)$. One of the most basic problems about $\hat{s}$-$\hat{p}$-flows is the following.

***Maximum Flow Problem*:** Given a digraph G with arc capacities $cap : A(G) \to \mathbb{R}_+$ and two distinguished vertices $\hat{s}$ and $\hat{p}$. Find an $\hat{s}$-$\hat{p}$-flow of maximum value.

This problem can be solved efficiently both in theory [⊛] and practice by using, for example, the Ford-Fulkerson algorithm (see Appendix A.5.4).

Now let us recall that an $\hat{s}$-$\hat{p}$-cut in a digraph G is an arc set $\partial_G^+(X)$ with $\hat{s} \in X$ and $\hat{p} \in V(G) \setminus X$. The *capacity* of an $\hat{s}$-$\hat{p}$-cut is the sum of the capacities of its arcs. One of the most basic problems about $\hat{s}$-$\hat{p}$-cuts is the following.

[⊛] i.e., In polynomial time with respect to the size of the input digraph, see Appendix A.4.

***Minimum Cut Problem*:** Given a digraph G with arc capacities $cap : A(G) \to \mathbb{R}_+$ and two distinguished vertices $\hat{s}$ and $\hat{p}$. Find an $\hat{s}$-$\hat{p}$-cut of minimum capacity.

The following min-max theorem is a central result of network flow theory that links the Maximum Flow Problem and the Minimum Cut Problem.

Theorem 1.2 - Max-Flow-Min-Cut Theorem
In a directed graph G, the maximum value of an $\hat{s}$-$\hat{p}$-flow equals the minimum capacity of an $\hat{s}$-$\hat{p}$-cut. ■

It results that the Minimum Cut Problem can be solved also by using the Ford-Fulkerson algorithm in the following way.

1. Apply the Ford-Fulkerson algorithm for finding a maximum $\hat{s}$-$\hat{p}$-flow f.
2. Build the digraph $G' = \{a \in A(G) : f(a) < cap(a)\}$.
3. The set of vertices x for which there exist an undirected $\hat{s}$-x-path in G', constitutes an $\hat{s}$-$\hat{p}$-cut of minimum capacity in G.

Another important structural result about flows is the following Flow Decomposition Theorem. It was proved independently by Gallai [99], and Ford and Fulkerson [95].

Theorem 1.3 - Flow Decomposition Theorem
Let G be a digraph and let f be an $\hat{s}$-$\hat{p}$-flow in G. Then there exists a family $\mathcal{P}$ of $\hat{s}$-$\hat{p}$-paths and a familiy $\mathcal{C}$ of circuits in G together with weights $w : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}_+$ such that $f(a) = \sum_{P \in \{\mathcal{P} \cup \mathcal{C} : a \in A(P)\}} w(P)$ for all $a \in A(G)$, $\sum_{P \in \mathcal{P}} w(p) = \mathrm{val}(f)$, and $|\mathcal{P}| + |\mathcal{C}| \leq |A(G)|$. Moreover, if f is integral then w can be chosen to be integral. ■

Now we proceed to define the Multicommodity Flow Problem which can be seen as a generalization of the Maximum Flow Problem.

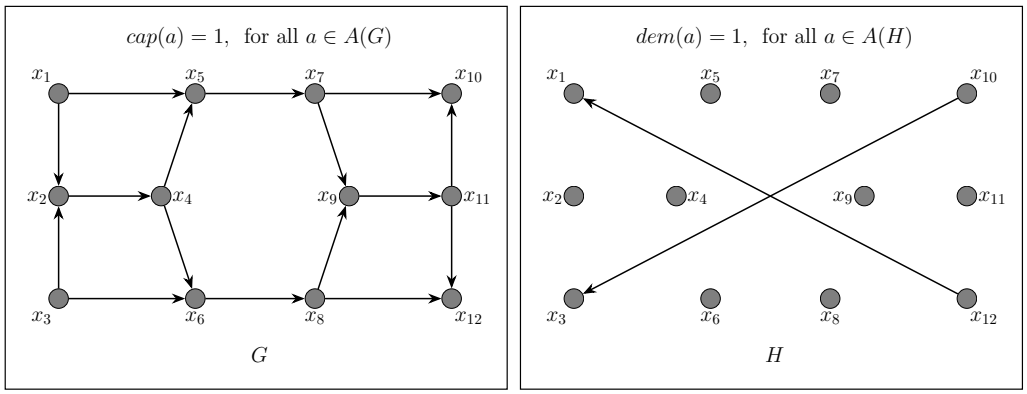
Let us consider a digraph G with arc capacities $cap : A(G) \to \mathbb{R}^+$ and a collection of pairs $(\hat{s}, \hat{p})$ in $V(G) \times V(G)$, the multicommodity flow problem consists in finding an $\hat{s}$-$\hat{p}$-flow for each pair $(\hat{s}, \hat{p})$ in such a way that the total flow through any arc does not exceed the arc capacity. Here, we follow the treatment of Korte and Vygen [142] and we specify the pairs $(\hat{s}, \hat{p})$ by a second digraph H and we add an arc from $(\hat{p}, \hat{s})$ to H when we ask for a flow from $\hat{s}$ to $\hat{p}$ in G. The arcs of G are called *supply arcs* and the arcs of H are *demand arcs* or *commodities*. Endpoints of demand arcs are called *terminals*. Let us state the problem in a formal way.

***Directed Multicommodity Flow Problem*:** Given a pair (G, H) of directed graphs with $V(G) = V(H)$, capacities $cap : A(G) \to \mathbb{R}_+$, and demands $dem : A(H) \to \mathbb{R}_+$. Find a family $(x^f)_{f \in A(H)}$, where x^f is an $\hat{s}$-$\hat{p}$-flow of value $dem(f)$ in G for each $f = (\hat{p}, \hat{s}) \in A(H)$, and

$$\sum_{f \in A(H)} x^f(a) \leq cap(a), \text{ for all } a \in A(G).$$

The Directed Multicommodity Flow Problem has some natural linear programming (LP) formulations of polynomial size and therefore it can be solved in polynomial time by means of linear programming. However, those LP formulations are usually very large and it is preferred to use faster combinatorial algorithms for solving the problem approximatively. Also, the totally unimodular property that is satisfied by the system matrix of an maximum flow LP problem with integral coefficients, is no longer true for Multicommodity Flow Problems as shows the following example.

Example 1.5 - A Directed Multicommodity Flow instance without integer solutions The two digraphs depicted in Figure 1.7 correspond to a Directed Multicommodity Flow Problem instance with unit capacities and unit demands.



(a) (b)

Figure 1.7: An example of a Directed Multicommodity Flow Problem instance. (a) Digraph G and capacities. (b) Digraph H and demands.

The multicommodity flow shown in Figure 1.8 is a feasible solution of the above instance, but it is not an integral solution. In fact, we can prove that there are no integral feasible solutions in the following way.

Digraph G is acyclic so, by Theorem 1.3, we deduce that any positive integral x-y-flow over G can be decomposed into a family of x-y-paths that carry integral positive amounts of flow. Suppose the instance has a feasible integral solution consisting of

an x_1-x_{12}-flow, and an x_3-x_{10}-flow. We apply the Flow Decomposition Theorem to those flows to obtain paths from x_1 to x_{12} and from x_3 to x_{10} that carry positive integral flow. Note that there are four distinct paths from x_1 to x_{12}, four distinct paths from x_3 to x_{10}, and that any x_1-x_{12}-path intersects to any x_3-x_{10}-path in at least one arc. Therefore, if we select an x_1-x_{12}-path and an x_3-x_{10}-path from the above flow decompositions, the amount of flow passing through any common arc of those paths must be at least two, but this violates the capacity constraints. Such a contradiction proves that there cannot be integral feasible solutions of the above instance. □

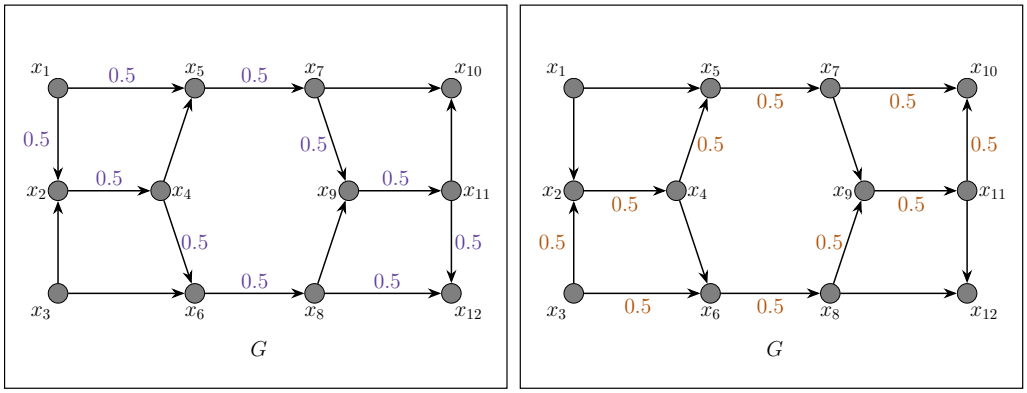


Figure 1.8: A feasible solution of the Directed Multicommodity Flow Problem instance illustrated in Figure 1.7. (a) x_1-x_{12}-flow. (b) x_3-x_{10}-flow.

Part II

The Item Relocation Problem with Transfers and Time Horizon

Chapter 2

The Projected Item Relocation Problem

In this chapter we introduce the Item Relocation Problem (IRP) which is the first problem that we treat in this work. The problem is motivated by shared mobility systems where a collection of items (e.g., bicycles) is distributed on a transit network, and a fleet of vehicles must relocate those items to meet an expected user demand. The IRP can be seen as a one-commodity many-to-many Capacitated Pickup-and-Delivery Problem with unpaired pickups and deliveries, transfers, and time horizon.

We start by defining the problem in a formal way and we propose the TEN IRP model which is a 2-commodity flow model over a Time-Expanded network for handling the IRP. Because Time-Expanded network models usually involve a big number of integer variables, they are very difficult to solve directly with a MILP solver. For that reason, we propose a "Project-and-Lift" approach for handling that TEN IRP model.

First, we "project" the TEN IRP model on the original transit network to obtain the Projected IRP model (PIRP) which is a simpler 2-commodity flow model that involves only a linear number of integer variables, but which drops the temporal dimension of the original IRP. Most of this chapter will be focused on that PIRP model, and the way it can be improved by the introduction of specific constraints for recovering part of the temporal dimension of the problem.

Next, we introduce the "Lift" problems, which consist in the construction of solutions for the TEN IRP model starting from solutions of the PIRP model. We present two Lift problems which can be distinguished by their degree of compatibility with PIRP solutions.

2.1 Introduction

Emerging mobility systems, based on vehicle sharing, aim at finding a flexible compromise between individual mobility and rigid public transportation systems (see e.g., [101, 182, 195] for surveys). Such systems include collective taxis, transport on demand, carpooling, shared riding, etc., and share some key features: they rely on emerging mobility technologies (e.g., electric vehicles, autonomous vehicles), require a responsive day-to-day operational management through intensive use of internet platforms, and aim at answering environmental concerns and urban congestion, while keeping part of the flexibility of individual transportation.

Managing such emerging mobility systems requires decisions at a strategic level (for pricing, infrastructure dimensioning, demand/cost analysis, integration into multimodality, see e.g., [17, 101, 216]), at an operational level (for real-time handling of the demands, relocation of free vehicles, synchronization, and unexpected event management, see e.g., [101, 168]), and also at a tactical level (for in advance management of recurrent demands or maintenance scheduling, see e.g., [26, 45, 62, 190]).

The resulting problems are difficult and must be usually addressed in a dynamic way, taking uncertainty into account. In order to overcome these difficulties, the problems are often studied in a static paradigm using aggregated representations of the circulation of vehicles and passengers to make decisions about the dimensions of the system and to precompute routes and schedules for its operational management. A popular approach is the use of multicommodity flow models, where the different commodities correspond to the different kinds of objects whose circulation is supported by the network (see e.g., see Section 1.2.4 or [4, 39, 40, 41, 46, 144]).

Hereby, it is crucial to take time constraints into account that are imposed by the synchronization of vehicle routes and user demands. In order to deal with this specific temporal dimension while taking profit from the powerful network flow machinery, mobility models are often cast in Time-Expanded networks (TENs), containing one copy of the original transit network for each considered time unit (see e.g., Section 1.2.3 or [14, 41, 40, 51]). This powerful conceptual tool is well fitted to the modeling of systems and decision problems involving the circulation of resources (e.g., vehicles, objects, passengers) over time. The disadvantage is that such networks grow with the size of the time space so that solving multicommodity flow problems in TENs typically requires very long computation times.

One possible approach to overcome this difficulty consists in restricting the TEN to an active part of the network of limited size (see e.g., [14, 144]) which typically results in heuristics without optimality guarantee. Another approach consists in projecting the TEN model to the original network or to some auxiliary simpler

network (see [45, 81]), and so making the temporal dimension become implicit: the algorithmic process consists in first solving the projected model, and next turning (i.e., lifting) the resulting solution into a solution of the original TEN model.

In this work, we adopt the latter approach. For the sake of simplicity, we work on a generic 2-commodity flow model, related to the Item Relocation Problem (see e.g., [26, 45, 81, 182]), which arises when vehicles are required to periodically exchange items between stations in order to rebalance the access to those items for potential users. According to such an interpretation, one flow is related to vehicles and the other to the (identical) items which are transported by those vehicles. In most cases, this relocation process has to be performed within a given time horizon, and vehicles are allowed to meet in order to exchange part of their load if necessary. The 2-commodity flow model that we present in this chapter is generic in the sense that it embraces the whole spectrum of possible quality criteria: the number of involved vehicles, vehicle operational costs, and item riding time.

Relation with the Existent Literature

Around 1960 Ford and Fulkerson [95] showed how problems involving both routing and scheduling could be cast into the network flow framework. This issue about how to take into account the time-dependence of a network motivated in the 1970/1980 years the emergence of the notion of "dynamic networks" (see [14]) and gave rise to the generalization of network flow models involving rational or integral flow values to models where flow functions become trajectories subject to constraints. Sometimes they are called "flow over time" functions because they correspond to transit rates for items which circulate within the network and whose values depend on the time.

In the years 1990/2000, several important contributions by Fleischer and Skutella [91, 92], Hall [112], and Powell [174], were carried on about both applications to evacuation planning and algorithm design. Those authors addressed the complexity issue and explored the way how standard flow algorithms could be adapted to the flow over time framework. They dealt with the multicommodity flow issue (see [159, 5, 180]) while adapting algorithms designed for standard static networks, and could state, in the continuous case, some "fully polynomial-time approximation schemes" (FPTAS) (see [91]). At this time, they also brought insights about the link between the TEN framework and the network flow over time models.

More recently, some authors tried to combine the TEN framework with the improvement of the mixed integer linear programming (MILP) libraries in order to directly address some transportation problems through the application of MILP libraries to multicommodity flow models set on Time-Expanded networks (see [174,

91, 180, 92, 182, 190]). In [96] and [144], authors adapted column generation and branch-and-price techniques in order to restrict the search process to an active part of the TEN and so to better control computational times.

As for the applications of the TEN framework, they were initially related to evacuation problems or energy transportation problems (see [5, 174]), which typically involve only one flow vector whose value is going to evolve throughout the time, on a network whose arc capacities or availability are going to evolve themselves throughout the time. But attempts to rely on the TEN framework, at least to formalize the problems and possibly to solve them in an approximate way were also done in order to optimize the rebalancing processes involved in the management of vehicle sharing systems (see [14, 159, 174, 17, 96]) and deal with Pickup-and-Delivery Problems (see [13]). In those last cases, the authors had to deal with multicommodity flow models, which imposed a significantly higher complexity (see [95, 159, 5]).

Outline

This chapter is organized as follows: we first introduce in Sections 2.2) and 2.3 the reference Item Relocation Problem (IRP) and its formulation through the TEN framework. In Section 2.4, we describe the Projected IRP (PIRP) model and the way how this model may be enhanced through the use of "Extended-Subtour" constraints which link the time horizon and the number of vehicles circulating through each subset of non-depot vertices. We propose two versions of those constraints, which we prove to be both separable in polynomial time.

Next, in Section 2.5 we propose a brief discussion about the Lift problems, we present some introductory examples, and we describe the Project-and-Lift Decomposition Scheme for handling the TEN IRP model. In Section 2.5.1 we present two Lift problems which can be distinguished by their degree of compatibility with PIRP solutions. In Section 2.5.2 we examine the feasibility of one of the Lift problems and as a result we derive the "Feasible-Path" constraints for the PIRP model (Section 2.5.3). Those constraints are necessary for the feasibility of the Lift problems and involve a path decomposition property that must be verified by the item flow. The separation of those constraints its known to be an $\mathcal{NP}$-hard problem, but we show a way to deal with them through column generation.

Finally, Section 2.6 is devoted to the description of the resulting branch-and-cut algorithms and to numerical experiments.

2.2 The Item Relocation Problem

We have a *transit network* represented over a simple digraph $G = (X, A)$ with a set of vertices X and a set of arcs A. There exists in X a distinguished *depot vertex* d where vehicles are located at the beginning and at the end of the relocation process. There are two weight functions $cost : A \to \mathbb{R}_+$ and $time : A \to \mathbb{R}_+$ associated with the digraph G. For all $a = (x, y) \in A$, the values $cost(a)$ and $time(a)$ respectively encode the cost and the time required by a vehicle in order to move from vertex x to vertex y. We extend those functions to the functions $cost : V(G) \times V(G) \to \mathbb{R}_+$ and $time : V(G) \times V(G) \to \mathbb{R}_+$ defined by $cost_G(x, y) = \text{dist}_{(G,cost)}(x, y)$ and $time_G(x, y) = \text{dist}_{(G,time)}(x, y)$, respectively.

Items are situated at the vertices of the digraph G and rebalancing their distribution may be required to guarantee the operability of the mobility system. When we decide to launch the relocation process, we are provided with an integral vector $\mathbf{b}$ of *balance coefficients* b_x, $x \in X$ which satisfies $\sum_{x \in X} b_x = 0$. A value $b_x > 0$ indicates that x is an *excess vertex* and vehicles must remove b_x items from x, a value $b_x < 0$ indicates that x is a *deficit vertex* and vehicles must bring $-b_x$ items to x, and a value $b_x = 0$ indicates that x is *neutral* and it is not required to remove or bring items to x. We denote by X^+ the set of all the excess vertices, and analogously we denote by X^- the set of all the deficit vertices.

For the relocation process, a fleet of identical vehicles is available. Each vehicle has a *capacity* κ, specifying the maximal number of items which may be transported by a vehicle at the same time. The relocation process must take place within a given *time horizon* $[0, \Omega]$.

The *Item Relocation Problem* (IRP) consists in organizing the transfer by vehicles of items from excess vertices to deficit vertices, while meeting time horizon and vehicle capacity requirements. Furthermore, preemption is allowed, which means that vehicles may exchange items. We next set the problem in a more formal way.

IRP Inputs:

A digraph $G = (X, A)$ with a specific depot vertex s, a weight function $time : A \to \mathbb{R}_+$, and a weight function $cost : A \to \mathbb{R}_+$. A vector $\mathbf{b}$ of balance coefficients indexed over X, and a time horizon $[0, \Omega]$.

IRP Outputs:

A *vehicle route* is a circuit which starts and ends in the depot vertex d. Because no more than one arc $a = (x, y)$ connects a vertex x to another vertex y, the route of a vehicle q can be represented as a sequence $\Gamma^q = (x_0^q = d, x_1^q, \ldots, x_{\nu(q)}^q =$

d) of $\nu(q)+1$ vertices, with both first and last elements equal to d. This route must be computed together with two *time sequences* $t^q = (t^q_0, \ldots, t^q_{\nu(q)})$ and $\bar{t}^q = (\bar{t}^q_0, \ldots, \bar{t}^q_{\nu(q)})$, and with a *load sequence* $\ell^q = (\ell^q_0, \ldots, \ell^q_{\nu(q)-1})$: the value t^q_i means the time when the vehicle q arrives at vertex x^q_i, the value $\bar{t}^q_i$ means the time when the vehicle q leaves vertex x^q_i, and the value ℓ^q_i is the load of vehicle q when it leaves vertex x^q_i. If vehicle q arriving at vertex x according to x^q_i transfers a load ℓ to vehicle q' leaving vertex x according to $x^{q'}_j$ then we say that vehicles q and q' perform a *transfer transaction* (x, q, i, q', j, ℓ), and we call q and q' the *emitting vehicle* and the *receiving vehicle* related to (x, q, i, q', j, ℓ), respectively. A solution of the IRP is defined by two main objects:

- a *route* collection $\Gamma = \{\Gamma^q : q = 1, \ldots, Q\}$, every route Γ^q being given together with the time sequences t^q, $\bar{t}^q$ and the load sequence ℓ^q. This collection Γ provides us with the route followed by the vehicles, together with their schedules and their loads;
- a collection $\Theta \subseteq \{(x, q, i, q', j, L) : x^q_i \in \Gamma^q, x^{q'}_j \in \Gamma^p\}$ of transfer transactions which expresses the way the vehicles interact during the process.

IRP Constraints:

- Loads $\ell^q_i, q = 1, \ldots, Q,\ i = 1, \ldots, \nu(q)$, must never exceed the capacity κ, and must fit with the relocation requirements as expressed by the vector $\mathbf{b}$ of balance coefficients. For any vertex x, the balance between the loads which arrive into x and those which leave x, including those involved into the transfer transactions, must be equal to the balance coefficient b_x.
- Time values $t^q_i, \bar{t}^q_i, q = 1, \ldots, Q,\ i = 1, \ldots, \nu(q)$, must belong to the interval $[0, \Omega]$, be consistent with the weight function $time : A \to \mathbb{R}_+$, and meet the *Weak Synchronization Constraints*: if a transfer transaction (x, q, i, q', j, ℓ) occurs, then the receiving vehicle q' cannot leave $x^{q'}_j$ before the emitting vehicle q could reach x^q_i.

IRP Objective Function:

- The considered cost function involves the following three components: the *number of vehicles* $c_1 = Q$, the *vehicle ride costs* $c_2 = \sum_{q=1}^{Q} \sum_{i=0}^{\nu(q)-1} cost(x^q_i, x^q_{i+1})$ which tends to express the costs induced by the routes for the fleet manager, and the *item ride time* c_3 which means the time during which items are not available for the users and aims at expressing the notion of *service quality*.
- In order to avoid a multiobjective formulation, we consider the minimization of a hybrid cost $\alpha \cdot c_1 + \beta \cdot c_2 + \gamma \cdot c_3$, where α, β, γ are positive scaling coefficients.

Example 2.1 - An example of an IRP instance
Consider the digraph $G = (X, A)$ depicted in Figure 2.1. It shows an example of an IRP instance over G and a solution involving two vehicles, say $q = 1$ and $q' = 2$, which transit over two paths Γ^1 and Γ^2, respectively.

The vehicle q moves from the depot vertex d to vertex u in one time unit. At time one, it balances u by picking up five items. Then, it moves from u to x in one time unit. At time two, q drops two items at vertex x and then it moves from x to y in one time unit. At time three, q satisfies the demand of vertex y by dropping three items. Finally, q goes back from y to d in one time unit.

On the other hand, the vehicle q' moves from the depot vertex d to vertex w in two time units. At time two, it balances w by picking up five items. Then, it moves from w to x in one time unit. At time three, q' picks up from x the two items dropped by q. Then it goes from x to z in one time unit. At time four, q' satisfies the demand of vertex z by dropping seven items. Finally, q' goes back from z to d in two time units.

We have that $\Gamma^1 = (d, u, x, y, d)$, $t^1 = (0, 1, 2, 3, 4)$, $\bar{t}^1 = (0, 1, 2, 3, 4)$, and $\ell^1 = (0, 5, 3, 0)$. Similarly, $\Gamma^2 = (d, w, x, z, d)$, $t^2 = (0, 2, 3, 4, 6)$, $\bar{t}^2 = (0, 2, 3, 4, 6)$, and $\ell^2 = (0, 5, 7, 0)$. There is a transfer transaction $(x, 1, 2, 2, 2, 2)$ involved at vertex x because two items are transferred from the vehicle in Γ^1 to the vehicle in Γ^2.

Finally, we check that $c_1 = 2$, $c_2 = \sum_{a \in A(\Gamma^1)} cost(a) + \sum_{a \in A(\Gamma^2)} cost(a) = 4 + 6 = 10$, and $c_3 = 5 \cdot time((u, x)) + 3 \cdot time((x, y)) + 5 \cdot time((w, x)) + 7 \cdot time((x, z)) = 20$. □

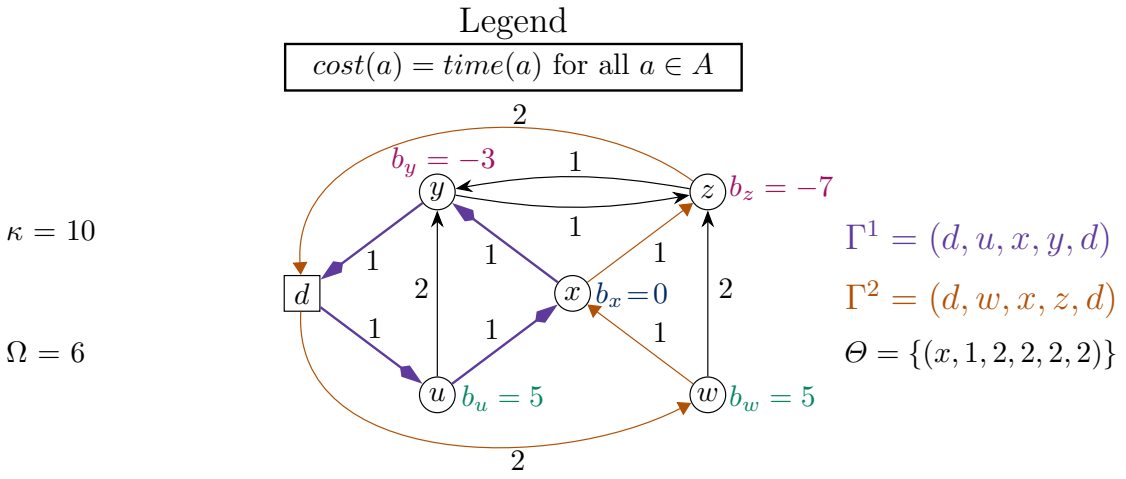


Figure 2.1: An example of an IRP instance over a digraph $G = (X, A)$ and a solution involving two vehicles and one transfer transaction.

2.3 A TEN 2-Commodity Flow Formulation

In order to cast the IRP into the TEN framework (see [14, 41, 51, 144]), we first derive from the digraph $G = (X, A)$ its time expansion $G^\Omega = (X^\Omega, A^\Omega)$ according to Ω. The vertex set X^Ω is the set of all pairs $x_t = (x, t)$, $x \in X$, $t \in \{0, 1, \ldots, \Omega\}$, augmented with two distinguished vertices, a *source* $\hat{s}$ and a *sink* $\hat{p}$. We associate with the digraph $G^\Omega = (X^\Omega, A^\Omega)$ two weight functions $time_{G^\Omega} : A^\Omega \to \mathbb{R}_+$ and $cost_{G^\Omega} : A^\Omega \to \mathbb{R}_+$ which are defined according to the following types of arcs.

- *Input-arcs* $a = \big(\hat{s}, (x, 0)\big)$, with $x \in X$, $time_{G^\Omega}(a) = 0$, and $cost_{G^\Omega}(a) = 0$.
- *Output-arcs* $a = \big((x, \Omega), \hat{p}\big)$, with $x \in X$, $time_{G^\Omega}(a) = 0$, and $cost_{G^\Omega}(a) = 0$.
- *Waiting-arcs* $a = \big((x, t), (x, t+1)\big)$, with $x \in X$, $t \in \{0, \ldots, \Omega-1\}$, $time_{G^\Omega}(a) = 0$, and $cost_{G^\Omega}(a) = 0$.
- *Active-arcs* $a = \big((x, t), (y, t + time(x, y))\big)$, with $(x, y) \in A$, $t \in \{0, \ldots, \Omega - time(x, y)\}$, $time_{G^\Omega}(a) = \gamma \cdot time(x, y)$, and $cost_{G^\Omega}(a) = \beta \cdot cost(x, y)$.
- *Backward-arc* $a = (\hat{p}, \hat{s})$ with $time_{G^\Omega}(a) = 0$ and $cost_{G^\Omega}(a) = \alpha$.

Now, we formalize the IRP as a 2-commodity flow model on $G^\Omega = (X^\Omega, A^\Omega)$.

***TEN IRP*:** Find functions $H : A^\Omega \to \mathbb{Z}_+$ and $h : A^\Omega \to \mathbb{Z}_+$ (for vehicles and items, respectively) such that

- H and h satisfy flow conservation at any vertex of X^Ω. (E1)
- For any active-arc $a = \big((x, t), (y, t + time(x, y))\big)$: $h(a) \le \kappa \cdot H(a)$. (E2)
- For any input-arc $a = \big(\hat{s}, (x, 0)\big)$, with $x \neq d$: (E3)
  $H(a) = 0$ and $h(a) = \max(b_x, 0)$.
- For any output-arc $a = \big((y, \Omega), \hat{p}\big)$, with $y \neq d$: (E4)
  $H(a) = 0$ and $h(a) = \max(-b_y, 0)$.
- The global cost

$$Cost(H, h) = \sum_{a \in A^\Omega} \big(H(a) \cdot cost_{G^\Omega}(a) + h(a) \cdot time_{G^\Omega}(a)\big)$$

  is minimized.

Explanation. The flow conservation constraints (E1) express the circulation of vehicle and items inside the digraph G, condition (E2) ensures that any item moving between two vertices x and y must be contained into some vehicle. Conditions (E3) and (E4) provide us with initial and final constraints: vehicles must start and end their route in d, while the flow h means that for any excess vertex x, b_x items must leave x, and that for any deficit vertex y, b_y items must arrive into y.

Example 2.2 - An example of TEN IRP instance

Figure 2.2 shows the construction of the Time-Expanded network $G^\Omega = (X^\Omega, A^\Omega)$ associated to the digraph $G = (X, A)$ of Figure 2.1. It also illustrates how to turn the vehicle routes and schedules of Example 2.1 into a 2-commodity flow (H, h). □

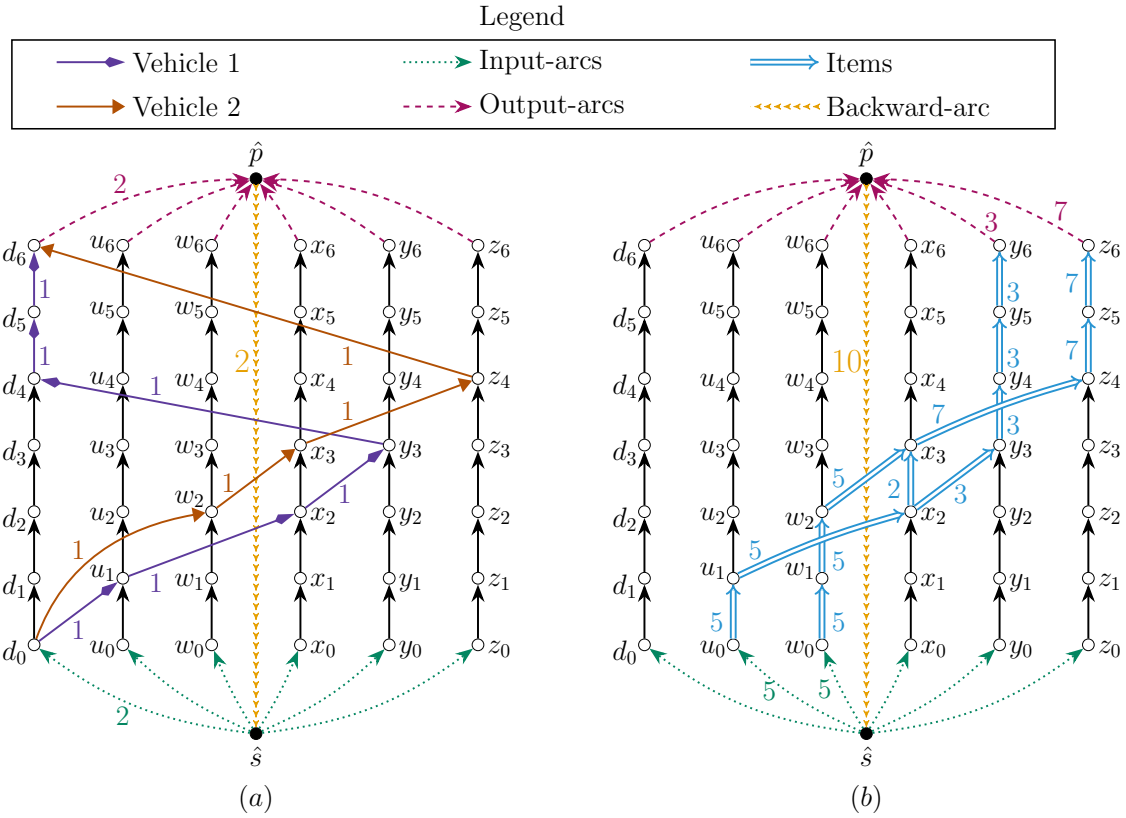


Figure 2.2: The routes and schedules from Example 2.1 viewed as a 2-commodity flow on the Time-Expanded network $G^\Omega = (X^\Omega, A^\Omega)$. (a) Vehicle flow H in G^Ω. (b) Item flow h in G^Ω.

Theorem 2.1 - TEN IRP ⇒ IRP

Solving the TEN IRP also solves the IRP.

Proof. If (H, h) is a feasible solution of the above TEN IRP formulation, then we may decompose H into a sum of $c_1 = Q = H\big((\hat{p}, \hat{s})\big)$ flows $(0, 1)$-valued, $H_1, \ldots, H_Q$, which are the supports of circuits containing the arc $(\hat{p}, \hat{s})$ and so connect vertex

$(s,0)$ with the vertex (s,Ω). Those $(0,1)$-valued flows may be turned into a vehicle route collection $\Gamma = \{\Gamma^1,\ldots,\Gamma^Q\}$, together with a time sequence $t^q = (t^q_0,\ldots,t^q_{\nu(q)})$, which fit with IRP requirements. Finally, we distribute the values $h(a)$, $a \in A^\Omega$ among the routes $\Gamma^q, q = 1,\ldots,Q$, in such a way that we get a load sequence $\ell^q = (\ell^q_0,\ldots,\ell^q_{\nu(q)} - 1)$ which meets IRP requirements.

Conversely, any scheduled route collection Γ of IRP can be turned into a 2-commodity flow (H,h) which meets (E1)-(E4), with the same global cost value. ■

2.3.1 A Characterization of the IRP Feasibility

Depending on the value Ω, the IRP may or may not admit feasible solutions. In order to characterize the existence of feasible solutions, let us recall that we have denoted by X^+ the set of the vertices x of the digraph $G = (X,A)$ such that $b_x > 0$ and by X^- the set of the vertices y such that $b_y < 0$. We define a *feasible-pair* (x,y) for the time horizon Ω as any pair (x,y), $x \in X^+$, $y \in X^-$ such that $time(d,x) + time(x,y) + time(y,d) \leq \Omega$. We denote by Φ^{FP} the set of all feasible-pairs and for any subset $U \subseteq X^+$ we denote by $\Phi^{FP}(U)$ the image of U through the feasible-pairs, that means the set of vertices $y \in X^-$ such that there exists a feasible-pair (x,y) with $x \in U$. The following theorem characterizes the IRP feasibility.

Theorem 2.2 - Characterization of the IRP Feasibility
IRP *admits a feasible solution (H,h) if and only if, for any subset U of the vertex set X^+, the following inequality holds:* $\sum_{x\in U} b_x \leq \sum_{y\in\Phi^{FP}(U)} |b_y|$. (F1)

Proof. The only if part is obvious, since the flow h must express the transportation of any item in U to some vertex of $\Phi^{FP}(U)$ in no more than Ω time units. Conversely, let us suppose that (F1) holds. Then we deduce from matching theory (Hall's marriage theorem [113]) that the following linear program PMATCH admits a feasible solution.

LP PMATCH: Compute an integral nonnegative vector $\mathbf{z} = \left(z_{(x,y)}, (x,y) \in \Phi^{FP}\right)$ such that

- for any $x \in X^+$, $\sum_{(x,y)\in\Phi^{FP}} z_{(x,y)} = b_x$,
- for any $x \in X^-$, $\sum_{(x,y)\in\Phi^{FP}} z_{(x,y)} = -b_y$.

For every (x,y) such that $z_{(x,y)} \neq 0$ we set: $t_x = time(d,x)$ and $t_y = time(y,d)$. Then we may derive in a natural way a flow $h^{(x,y)}$ which transports $z_{(x,y)}$ items from (x,t_x) to (y,t_y) in the TEN G^Ω while following a shortest path (with respect to the weight function $time$) in G from x to y, and a nonnegative integral flow $H^{(x,y)}$ which involves $\lceil z_{(x,y)}/\kappa \rceil$ vehicles along a shortest path in G^Ω from $\hat{s}$ to $\hat{p}$ (with respect to the weight function $time_{G^\Omega}$), that visits (x,t_x) before (y,t_y), and whose support contains the support of $h^{(x,y)}$. We only need to set $H = \sum_{(x,y)\in A} H^{(x,y)}$ and $h = \sum_{(x,y)\in A} h^{(x,y)}$ in order to conclude. ■

2.4 The Projected IRP Model

This section is devoted to some of the major results of the present chapter. We are going to deal with the studied IRP while projecting the TEN IRP model on the original digraph G.

Consider an IRP instance on a digraph $G = (X,A)$ with a time horizon $[0,\Omega]$. Given a flow H on the Time-Expanded network $G^\Omega = (X^\Omega, A^\Omega)$ we define the *projection* of flow H on G as the flow $F : A \to \mathbb{Z}_+$ defined for all $a = (x,y) \in A$ by

$$F(a) = \sum_{t=0}^{\Omega - time_G(a)} H\big((x,t),(y,t+time_G(a))\big).$$

Similarly, given a 2-commodity flow (H,h) on the Time-Expanded network $G^\Omega = (X^\Omega, A^\Omega)$ we define the *projection* of the 2-commodity flow (H,h) on G, as the 2-commodity flow $(F,\ f)$ such that F and f are the projections on G of the flows H and h, respectively.

Additional Notation. Given a digraph $G = (X,A)$ and sets $U,V \subseteq X$, we use the notation $\partial_G^-(U) = \{(x,y) \in A : x \notin U,\ y \in U\}$, $\partial_G^+(U) = \{(x,y) \in A : x \in U,\ y \notin U\}$, $\partial_G(U) = \partial^-(U) \cup \partial^+(U)$, and $A(U,V) = \{(x,y) \in A : x \in U,\ y \in V\}$. For a singleton $\{x\}$ we will write $\partial_G^-(x)$, $\partial_G^+(x)$, and $\partial_G(x)$, instead of $\partial_G^-(\{x\})$, $\partial_G^+(\{x\})$, and $\partial_G(\{x\})$, respectively. Also, for any $U \subseteq X \setminus \{d\}$ we define a value $d_U^- = dist_{(G,time)}(d,U) = \inf_{\{x\in X:\exists (x,y)\in\partial_G^-(U)\}} time(d,x)$ and a value $d_U^+ = dist_{(G,time)}(U,d) = \inf_{\{y\in X:\exists (x,y)\in\partial_G^+(U)\}} time(y,d)$.

Projecting the TEN IRP on the digraph G means to deal with the projections F and f, of flows H and h on the arcs of G. If we restrict ourselves to G (i.e., we do not take source $\hat{s}$ and sink $\hat{p}$ into account), then we obtain the following constraints.

- F satisfies flow conservation at any vertex of X. (E5.1)
- For any vertex x of G: $\sum_{a\in\partial^+_G(x)} f(a) - \sum_{a\in\partial^-_G(x)} f(a) = b_x$; (E5.2)
- For any arc a of G: $f(a) \leq \kappa \cdot F(a)$; (E6)
- Vehicle ride cost $c_2 = \beta \cdot (\sum_{a\in A} cost(a) \cdot F(a))$. (E7.1)
- Item ride time $c_3 = \gamma \cdot (\sum_{a\in A} time(a) \cdot f(a))$. (E7.2)

2.4.1 Projected Cost and Extended-Subtour Constraints

Constraints (E5.1), (E5.2), and (E6) are not enough to characterize F and f in a satisfactory way, since they do not forbid subtours (see Figure 2.3), likely to induce a significant distortion between vehicle ride cost c_2 and item ride time c_3 related to H and h, and the values $\beta \cdot (\sum_{a\in A} cost(a) \cdot F(a))$ and $\gamma \cdot (\sum_{a\in A} time(a) \cdot f(a))$.

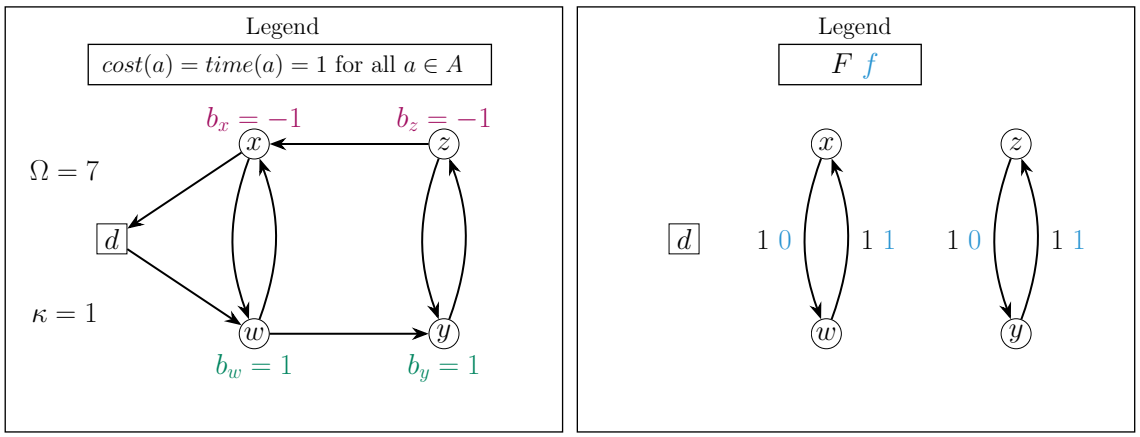


Figure 2.3: A 2-commodity flow (F, f) with subtours. (a) An example of an IRP instance over a digraph $G = (X, A)$. (b) A 2-commodity flow (F, f) that satisfies (E5.1)-(E6) but contains two subtours which are not reachable from the depot.

Besides, (E5.1)-(E7.2) do not provide us with a well-fitted estimation of the vehicle number $c_1 = Q = H\big((\hat{t}, \hat{s})\big)$. However, we can check that the following statements hold.

Lemma 2.1. *Let (H,h) be a feasible solution of TEN IRP and let (F,f) be its projection on the network $G=(X,A)$, then $\frac{(\sum_{a\in A} time(a)\cdot F(a))}{\Omega}$ is a lower bound for the vehicle number.*

Proof. The quantity $\sum_{a\in A} time(a)\cdot F(a)$ provides us with the global time vehicles spend running inside G, waiting times being excluded. Since the whole process must be performed in no more than Ω time units, we see that we need at least $\frac{(\sum_{a\in A} time(a)\cdot F(a))}{\Omega}$ vehicles in order to achieve it. ■

As a consequence, we should search for a 2-commodity flow (F,f) that minimizes the *projected cost*:

$$PCost(F,f)=\alpha\cdot\frac{\left(\sum_{a\in A} time(a)\cdot F(a)\right)}{\Omega}+\beta\cdot\left(\sum_{a\in A} cost(a)\cdot F(a)\right)+\gamma\cdot\left(\sum_{a\in A} time(a)\cdot f(a)\right).$$

Lemma 2.2. *For all $U\subseteq X\setminus\{d\}$, the following Weak-Extended-Subtour constraint holds:*

$$\Omega\cdot\left(\sum_{a\in\partial_G^-(U)} F(a)\right)\geq\sum_{a\in\partial_G(U)\cup A(U,U)} time(a)\cdot F(a). \tag{E8.1}$$

Proof. Let us denote by Q the number of vehicles involved in a TEN IRP solution (H,h). Any vehicle q may enter into U, one or several times, get out of U, and move inside U. Hence the global time that vehicles spend inside U, entering to U, and going out from U is equal to $\sum_{a\in\partial_G(U)\cup A(U,U)} time(a)\cdot F(a)$. For each vehicle q, this time cannot exceed Ω. Hence, we deduce that $\Omega\cdot Q\geq\sum_{a\in\partial_G(U)\cup A(U,U)} time(a)\cdot F(a)$. Since $\sum_{a\in\partial_N^-(U)} F(a)\geq Q$, we conclude. ■

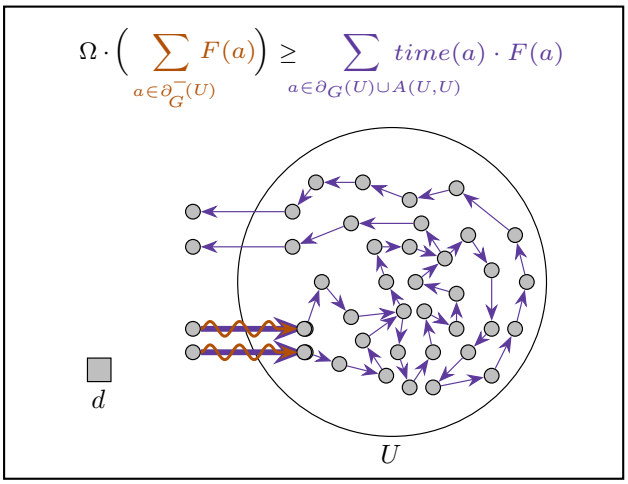


Figure 2.4: Interpretation of the Weak-Extended-Subtour constraints. The number of vehicles entering $U\subseteq X\setminus\{d\}$ is upper bounded by $\sum_{a\in\partial_G^-(U)} F(a)$ and the total amount of time available for those vehicles must be enough to visit, within the time horizon $[0,\Omega]$, each arc $a\in\partial_G(U)\cup A(U,U)$, a total of $F(a)$ times (i.e., $\sum_{a\in\partial_G(U)\cup A(U,U)} time(a)\cdot F(a)$).

An interpretation of the Weak-Extended Subtour constraints can be found in Figure 2.4. It results that those constraints can be strengthened as follows.

Lemma 2.3. *For all $U \subseteq X \setminus \{d\}$, the following Strong-Extended-Subtour constraint holds:*

$$(\Omega - d_U^- - d_U^+) \cdot \left(\sum_{a \in \partial_N^-(U)} F(a) \right) \geq \sum_{a \in \partial_N(U) \cup A(U,U)} T_a \cdot F(a). \qquad \text{(E8.2)}$$

Proof. We adapt the proof of Lemma 2.2, while noticing that for each of the Q vehicles involved into U, the time it spends while entering into U, getting out of U and moving inside U, cannot exceed $\Omega - d_U^- - d_U^+$, since it moves first from d until some vertex x such that $(x,u) \in \partial_G^-(U)$ and next goes back the same way from some vertex y, such that $(u',y) \in \partial_G^+(U)$ into d. ∎

Figure 2.5 shows an interpretation of the Strong-Extended-Subtour constraints.

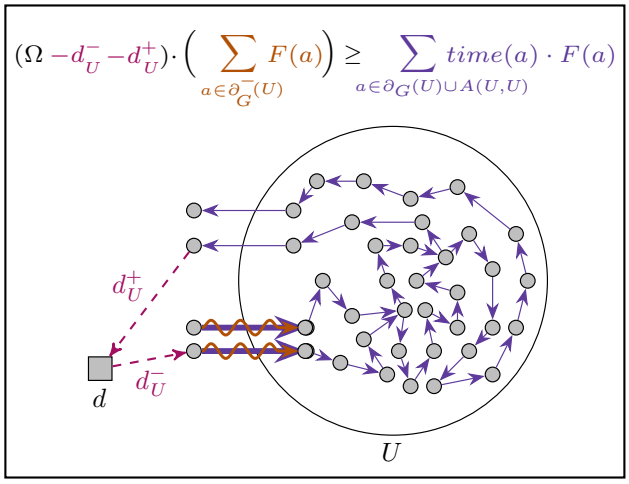


Figure 2.5: Interpretation of the Strong-Extended-Subtour constraints. The amount of time that a vehicle needs to reach the tail of an arc in $\partial^-(U)$ from the depot d, is lower bounded by d_U^-; symmetrically the amount of time that a vehicle needs to reach the depot d from the head of arc in $\partial^+(U)$, is lower bounded by d_U^+. The number of vehicles entering $U \subseteq X \setminus \{d\}$ is upper bounded by $\sum_{a \in \partial_G^-(U)} F(a)$ and the total amount of time available for those vehicles must be enough to visit, within the time horizon $[d_U^-, \Omega - d_U^+]$, each arc $a \in \partial_G(U) \cup A(U,U)$, a total of $F(a)$ times (i.e., $\sum_{a \in \partial_G(U) \cup A(U,U)} time(a) \cdot F(a)$).

Remark 1. Though Lemma 2.3 implies Lemma 2.2, we distinguish both, because they are going to induce very different separation algorithms, and because in practice, only constraints (E8.1) are going to be efficient.

We deduce that we should search for (F, f) as a solution of the following projected model.

Projected Item Relocation Problem (PIRP): Given an IRP instance with a capacity k, a time horizon $[0, \Omega]$, and a digraph $G = (X, A)$ with depot vertex d. Find two functions $F : A \to \mathbb{Z}_+$ and $f : A \to \mathbb{Z}_+$ such that:

- F satisfies flow conservation at any vertex of X; (E5.1)
- for any vertex $x \in X$, $\sum_{a \in \partial_N^-(x)} h(a) - \sum_{a \in \partial_N^+(x)} h(a) = b_x$; (E5.2)
- for any arc $a \in A$, $f(a) \leq \kappa \cdot F(a)$; (E6)
- for any $U \subseteq X \setminus \{d\}$,
  $(\Omega - d_U^+ - d_U^-) \cdot \Big(\sum_{a \in \partial_N^-(U)} F(a) \Big) \geq \sum_{a \in \partial_N(U) \cup A(U,U)} time(a) \cdot F(a)$; (E8.2)
- Minimize the projected cost $PCost(F, f) =$ (E9)
  $\alpha \cdot \frac{\left(\sum_{a \in A} time(a) \cdot F(a) \right)}{\Omega} + \beta \cdot \Big(\sum_{a \in A} cost(a) \cdot F(a) \Big) + \gamma \cdot \Big(\sum_{a \in A} time(a) \cdot f(a) \Big)$.

2.4.2 Separating the Extended-Subtour Constraints

We have the two following results.

Theorem 2.3 - Complexity of the Weak-Extended-Subtour Constraints Separation *The Weak-Extended-Subtour constraints* (E8.1) *can be separated in polynomial time, through a classical Min-Cut algorithm.*

Proof. Given some flow F on a digraph $G = (X, A)$ and a time horizon $[0, \Omega]$. Separating (E8.1) means searching for $U \subseteq X \setminus \{d\}$ such that $\Omega \cdot \Big(\sum_{a \in \partial_G^-(U)} F(a) \Big) - \sum_{a \in \partial_G(U) \cup A(U,U)} time(a) \cdot F(a) < 0$. If we add a term $\sum_{a \in A} time(a) \cdot F(a)$ on both sides, and divide by Ω, we see that it also means searching for $U' = X \setminus U$, such that $\sum_{a \in \partial_G^+(U')} F(a) + \frac{(\sum_{a \in A(V,V)} time(a) \cdot F(a))}{\Omega} < \frac{(\sum_{a \in A} time(a) \cdot F(a))}{\Omega}$. Let us set $\Delta = \frac{\sum_{a \in A} time(a) \cdot F(a)}{\Omega}$, and for any $a \in A$, we define a weight $w(a) = \frac{time(a) \cdot F(a)}{\Omega}$ (notice that every $w(a)$ is nonnegative). Our problem becomes to search for $U' \subseteq X$, which contains d and it is such that:

$$\sum_{a \in \partial_G^+(U')} F(a) + \sum_{a \in A(U', U')} w(a) < \Delta \tag{E10}$$

In order to search for such a set U', we construct an auxiliary digraph $G' = (X', A')$. The vertex set X' of G' consists of X augmented with a new auxiliary vertex $\hat{t}$. Then with any arc $a = (x, y)$ in A, we associate an arc $\hat{a} = (x, \hat{t})$, and we denote

by $\hat{A}$ the set of all arcs $\hat{a}$, such that $a \in A$. The arc set A' of G' is $A \sqcup \hat{A}$. Now, we provide any arc $a \in A'$ with a weight $w'(a) \geq 0$, defined as follows.

- If $a \in A$, then we set $w'(a) = F(a) - w(a)$.
- If $\hat{a} \in \hat{A}$ is associated with $a \in A$, then we set $w'(\hat{a}) = w(a)$.

This construction is illustrated in Figure 2.6.

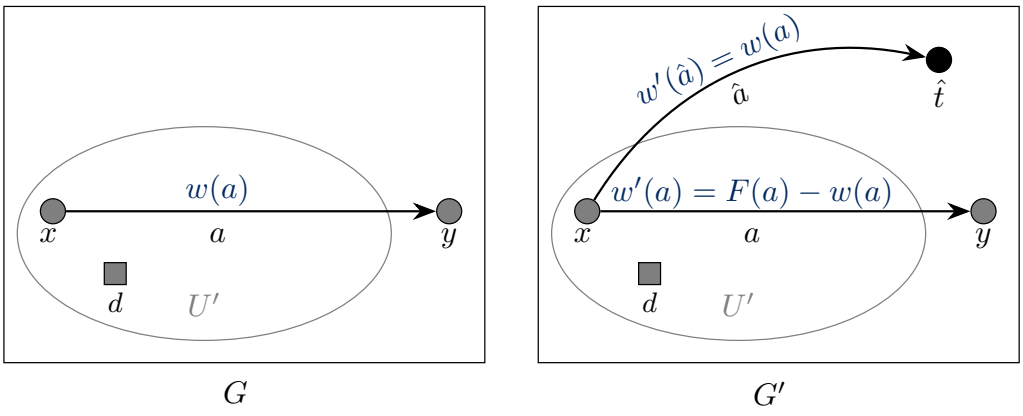


Figure 2.6: Switching from a digraph G to the auxiliary digraph G'.

Let us prove now that: *Searching for $U' \subseteq X$, with $d \in U'$ and such that* (E10) *holds, it is equivalent to search for $U' \subseteq X$, with $d \in U'$ and such that* $\sum_{a \in \partial^+_{G'}(U')} w'(a) < \Delta$. (E11)

If we prove (E11), then we get our separation result, since U' which meets (E11) is nothing but a d-$\hat{t}$-cut in the digraph G', whose weight according to w' does not exceed Δ. Its existence may be checked with a standard minimum cut algorithm.

So let us check (E11). In order to do so, we consider $U' \subseteq X$, which contains d, and compare the quantity $\sigma_1 = \sum_{a \in \partial^+_G(U')} F(a) + \sum_{a \in A(U', U')} w(a)$, and the quantity $\sigma_2 = \sum_{a \in \partial^+_{G'}(U')} w'(a)$. We are going to prove first that $\sigma_1 \leq \sigma_2$. Let a be an arc in A, the following four cases have to be considered.

- *Case 1:* If arc $a = (x, y) \in A(U', U')$ then arc a counts with a weight of $w(a)$ in σ_1, and arc $\hat{a}$ counts with a weight of $w(a)$ in σ_2 (see Figure 2.7).

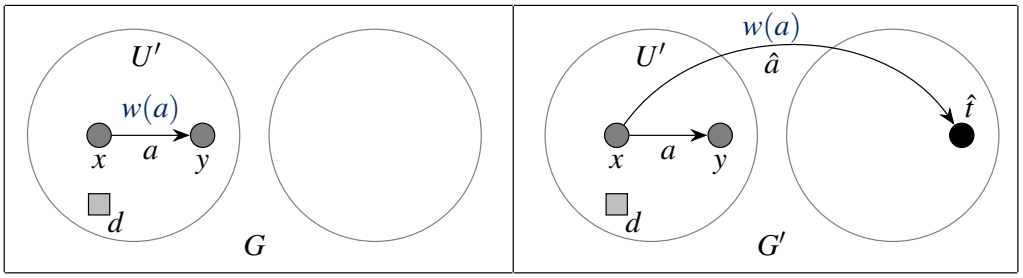


Figure 2.7: An arc $a = (x, y) \in A(U', U')$ in the digraph G and its two associated arcs in the auxiliary digraph G'.

- *Case 2:* If arc $a = (x, y) \in \partial_G^+(U')$ then, arc a counts with a weight of $F(a)$ in σ_1; the arcs a and $\hat{a}$ count in σ_2 with a weight of $w(a) + \big(F(a) - w(a)\big) = F(a)$ (see Figure 2.8).

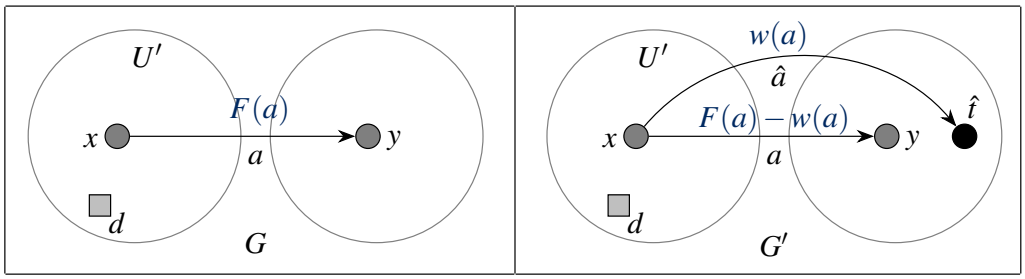


Figure 2.8: An arc $a = (x, y) \in \partial_G^+(U')$ in the digraph G and its two associated arcs in the auxiliary digraph G'.

- *Case 3:* If arc $a = (x, y) \in \partial_G^-(U')$ then arc a does not count in σ_1; the arcs a and $\hat{a}$ do not count in σ_2 (see Figure 2.9).

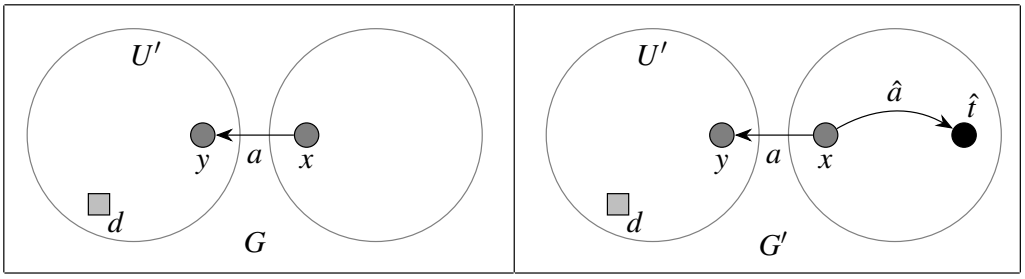


Figure 2.9: An arc $a = (x, y) \in \partial_G^+(U')$ in the digraph G and its two associated arcs in the auxiliary digraph G'.

- *Case 4:* If arc $a = (x, y) \in A(X \setminus U', X \setminus U')$ then the arc a does not count in σ_1; the arcs a and $\hat{a}$ do not count in σ_2 (see Figure 2.10).

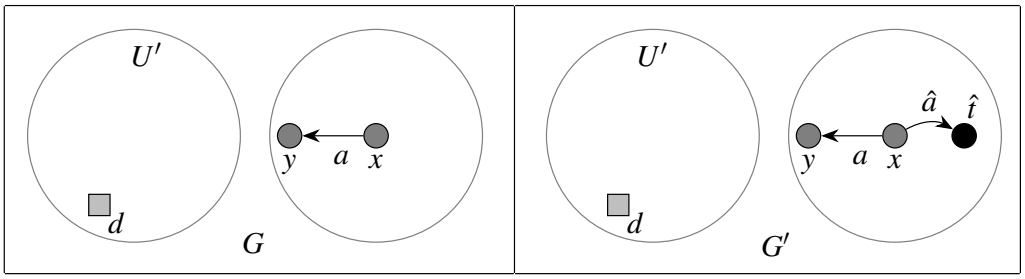


Figure 2.10: An arc $a = (x, y) \in A(X \setminus U', X \setminus U')$ in the digraph G and its two associated arcs in the auxiliary digraph G'.

It follows that $\sigma_1 = \sum_{a \in \partial_G^+(U')} F(a) + \sum_{a \in A(U', U')} w(a) \leq \sum_{a \in \partial_{G'}^+(U')} w'(a) = \sigma_2$.

Conversely, if $u \in A'$, then either u can be written $u = a$ with $a = (x, y) \in A$ or it can be written $u = \hat{a}$, with $a \in A$. In both cases we can see that we enter into one of the above configurations of *Case 1*-*Case 4*. Therefore, we deduce that $\sigma_1 = \sum_{a \in \partial^+_G(U')} F(a) + \sum_{a \in A(U',U')} w(a) \geq \sum_{a \in \partial^+_{G'}(U')} w'(u) = \sigma_2$ and so we conclude.

■

Theorem 2.4 - Complexity of the Strong-Extended-Subtour constraints separation *The Strong-Extended-Subtour constraints* (*E*8.2) *can be separated in polynomial time, by a sequence of* $|X|^2$ *applications of a minimum cut algorithm.*

Proof. Let us suppose, as at the beginning of the proof of Theorem 2.3, that we are provided with a current flow F. So we denote by Λ^- the set of all values $time(d, x)$, such that $x \in X$, and by Λ^+ the set of all values $time(y, d)$, such that $y \in X$, and for any pair $(L^-, L^+) \in \Lambda^- \times \Lambda^+$, we set $X(L^-, L^+) = \{x \in X : time(d, x) \leq L^-\} \cup \{y \in X : time(y, d) \leq L^+\}$ (see Figure 2.11 (a)).

Now, we denote by $G''(L^-, L^+) = (X'', A'')$ the digraph obtained by identifying the vertices of $X(L^-, L^+)$ into a single vertex $d^{L^+}_{L^-}$. In the rest of this proof, we abbreviate the digraph $G''(L^-, L^+)$ simply as G''. We associate an arc $a'' = (d^{L^+}_{L^-}, y)$ with any arc $a = (x, y)$, such that $x \in X(L^-, L^+)$, $y \notin X(L^-, L^+)$, and also an arc $a'' = (y, d^{L^+}_{L^-})$ with any arc $a = (y, z)$, such that $y \notin X(L^-, L^+)$, $z \in X(L^-, L^+)$. We provide these associated arcs with a weight $time''(a'') = time(a)$ and a flow $F''(a'') = F(a)$ (see Figure 2.11 (b)). All arcs $a = (y, w) \in A$ such that y and w are not in $X(L^-, L^+)$, are maintained as arcs of G'', with a weight $time''(a) = time(a)$ and a flow $F''(a) = F(a)$.

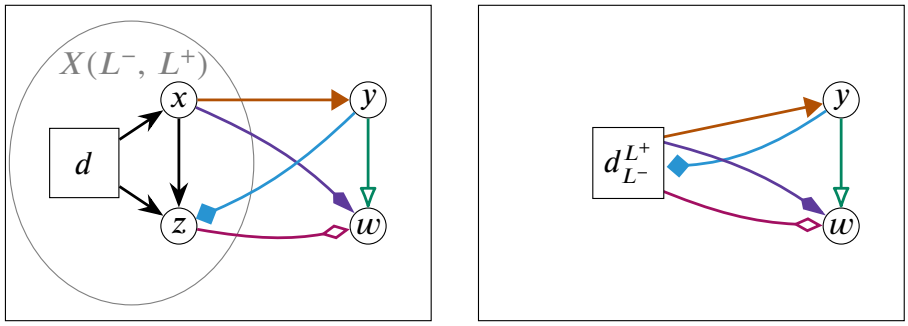


Figure 2.11: Identifying $X(L^-, L^+)$ into a single vertex $d^{L^+}_{L^-}$. (a) An example of set $X(L^-, L^+)$ in a digraph G. (b) The digraph $G''(L^+, L^-)$ corresponding to (a).

Then in order to separate constraints (E8.2), we first notice that all previous developments, including Theorem 2.3, hold even in the case when G is a multidigraph, that means when several arcs a, provided with distinct weights $time(a)$ and $cost(a)$, may connect a same pair of vertices (x, y).

Next, for any pair $(L^-,\ L^+)$ in $\Lambda^- \times \Lambda^+$, we build the corresponding network G'', and next we separate (E8.1) in the sense of Theorem 2.3, while replacing Ω by $(\Omega - L^- - L^+)$, digraph G by digraph G'', flow F by flow F'', and weight *time* by weight *time*$''$.

By doing this, we get either a `success` signal (no subset $U \subseteq X \setminus X(L^-, L^+)$ which violates (E8.2) in above sense, or a subset $U \subseteq X \setminus X(L^-, L^+)$ which does it. So we claim the following statement.

Claim. *There exists $U \subseteq X \setminus \{d\}$ such that, F violates* (E8.2) *if and only if there exist L^- and L^+ and $U' \subseteq X \setminus X(L^-, L^+)$ such that*

$$(\Omega - L^- - L^+) \cdot \sum_{a \in \partial^-_{G''}(U')} F''(a) < \sum_{a \in \partial_{G''}(U') \cup A''(U', U')} time''(a) \cdot F''(a). \qquad \text{(E12)}$$

Let us first check the (if) part of above claim. In order to do so, we consider U such that (E8.2) is violated by F in the sense of digraph G, that means:

$$(\Omega - d^-_U - d^+_U) \cdot \Big(\sum_{a \in \partial^-_G(U)} F(a) \Big) < \sum_{a \in \partial_G(U) \cup A(U,U)} time(a) \cdot F(a).$$

We may select x and y both in $X \setminus U$ such that $d^-_U = time(d, x)$ and $d^+_U = time(y, d)$. If we set $L^- = time(d, x)$ and $L^+ = time(y, d)$, then we have that $X(L^-, L^+) \subseteq X \setminus U$, which means that U can be considered as a subset U' which results in the separation of (E8.1) in digraph G''.

Let us now check the (only if) part of previous claim. In order to do so, we consider L^-, L^+ and related U' as above. We first notice that, for any L^-, L^+, and any $U' \subseteq X \setminus X(L^-, L^+)$, we have $d^-_U \geq L^-$ and $d^+_U \geq L^+$. Then we see that $\sum_{a \in \partial^-_G(U)} F(a) = \sum_{a \in \partial^-_{G''}(U')} F''(a)$ and $\sum_{a \in \partial_G(U) \cup A(U,U)} time(a) \cdot F(a) = \sum_{a \in \partial_{G''}(U') \cup A''(U', U')} time''(a) \cdot F''(a)$, since arcs a in $A(U, U)$ are not modified by the fusion of vertices of $X(L^-, L^+)$ and since arcs a of $\partial_G(U)$ are turned into arcs a''. Now we note that, if U' is such that $(\Omega - L^- - L^+) \cdot \big(\sum_{a \in \partial^-_{G''}(U')} F''(a) \big) < \sum_{a \in \partial_{G''}(U) \cup A''(U', U')} time''(a) \cdot F''(a)$, then we also have $(\Omega - d^-_U - d^+_U) \cdot (\sum_{a \in \partial^-_G(U)} F(a)) < \sum_{a \in \partial^-_G(U) \cup A(U,U)} time(a) \cdot F(a)$, since $d^-_U \geq L^-$ and $d^+_U \geq L^+$. This completes the proof of the above claim and concludes the proof of the theorem. ■

A Separation Algorithm

We separate both Extended-Subtour constraints (E8.1), (E8.2) in a hybrid way, while applying the Min-Cut procedure designed for constraints (E8.1) and sending back the related constraints (E8.2) to the main branch-and-cut process.

This gives rise to the following Algorithm 4.

Algorithm 4: Separation of constraints (E8.1)

input : Digraph G, flow F, and time horizon $[0, \Omega]$.

output: A `success` signal if the constraints (E8.1) are satisfied by F, otherwise a (E8.2) constraint violated by F.

```
Derive from F, G, and Ω the network G' described in the proof of Theorem 2.3;
Apply the Ford-Fulkerson algorithm to compute a maximum d-t̂-flow φ in G', and let
  val(φ) be the resulting flow value;
if val(φ) ≥ (Σ_{a∈A} time(a)·F(a)) / Ω then
    return success;
else
    Retrieve an d-t̂-cut ∂⁺(U) with d ∈ U, t̂ ∈ X ∖ U, and capacity val(φ);
    return constraint (E8.2) related to U;
```

The correctness of Algorithm 4 derives from the procedure to compute a d-$\hat{t}$-cut of minimum capacity using the Ford-Fulkerson algorithm (see Theorem 1.2 and the three-step procedure that follows it).

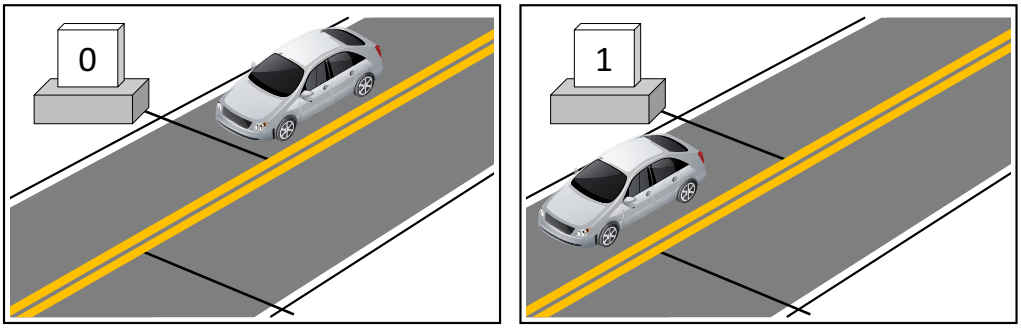
2.5 The Lift Problems

The projected model that we have described in Section 2.4 can be very useful because, once we have found an optimal solution for a problem instance, we obtain a lower bound for the value of any optimal solution for the corresponding Item Relocation Problem, and we can also get an idea about which are the vertices and arcs that should be used by the vehicles to perform the relocation process. It turns out that even with all this information, it can be difficult to determine a solution for the Item Relocation Problem.

In this section we will see that, given a solution of a PIRP instance, it is not always possible to construct an IRP solution with the same cost that uses exactly the same vertices and arcs. Therefore, we will introduce some problems related to the ways in which, starting from a solution for the PIRP model, we can construct feasible solutions for the corresponding Item Relocation Problem.

Before starting with the formal study of the problems that will be dealt in this section, we present a simpler problematic arising from a real life situation. It will give us some intuition about the problems that we are going to face later.

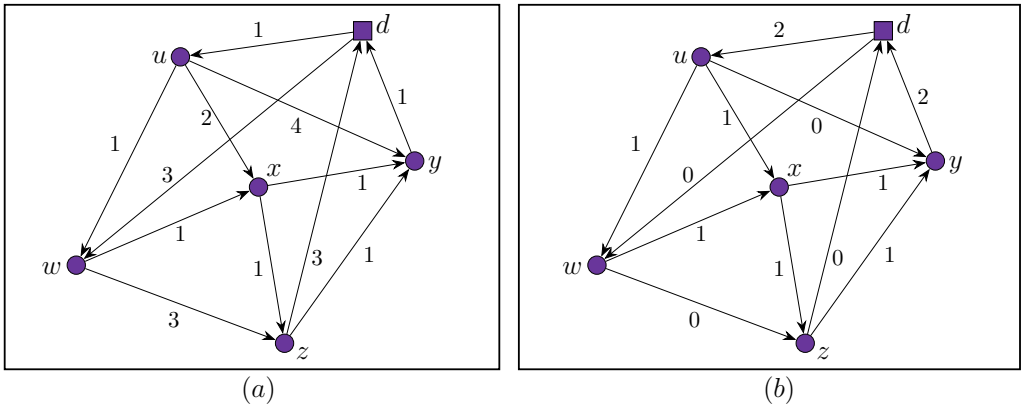
Example 2.3 - Pneumatic road tube counters
A pneumatic road tube counter (see [167]) is a temporary electronic traffic recording device that sends a burst of air pressure along a rubber tube when the tires of a vehicle pass over the tube. The pressure pulse closes an air switch, producing an electrical signal that is transmitted to a counter (see Figure 2.12).



(a) (b)

Figure 2.12: (a) A pneumatic road tube counter, indicating that zero vehicles have passed in a particular sense of a street. (b) The same pneumatic road tube counter in (a), but after the passing of a vehicle. Now, the counter indicates that one vehicle has passed.

Consider the transit network depicted in Figure 2.13 (a) consisting only of one-way streets. Suppose that we have installed one pneumatic road tube counter initialized at zero on each street, and that there is an unknown number of vehicles located at vertex d at time zero.



(a) (b)

Figure 2.13: The transportation networks from Example 2.3. (a) Any numerical label on an arc indicates the units of time that are necessary to traverse the arc. (b) Any arc label represents the value shown by the pneumatic road tube counter that is installed on the corresponding arc.

We allow those vehicles to circulate on the transit network during six units of time, without paying attention to the process. At time six, we find out that all the vehicles have already returned to vertex d, and the pneumatic road tube counters indicate the values shown in Figure 2.13(b).

Now, we are required to answer: how many vehicles have transited on the network between time zero and time six?

To answer this question in a more systematic way, we are going to construct an auxiliary multidigraph using the original transit network, and the values in the pneumatic rubber tube counters. The construction is as follows. First, we start with an empty digraph with the same vertices as the original transit network. Then, for every arc (x, y) in the transit network, we put in the multidigraph as many arcs (x, y) as indicates the corresponding pneumatic rubber tube counter. Finally, we remove all the isolated vertices. Note that, every arc on this multidigraph represents the pass of a single vehicle through the corresponding street. Also, it is not difficult to prove♣ that we always obtain an Eulerian multidigraph with this construction.

If we take the digraph in Figure 2.13(a) together with the values of the pneumatic road tube counters displayed in Figure 2.13(b), and we follow the above construction, we obtain the multidigraph depicted in Figure 2.14.

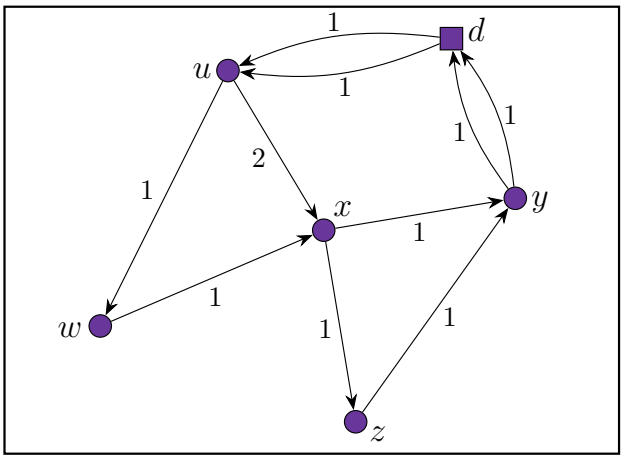


Figure 2.14: The multidigraph used in Example 2.3. Any numerical label on an arc indicates the units of time that are necessary to traverse the arc.

Because this multidigraph is Eulerian, in principle, we could explore it with a single vehicle, passing only once through each arc. However, if we follow any Eulerian path, we will find out that using a single vehicle we do not have enough time to traverse all the arcs. Then, we need to use at least two vehicles. On the other hand, we can see that there are two arcs going out from vertex d, and hence there were at most two vehicles leaving vertex d. Therefore the answer is two vehicles. □

♣Provided that all vehicles started in a depot and returned to it.

The question posed in Example 2.3 cannot be answered in general because the value of a pneumatic road tube counter represents only the number of times that the corresponding street has been traversed by vehicles. For example, if we do not have enough time to follow an Eulerian path, and a counter indicates a number greater than one, it is not always possible to deduce if some car has transited several times on the same street. As a consequence, we cannot determine, in general, how many vehicles there were on the transit network.

The following example aims to introduce the notion of "lift".

Example 2.4 - Lifting a projected solution
Let us return to the situation presented in Example 2.3. Suppose that this time we are required to construct the paths that were followed by the two vehicles. How should we proceed?

It is not difficult to see that the temporal dimension of the process has been lost because the pneumatic rubber tube counters do not allow us to know at which times the vehicle have passed. On the other hand, the vehicle's routes involve a temporal dimension because any vehicle requires an amount of time to traverse a street, and the vehicle routes induce precedence relations on the multidigraph arcs.

Now, consider the subgraph of our transportation network that is induced by the arcs that were visited by at least one vehicle during the transportation process, and imagine that we have drawn this subgraph on a plane surface (see Figure 2.15 (a)). Suppose that we have a collection of real-size elastic stickers representing the arcs, and that we paste, by matching the shape of the corresponding arc in the drawing, as many stickers of the arc as the corresponding pneumatic rubber counter indicates.

If we proceed to peel off the stickers, taking one sticker at a time, and concatenating the head of the current sticker with the tail of the previously detached one, and we achieve to obtain paths starting and ending at vertex d as in Figure 2.15 (b), then we would have obtained a solution for the problem.

Nevertheless, it is not difficult to find other different solutions for this problem, and therefore we cannot decide precisely which one corresponds to the routes followed by the vehicles.

The mathematical analogy of the above process is as follows. In Figure 2.15 (a) we have a weighted digraph embedded in a plane, whilst in Figure 2.15 (b) we have a digraph in a three-dimensional space with a new temporal component t. Furthermore, if we project the digraph in Figure 2.15 (b) by dropping the t component, then we obtain again the digraph in Figure 2.15 (a). □

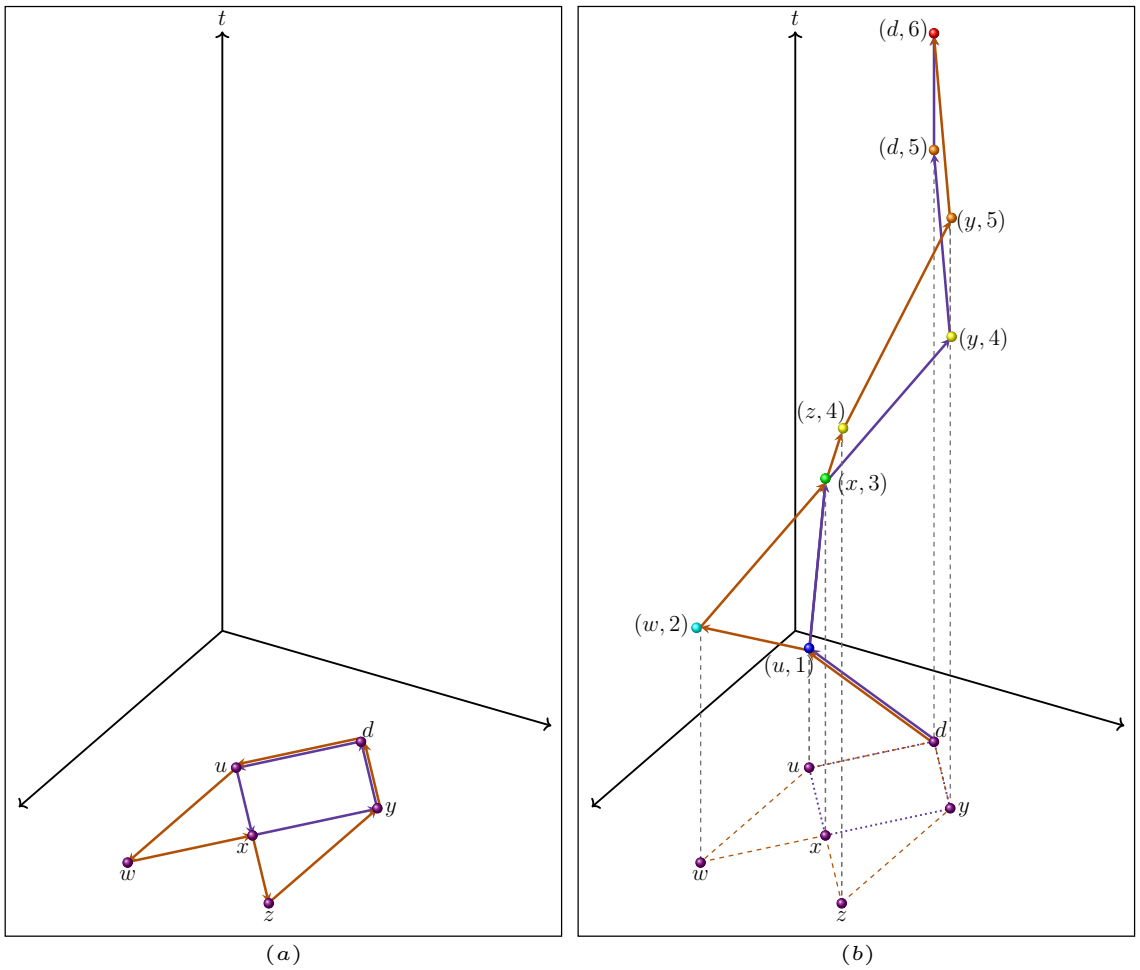


Figure 2.15: An example of "lift". (a) A digraph embedded on a plane surface. (b) "Lifting" the digraph in (a) into a three-dimensional space with a new component t.

Examples 2.3 and 2.4 allow us to establish some analogies with the problems that we will treat in this section. First of all, it is not difficult to conceive an interpretation of the PIRP solutions in terms of pneumatic rubber tube counters. The idea is to install two kinds of pneumatic rubber tube counters on each one-way street: one for the vehicles and the other one for the items. Next, we will see that, although this interpretation can be very illustrative, it is not completely realistic.

It is true that given a solution of an IRP, we can deduce trivially a 2-commodity flow which is a solution of the corresponding PIRP model; and conversely the PIRP solutions obtained that way can be lifted again (at least in the theory) to obtain the original IRP solution from which they have been projected.

However, the projected model involves integral flows, and so, they are more general than the integral flows induced by the circulation of discrete objects (like vehicles or items) on a transit network. In particular, integral flows can exhibit a counterintuitive behavior (like ubiquity) on certain parts of a network and this in

turn can give rise to some issues that are difficult to remediate when we try to lift a projected solution into a solution for the corresponding relocation problem.

The following two examples illustrate some of those situations.

Example 2.5 - The "mushroom"
The digraph in Figure 2.16 shows that a solution (F, f) of the PIRP cannot always be viewed as the projection of a feasible solution (H, h) of IRP TEN. In Figure 2.16, we see that the vehicle follows the route (d, y, x, z, y, d), but cannot transport that way any item from z to x. □

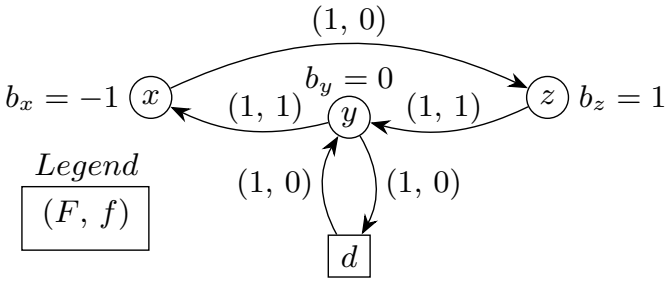


Figure 2.16: A solution of PIRP which cannot be lifted.

Example 2.6 - The "helix"
Every arc of the digraph depicted in Figure 2.17 requires one unit of time to be traversed. If we consider vehicles with a capacity $\kappa = 2$, and a time horizon $\Omega = 3$ (i.e. three units of time), then the values F and f that are shown in this figure correspond to a solution of some PIRP instance.

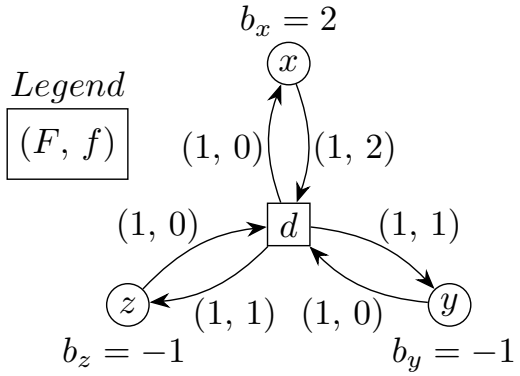


Figure 2.17: A solution of PIRP which cannot be lifted.

However, lifting this solution would require at least two units of time to transport the two items at vertex x, to vertex d; and then we need at least two additional units of time to carry one of those items to y (or to z) and return to d. □

Examples 2.5 and 2.6, show that if we insist in lifting a solution for the Item Relocation Problem that has the same quality and uses exactly the same arcs and vertices than a given projected solution, it is not always possible to lift such a solution.

This raises in a natural way the following "lift" issue: given a feasible (optimal) PIRP solution $(F,\ f)$, how can we derive from $(F,\ f)$ a feasible/optimal IRP solution, according to the TEN framework? Clearly, though we refer here to a specific relocation problem, this issue may be set in a far more generic setting: how can we efficiently deal with a problem originally set on a TEN while applying the following decomposition scheme?

***Project-and-Lift* Decomposition Scheme**

1. Solve a projected version of the problem which skips the temporal dimension.
2. Turn (i.e., lift) the resulting solution $(F,\ f)$ into a *"good"* solution $(H,\ h)$ of the original problem, while restricting ourselves to a reduced representation of the related TEN.

2.5.1 Two Lift Problems

For all the examples presented in this section, we are going to consider that we are given an IRP instance over a graph $G = (X, A)$, and that we have computed an optimal 2-commodity flow (F, f) of the corresponding PIRP model, with respect to the cost parameters $\alpha = 10$, $\beta = 1$, and $\gamma = 1$.

The most natural way to formalize this lift issue consists in setting the following Strong Lift Problem.

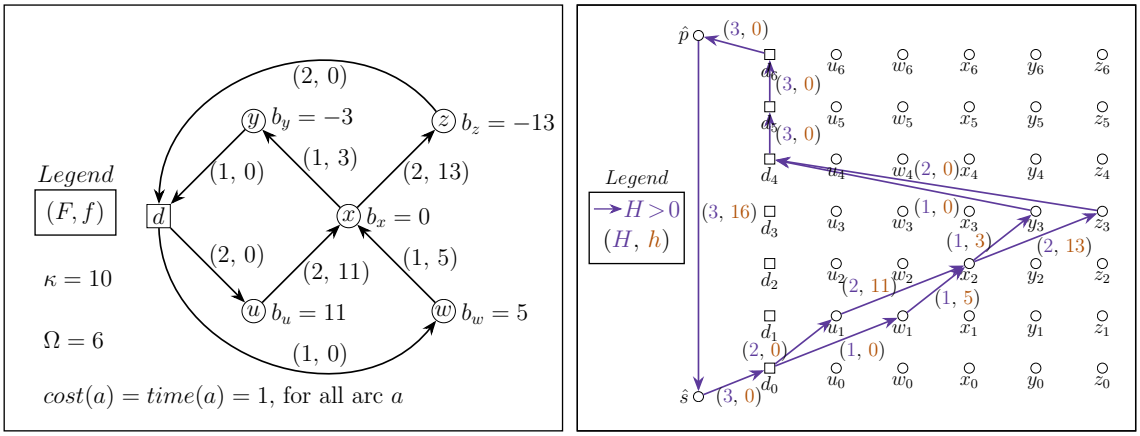
Strong Lift Problem: Given an IRP instance on a digraph $G = (X, A)$ and a PIRP solution $(F,\ f)$. Compute a feasible IRP solution $(H,\ h)$ so that

- the projection of H (respectively, h) on the transit network G is equal to F (respectively, f);
- the cost value $Cost(H,\ h)$ is smallest possible.

Remark 2. If $(H,\ h)$ is a feasible solution of the previous Strong Lift Problem then the difference between $Cost(H,\ h)$ and $PCost(F, f)$ only reflects the difference between the true number of vehicles $H(\hat{p}, \hat{s})$ and its approximation $\frac{\sum_{a\in A} time(a)\cdot F(a)}{\Omega}$ as expressed in the PIRP Model.

Example 2.7 - An example of the Strong Lift Problem

The digraph depicted in Figure 2.18 (a) shows an optimal solution (F, f) of a PIRP instance. Suppose that we have $cost(a) = time(a) = 1$, for every arc a. If we consider a fleet of vehicles with capacity $\kappa = 10$, and a time horizon $\Omega = 6$, then we would have that $\frac{\sum_{a\in A} time(a)\cdot F(a)}{\Omega} = \frac{2+1+1+2+1+2+2+1}{6} = 2$. So, from this projected solution we can estimate that two vehicles are necessary to perform the relocation process.



(a) (b)

Figure 2.18: An example of the Strong Lift Problem. (a) An optimal solution (F, f) of a PIRP instance. (b) A solution (H, h) for the TEN IRP such that it is the minimal cost solution with time-projection equal to the 2-commodity flow (F, f) in (a). We only have depicted the arcs a with $H(a) > 0$.

On the other hand, we can observe that there are three units of flow F leaving vertex d, and that when a vehicle leaves the depot vertex d, it needs at least four units of time to return to vertex d. If we try to use only two vehicles to perform the relocation process, one of the vehicles would be obligated to leave the depot vertex two times, and then it would require at least eight units of time to complete its route. However, we have a time horizon $\Omega = 6$, and therefore it is not possible to use only two vehicles to perform a relocation process with time-projection (F, f).

The TEN IRP solution (H, h) shown in Figure 2.18(b) is the minimal cost solution among all the IRP solutions with a time projection equal to the 2-commodity flow (F, f) in (a). Note that, because the flow H involves three vehicles, the cost of (H, h) exceeds by α the cost of (F, f). □

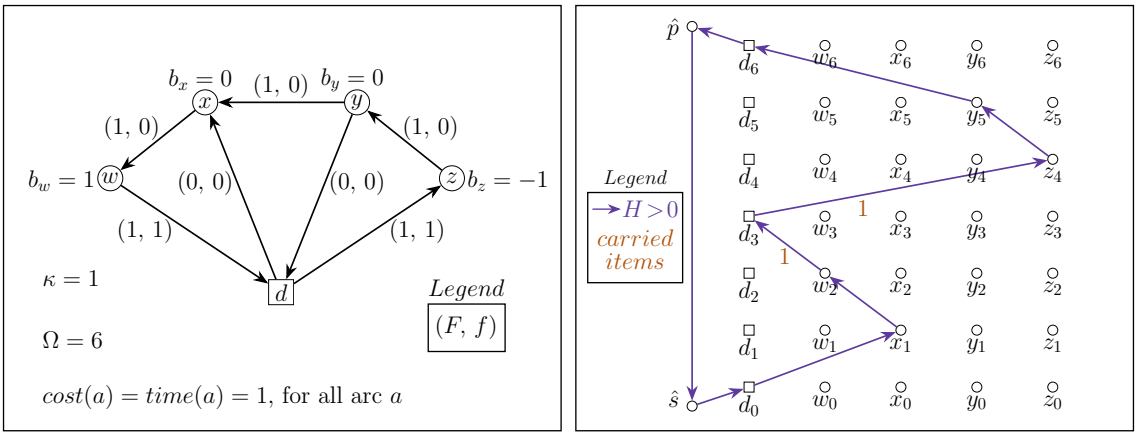
As we shall see in Chapter 3, solving this problem is difficult. Worse, numerical experiments will show that, in many cases, the Strong Lift Problem does not admit any feasible solution. This leads us to propose a more flexible version of Lift problem, which does not involve flow F.

Partial Lift Problem: Given an IRP instance on a digraph $G = (X, A)$ and an item flow f. Compute a feasible IRP solution (H, h) so that

- the projection of h on the digraph G is equal to f;
- the cost value $Cost(H, h)$ is smallest possible.

Example 2.8 - An example of the Partial Lift Problem

Figure 2.19 (a) shows an example of a PIRP solution (F, f). If we consider a fleet of vehicles with capacity $\kappa = 1$, a time horizon $\Omega = 6$, and suppose that $cost(a) = time(a) = 1$, for every arc a in the depicted digraph; then the 2-commodity flow (F, f) cannot be lifted into a TEN IRP solution



(a) (b)

Figure 2.19: An example of the Partial Lift Problem. (a) A PIRP solution (F', f') on a digraph $G = (X, A)$. (b) A solution (H, h) for the TEN IRP with optimal cost among all the the TEN IRP solutions whose time-projection (F', f') is such that $f' = f$. Here, H is a unitary flow, and we only have depicted the arcs a with $H(a) > 0$. Arcs labels indicate the number of carried items.

On the other hand, it is not difficult to show that the 2-commodity flow (H, h) represented in Figure 2.19 (b) is an optimal TEN IRP solution among all the TEN IRP solutions whose time-projection (F', f') is such that $f' = f$. Note that F and F' are not comparable (that is $F \not\geq F'$, and $F \not\leq F'$). □

2.5.2 Feasibility of the Partial Lift Problem

Once again, we must first address feasibility. In order to answer it, we introduce the following notion of "feasible-path".

- A *feasible-path* (viewed as a collection of arcs) of $G = (X, A)$ is any path π from $x \in X^+$ to $y \in X^-$ whose weight $time(\pi)$ in the sense of the weight function $time$ is such that: $time(d, x) + time(\pi) + time(y, d) \leq \Omega$. We associate, with any such feasible-path, a flow vector $\mathbf{e}^\pi$ which transports one item from x to y along the path π. We denote by Π_G^Ω the set of all feasible-paths.

- A flow f is *feasible-path-decomposable* if and only if, for all $a \in A$ we can write $f(a) = \sum_{\pi \in \Pi_G^\Omega} \lambda_\pi \ e_a^\pi$, with $\lambda_\pi \in \mathbb{R}_+$ for all $\pi \in \Pi_G^\Omega$.

- We say that a Π_G^Ω-indexed vector $\mathbf{w} = (w_\pi, \pi \in \Pi_G^\Omega)$ is a *feasibility-path* vector if, for any feasible-path π, we have $\sum_{a \in \pi} w_a \geq 0$.

The meaning of the notion of feasible-path is obvious: an item starting from a vertex $x \in X^+$ can be transported to some vertex $y \in X^-$ along some path π only if π is a feasible-path. It follows that a solution $(F,\ f)$ of the PIRP model may be lifted into a feasible IRP solution $(H,\ h)$ only if f is feasible-path-decomposable. This leads us to the following characterization of the feasibility of the Partial Lift Problem.

Theorem 2.5 - A Characterization of the Partial Lift Problem Feasibility *The Partial Lift Problem admits a feasible solution if and only if for any feasibility-path vector* $\mathbf{w}$*, the following inequality holds:*

$$\sum_{a \in A} w_a \cdot f(a) \geq 0. \tag{F2}$$

Proof. Necessity follows in a straightforward way from the above explanation. As for sufficiency, we get it by noticing that (F2) is nothing more than a formulation of Farkas Lemma (see [83]) in the case of vector $\mathbf{f} = (f(a), a \in A)$ and the vector collection $\{\mathbf{e}^\pi = (e_a^\pi, a \in A), \pi \in \Pi_G^\Omega\}$: flow f is feasible-path-decomposable if and only if vector $\mathbf{f}$ belongs to the cone defined by the collection $\{\mathbf{e}^\pi, \pi \in \Pi_G^\Omega\}$, that means (by Farkas Lemma) if and only if for any vector $\mathbf{w}$ whose scalar product with any vector $\mathbf{e}^\pi$ is nonnegative (i.e., $\mathbf{w} \cdot \mathbf{e}^\pi = \sum_{a \in A} w_a \cdot e_a^\pi \geq 0$, for all $\mathbf{e}^\pi$ with $\pi \in \Pi_G^\Omega$), then the scalar product $\mathbf{w} \cdot \mathbf{f}$ is also nonnegative. ■

2.5.3 Enhancing the PIRP Model with Feasible-Path Constraints

This characterization of the feasibility of the Partial Lift Problem suggests us to enhance our PIRP model with an additional constraint which imposes the flow f to be feasible-path-decomposable. This *Feasible-Path* constraint reads as follows.

- For any feasibility-path vector $\mathbf{w}$, the following inequality holds:

$$\sum_{a \in A} w_a \cdot f(a) \geq 0. \tag{E13}$$

Handling (E13) can be done through branch-and-cut and a separation scheme which relies on a current collection Π of feasible-paths, according to the algorithmic scheme described in Algorithm 5.

Algorithm 5: Feasible-Path constraints separation scheme.

input : Flow vector $\mathbf{f} = (f(a), a \in A)$, a parameter $\varepsilon > 0$, and a collection Π of feasible-paths.

output: A `success` signal if $\mathbf{f}$ is feasible-path-decomposable, otherwise a Feasible-Path constraint (E13) which is not satisfied by $\mathbf{f}$.

```
stop ← False ;
while not stop do
    if linear system f = Σ_{π∈Π} y_π e^π has a solution y = (y_π, π ∈ Π) then
        stop ← True, and return success
    else
        Retrieve (by Farkas Lemma) some vector w = (w_a, a ∈ A) such that
          Σ_{a∈A} w_a · f_a < 0, and Σ_{a∈A} w_a · e^π_a ≥ 0 for all π ∈ Π;
        Search for a feasible-path^(*) π_0 such that Σ_{a∈π_0} w_a ≤ −ε ;
        if π_0 has been found then
            add π_0 into current collection Π;
        else
            stop ← True, and return constraint Σ_{a∈A} w_a · f_a ≥ 0
```

(*) **Instruction** "Search for a feasible-path" **.** This instruction refers to the Constrained Shortest Path Problem, which is weakly $\mathcal{NP}$-complete (see [4]) and can be handled with moderate computational effort through tree search algorithms (see [151]). For simplicity we have chosen to formulate that Constrained Shortest Path Problem as a constraint system $\mathsf{CPATH}(\mathbf{w})$ and solve it with a MILP solver (see the following details of implementation).

Details of Implementation

We start by constructing a supergraph $G(\hat{s},\hat{t})$ of digraph G in the following way.

First, we add to $G(\hat{s},\hat{t})$ all the vertices and arcs of G, and for every arc $a \in A$ we define $time_{G(\hat{s},t)}(a) = time_G(a)$. We also add two new auxiliary vertices: a source vertex $\hat{s}$ and a sink vertex $\hat{t}$. We add an arc from $\hat{s}$ to every excess vertex x. Each of those arcs $(\hat{s},x)$ has a weight $cost_{G(\hat{s},\hat{t})}((\hat{s},x)) = \text{dist}_{(G,cost)}(d,x))$ and a weight $time_{G(\hat{s},\hat{t})}((\hat{s},x)) = \text{dist}_{(G,time)}(d,x)$. Symmetrically, we add an arc from every deficit vertex y to vertex $\hat{t}$. Each of those arcs $(y,\hat{t})$ has a weight $cost_{G(\hat{s},t)}((y,t)) = \text{dist}_{(G,cost)}(y,d)$ and a weight $time_{G(\hat{s},\hat{t})}((y,t)) = \text{dist}_{(G,time)}(y,d)$.

Next, we define the vector $\mathbf{f} = (f(a), a \in A(G))$ and we construct a matrix $\mathbf{A}$ such that every column of $\mathbf{A}$ is the characteristic vector (indexed over $A(G)$) of a path π in G that goes from an excess vertex x to a deficit vertex y, and such that $time_G((d,x)) + time_G(\pi) + time_G((y,d)) \leq \Omega$. For example, we might start with a matrix $\mathbf{A}$ consisting of all the characteristic vectors of the minimum weight x-y-paths π (with respect to the weight $time$) such that x is an excess vertex, y is a deficit vertex, and $time_G((d,x)) + time_G(\pi) + time_G((y,d)) \leq \Omega$.

We try to solve the linear system $\mathbf{Ay} = \mathbf{f}$. If the system is feasible, then the item flow f can be decomposed into a collection of paths all with a weight $time$ less than or equal to the time horizon Ω, and we are done. Otherwise, if the linear system is not feasible, then we compute a Farkas certificate vector $\mathbf{w}$ (again indexed over $A(G)$) such that $\mathbf{w}\cdot\mathbf{f} > \mathbf{0}$ and $\mathbf{wA} \leq \mathbf{0}$, we select a threshold value $\varepsilon > 0$ (for example $\varepsilon = 0.01$), and we construct the following constraint system.

CPATH(w) constraint system

Variables

- $clock_x \in \mathbb{R}^+$ for all $x \in V(G)$.
- $arc_a \in \{0,1\}$ for all $a \in A(G(s,t))$.

Constraints

- For all $x \in V(G)$: $\text{dist}_{(G,time)}(d,x) \leq clock_x \leq \Omega - \text{dist}_{(G,time)}(x,d)$.
- $\sum_{a \in \partial^-_{G(\hat{s},\hat{t})}(\hat{s})} arc_a = 1$.
- For all $x \in V(G(\hat{s},\hat{t})) \setminus \{\hat{s},\hat{t}\}$: $\sum_{a \in \partial^-_{G(\hat{s},\hat{t})}(x)} arc_a = \sum_{a \in \partial^+_{G(\hat{s},\hat{t})}(x)} arc_a$.
- For every $a = (x,y) \in A(G)$:

$$clock_x + \text{dist}_{(G(\hat{s},\hat{t}),time)}(x,y) \leq clock_y + \Omega(1 - arc_a).$$

- $\sum_{a \in A(G(s,t))} time_{G(\hat{s},\hat{t})}(a) \cdot arc_a \leq \Omega$.
- $\sum_{a \in A(G)} w_a \cdot arc_a > \epsilon$.

If CPATH($\mathbf{w}$) is feasible, then we construct from any feasible solution the corresponding vector $(arc_a, a \in A(G))$, we add this vector as a new column of matrix $\mathbf{A}$, and we restart the process from the step where we try to solve the system $\mathbf{Ay} = \mathbf{f}$.

On the other hand, if the above MILP is infeasible, then the current item flow cannot be decomposed using only paths with a time duration less than or equal to Ω, then to improve the PIRP formulation we can add the constraint $\mathbf{w} \cdot \mathbf{f} \leq \mathbf{0}$ to the PIRP model to forbid the current item flow f.

2.6 Numerical Experiments

We handle the PIRP model through branch-and-cut with the help of a commercial solver. We let the solver decide for the choice of a branching strategy and focus on the design and implementation of the separation procedures.

We separate both Extended-Subtour constraints and Feasible-Path constraints while using the Algorithms 4 and 5 described in Sections 2.4.2 and 2.5.3.

Purpose. The purpose of our experiments is to evaluate the influence on the behavior of the global resolution process (running time, solution value, number of generated cuts, number of nodes of the search tree visited during the process, etc.) of the different constraints (E8), (E13) introduced in sections 2.4.2 and 2.5.2.

Technical context. The experiments were performed on a computer with a 2.3GHz Intel Core i5 processor and 16GB RAM. The implementations were coded in the C++ language, compiled with Apple Clang 10, and made use of the CPLEX12.10 MILP libraries.

An important implementation detail. Note that we can relax the integrality constraint over the item flow variables. This is because if we assign integer values to the vehicle flow variables and we relax the integrality of the item flow variables, the resulting problem is nothing more than a Minimum Cost Flow linear programming problem over the item flow variables, and it always has integral optimal solutions due to the total unimodularity of its constraint matrix. Such a relaxation cannot be guessed by the optimization libraries and may have an important impact on the efficiency of the implementations, because it reduces the number of integer variables.

Instances. No standardized benchmarks exist for the generic IRP. So we built instances as follows: the station set X is a set of n points inside a 100×100 integral grid, the set of arcs A consists of m arcs generated randomly, the weight $time : X \times X \to \mathbb{Z}_+$ corresponds to the rounded Euclidean distance and the weight $cost : X \times X \to \mathbb{Z}_+$ to the taxicab distance. Each vertex x but d is assigned to a balance coefficient b_x in $\{-10, \ldots, 10\}$, the capacity κ is chosen from $\{2, 5, 10, 20\}$, the time horizon limit Ω is a product $\lambda \cdot (\max_{(x,y)\in A} time(x, y))$ when choosing $\lambda \in \{4, 6, 8\}$.

The scaling coefficients α, β, γ are chosen in such a way that the values of cost components c_1 (number of vehicles), c_2 (vehicle ride cost) and c_3 (item ride time) become comparable. Table 2.1 provides us with a characteristics summary of the 20 instances that we analyzed.

Table 2.1: Instance characteristics.

Id	n	m	κ	Ω	λ	α	β	γ
1	20	78	2	324	4	304	1.0	1.000
2	20	65	5	320	4	150	0.4	0.500
3	20	77	10	440	4	328	0.2	0.250
4	20	75	2	680	8	328	1.0	1.000
5	20	50	5	536	8	392	0.4	0.250
6	20	57	10	840	8	376	0.2	0.250
7	20	62	5	420	6	300	0.4	0.500
8	50	163	2	460	4	170	1.0	1.000
9	50	155	5	260	4	196	0.4	0.500
10	50	149	10	440	4	164	1.0	0.500
11	50	146	20	436	4	312	0.1	0.125
12	50	175	2	728	8	268	1.0	1.000
13	50	217	5	912	8	672	0.4	0.250
14	50	154	10	1040	8	416	0.2	0.125
15	100	363	2	336	4	252	1.0	1.000
16	100	236	5	516	4	188	0.4	0.250
17	100	289	10	432	4	360	0.2	0.250
18	100	419	2	1032	8	412	1.0	1.000
19	100	327	5	552	8	392	0.5	0.200
20	100	313	10	712	8	312	0.5	0.500

Results and Comments

We first observe the behavior of the PIRP model when neither of the constraints (E8, E13) nor the approximation of the number of the vehicles provided by Lemma 2.1 are involved. So, for any instance in Table 2.1, we provide in Table 2.2:

- the optimal value **G1** of the PIRP model, computed while considering neither the term $\alpha\big(\sum_{a\in A} time(a)\cdot F(a)\big)/\Omega$ (related to the approximate number of vehicles) in the objective function, nor the Extended-Subtour constraints, nor the Feasible-Path constraints; we also provide the related running time (in seconds) **T1** used to compute **G1**;

- the optimal value **G2** of the PIRP model, computed with the objective function (E9), but without considering the Extended-Subtour constraints, and without considering the Feasible-Path constraints; we also provide the related running time (in seconds) **T2** used in the computation of **G2**, and the estimated number of vehicles **V2** obtained from the expression $\left\lceil \frac{\sum_{a\in A} time(a)\cdot F(a)}{\Omega} \right\rceil$.

Table 2.2: Numerical results for the PIRP model with/without including the term $\alpha\big(\sum_{a\in A} time(a)\cdot F(a)\big)/\Omega$ in the objective function (E9). In these experiments neither the Extended-Subtour constraints nor the Feasible-Path constraints were considered.

Id	G1	T1	G2	T2	V2
1	999.00	0.01	1621.11	0.03	2
2	765.90	0.05	1127.71	0.05	3
3	441.85	0.02	819.47	0.07	2
4	2925.00	0.02	3708.34	0.02	3
5	1226.20	0.02	2323.21	0.03	3
6	1016.20	0.02	1532.38	0.03	2
7	1600.10	0.01	2445.81	0.02	3
8	11606.00	0.04	13951.30	0.08	14
9	2783.70	0.13	4292.61	0.27	8
10	6910.50	0.23	7829.03	0.39	6
11	687.08	0.25	1590.26	0.45	3
12	5640.00	0.06	6878.19	0.10	5
13	777.25	0.45	1525.93	0.96	2
14	1627.28	0.77	2547.22	0.78	3
15	9587.00	1.39	13507.00	4.61	16
16	3113.05	0.40	4391.96	1.38	7
17	1743.65	0.21	3018.55	0.78	4
18	16164.00	0.57	19882.20	0.65	9
19	3146.00	1.44	5748.40	7.51	7
20	4706.50	0.63	5883.62	2.46	4

We next observe the impact of adding constraints (E8, E13). For any instance described in Table 2.1, we provide in Table 2.3:

- the optimal value **G3** (respectively **LB3**) of the PIRP model (respectively of the LP relaxation of the PIRP model) computed with the objective function (E9), with the Extended-Subtour constraints, but without the Feasible-Path constraints; the time value **T3** corresponding to the running time in seconds spent in the computation of **G3**, the estimation **V3** of the number of vehicles used in the solution (obtained from the expression $\left\lceil \frac{\sum_{a\in A} time(a)\cdot F(a)}{\Omega} \right\rceil$), and the Boolean quantity **PD** which indicates whether the item flow vector $\mathbf{f}$ computed by the algorithm is feasible-path-decomposable (**PD** is equal to 1 if and only if, the corresponding item flow vector $\mathbf{f}$ is feasible-path-decomposable);
- the optimal value **G4** of the PIRP model computed with the objective function (E9), with the Extended-Subtour constraints, and with the Feasible-Path constraints; the time value **T4** corresponding to the running time in seconds required by the computation of **G4**, the estimation **V4** of the number of vehicles used in the solution (obtained from the expression $\left\lceil \frac{\sum_{a\in A} time(a)\cdot F(a)}{\Omega} \right\rceil$), the

quantity **CT4** equals the global number of cuts added by the separation procedure, and the quantity **ND4** equals the number of nodes in the branch-and-cut tree that were visited during the algorithm. Missing values are indicated by a hyphen symbol -, and correspond to PIRP instances that turned out to be infeasible when including the Feasible-Path constraints.

Table 2.3: Numerical results for the PIRP model computed with the objective function (E9), with the Extended-Subtour constraints, and with/without the Feasible-Path constraints.

Id	G3	T3	V3	LB3	PD	G4	T4	V4	ND4	CT4
1	2109.12	0.47	3	1323.04	0	2110.85	1.61	3	808	13
2	1290.09	0.08	3	971.52	0	-	0.01	-	0	1
3	854.83	0.12	2	546.78	1	854.83	3.25	2	9	3
4	3805.81	0.07	3	3599.04	1	3805.81	0.65	3	22	2
5	2432.86	0.03	3	2064.80	0	-	0.44	-	0	10
6	1532.38	0.03	2	1276.16	1	1532.38	1.09	2	0	1
7	2727.30	0.07	4	2448.12	1	2727.30	0.32	4	35	2
8	15561.30	0.27	17	15163.21	1	15561.30	0.81	17	38	3
9	5612.59	6.71	12	4927.96	0	-	0.01	-	0	1
10	7966.03	0.78	6	7443.35	1	7966.03	12.76	6	479	11
11	1840.17	2.49	3	933.32	1	1840.17	19.33	3	6860	11
12	6976.11	0.14	5	6531.15	1	6976.11	1.17	5	142	1
13	1643.76	0.98	2	1044.28	1	1643.76	124.47	2	80	7
14	2643.62	3.20	3	2230.72	1	2643.62	7.29	3	2464	4
15	17131.00	44.97	22	16286.10	0	17179.00	81.58	22	23364	36
16	4826.24	1.05	8	4257.80	1	4826.24	2.54	8	90	7
17	3227.92	0.60	4	2696.90	0	3272.98	70.22	4	446	13
18	20219.30	1.00	10	19556.76	1	20219.30	63.24	10	52	1
19	5944.23	5.99	7	5175.84	1	5944.23	46.90	7	1564	2
20	6091.24	4.68	4	4885.78	1	6091.24	31.09	4	1986	8

By comparing columns **G2** and **G3**, we see that the Extended-Subtour constraints improve the cost of some solutions ($\mathbf{F}$, $\mathbf{f}$), in such a way that those solutions become closer to the projections of optimal TEN IRP solutions ($\mathbf{H}$, $\mathbf{h}$).

The values in columns **T3** and **T4** show that the separation of the Feasible-Path constraints has a non-negligible impact on the running times of the branch-and-cut algorithm for solving the PIRP model (about 2,200% more on average). Also, the Feasible-Path constraints allow us to assert the infeasibility of instances 2, 5, and 9. Furthermore, by examining the computed solutions with costs **G3** and **G4**, we confirm that in 14 of the 17 remaining feasible instances, the Feasible-Path constraints were already satisfied even though we did not take them explicitly into account.

2.7 Conclusions

In this chapter we have introduced the Item Relocation Problem and, in order to handle it, we have formulated the TEN IRP model which is a 2-commodity flow over a Time-Expanded network.

Because the TEN IRP model is too complex for solving directly with a MILP solver we have proposed a Project-and-Lift approach: first we project the TEN IRP on the original digraph to obtain an easier PIRP model. Then we use the solutions of that model for constructing (i.e., lifting) solutions for the original TEN IRP model.

Because while projecting the TEN IRP model the temporal dimension of the original problem is lost, we have introduced the Extended-Subtour constraints for linking the time horizon and the number of vehicles circulating through a set of non-depot vertices. We have proposed two versions of those constraints and we have shown how they can be separated in polynomial time. As a result we have obtained an improved PIRP model which can be handled efficiently with a branch-and-cut algorithm.

With regards to the lift part, we have presented two Lift problems which aim to construct solutions of the TEN IRP model starting from solutions of the PIRP model. From those problems we have derived the Feasible-Path constraints related to a feasibility property that must be satisfied by the item flow of the PIRP model. We have shown how to handle those constraints by a column generation procedure that involves a Constrained Shortest Path as the pricing problem.

Finally we have tested the branch-and-cut algorithms over a set of 20 problem instances. We have confirmed experimentally the improvement of the costs due to the introduction of the Extended-Subtours constraints and the Feasible-Path constraints, and their impact on the related running times.

The Lift problems that we have introduced are difficult and the probability of getting feasible/good solutions for them depends on the structure of the computed PIRP solutions. For that reason it would be interesting to search for additional constraints for improving the PIRP model.

Until now, we only have introduced two Lift problems but we have not proposed models or algorithms to solve them. That will be the main subject of the following chapter.

Chapter 3

Lifting Projected IRP Solutions

In this chapter we propose models and algorithms for handling the Lift problems that we have introduced in Section 2.5.1.

In Section 3.1 we introduce an auxiliary digraph $Strong(G, F)$ and we use it to set a MILP model for solving the Strong Lift Problem in an exact way. Then we perform some numerical experiments and we show that when this model is feasible, its solution provides us with a solution for the TEN IRP model.

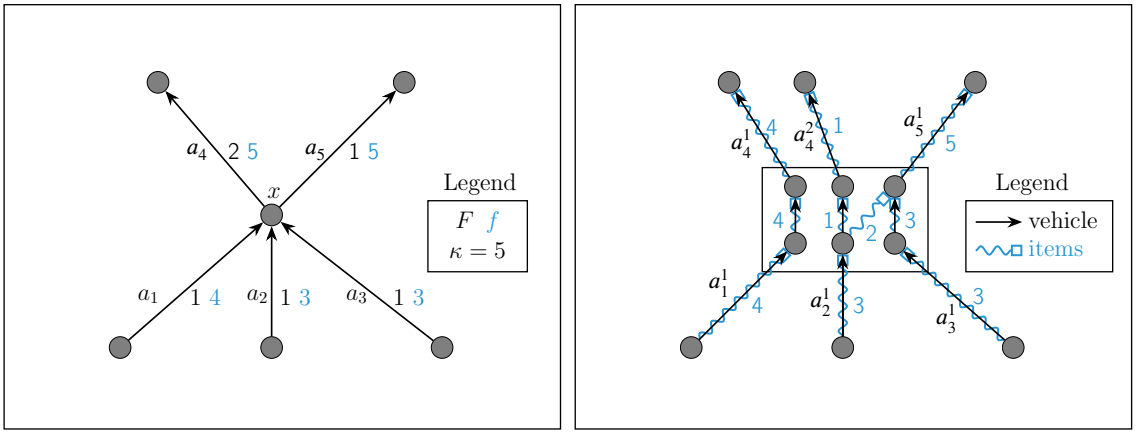
In Section 3.2 we describe a "Weak/Cover" decomposition approach to deal with the Partial Lift Problem. In Section 3.2.1 we start by defining two auxiliary digraphs $Weak(G, f)$ and $Cover(G, f)$, and we use them in Section 3.2.2 for setting three systems of mixed integer linear constraints that are necessary for the Weak /Cover decomposition. In Section 3.2.3, we describe a particular way of implementing the Weak/Cover decomposition and we examine a property that must be verified by the item flow in order to apply this particular implementation. Finally, in Section 3.3 we handle that particular implementation of the Weak/Cover decomposition by using two heuristic algorithms, and we show the results of some numerical experiments for comparing both algorithms.

3.1 A MILP Model for the Strong Lift Problem

Let us recall that the Strong Lift Problem is about the search of an IRP solution $(H,\, h)$ whose projection on the digraph G is exactly the solution $(F,\, f)$ which we obtained through the resolution of the projected PIRP model. So let us consider a feasible (optimal) solution $(F,\, f)$ of the PIRP model. We denote by $S(F)$ the sum $\sum_{a\in A} F(a)$, and by $Q(F)$ the sum $\sum_{x\in X}\Big(\big(\sum_{a\in\partial_G^-(x)} F(a)\big)\cdot\big(\sum_{a\in\partial_G^+(x)} F(a)\big)\Big)$. We are now going to show that it is possible to set a MILP formulation of the Strong Lift

Problem, which involves $2 \cdot Q(F)$ decision variables, together with $Q(F)$ $+3 \cdot S(F)$ integral load and time variables.

The idea is that solving the Strong Lift Problem mainly means determining what happens "inside" the vertices of the digraph G: more precisely, a vertex x being given, we want to know along which arc a' a given vehicle (respectively, a given item) which arrives into x along some arc a is going to leave x, and at which time. Figure 3.1 illustrates the way vehicles and items which arrive into a vertex $x \in X$ along arcs a_1, a_2, and a_3 distribute themselves among the arcs a_4 and a_5 whose origin is x. In order to formalize this idea, we are first going to build an auxiliary digraph $Strong(G, F)$ by "expanding" any vertex x of the digraph G according to the flow value F which arrives into x. Next we shall set a Strong Lift model as a MILP model involving variables indexed on the arcs of this auxiliary digraph.



(a) (b)

Figure 3.1: The Strong Lift Problem. (a) A set of arcs a_1, a_2, a_3, a_4, a_5 that are incident with a common vertex x, together with their corresponding flow values in a PIRP solution (F, f). (b) "Expanding" vertex x makes appear the way vehicles and items distribute themselves between the arcs a_1, a_2, a_3 which arrive into x and the arcs a_4, a_5 which leave x.

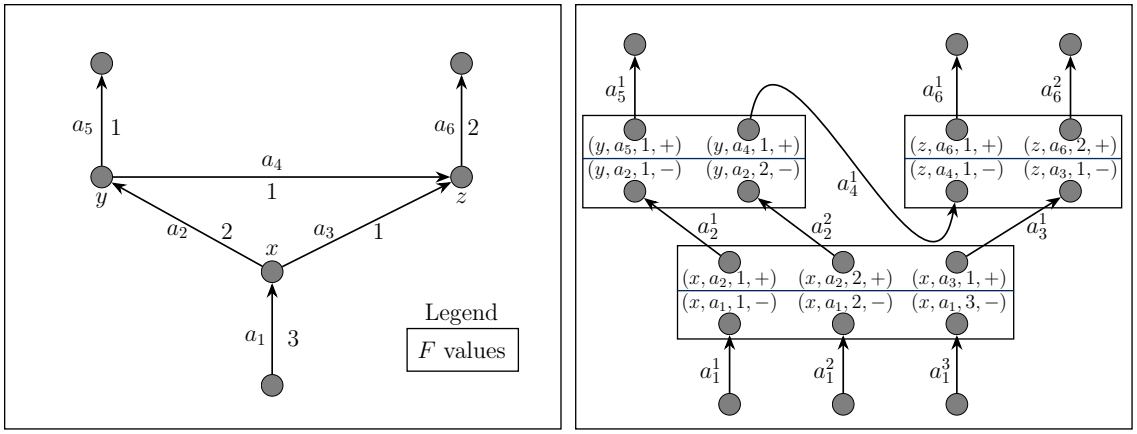
In order to formalize this idea, given a PIRP solution (F, f) on a digraph G we construct the following digraph $Strong(G, F)$.

- With any arc $a = (x, y) \in A$ such that $F(a) > 0$, we associate $F(a)$ *copy-arcs* a^m, $m = 1, \ldots, F(a)$, with respective tail vertices $p = (x, a, m, +)$, and respective head vertices $q = (y, a, m, -)$, a weight $time_{(Strong(G,F))}(a^m) = time_G(a)$, and a weight $cost_{(Strong(G,F))}(a^m) = cost_G(a)$. For any arc a of G such that $F(a) > 0$, we denote by $Copy(a)$ the set of arcs a^m, $m = 1, \ldots, F(a)$. It follows that, at the same time we create those copy-arcs, we also create *copy-vertices* $p = (x, a, m, +)$ and $q = (y, a, m, -)$, which respectively correspond to the vehicles which leave x and to the vehicles which arrive into y. We denote by X^* the resulting vertex set and by $Copy(A)$ the set of all copy-arcs. For any such

vertex $p = (y, a, m, \varepsilon)$, we set $x(p) = y$ and $\varepsilon(p) = \varepsilon$, and, for any vertex y of G, we set:

- $X^*(y) = \{p \in X^* \text{ such that } x(p) = y\}$;
- $X^*Plus(y) = \{p \in X^* \text{ such that } x(p) = y, \varepsilon(p) = +\}$;
- $X^*Minus(y) = \{p \in X^* \text{ such that } x(p) = y, \varepsilon(p) = -\}$;
- $CopyIn(y) = \{a \in Copy(A) \text{ such that } a \text{ has its head in } X^*Minus(y)\}$;
- $CopyOut(y) = \{a \in Copy(A) \text{ such that } a \text{ has its tail in } X^*Plus(y)\}$.

- We complete the arc collection $\{a^m, a = (x, y) \text{ such that } F(a) > 0, m = 1, \ldots, F(a)\}$ by *router-arcs* $u = ((x, a, m, -), (x, a', m', +))$, with a weight $time_{(Strong(G,F))}(u) = 0$, a weight $cost_{(Strong(G,F))}(u) = 0$, and such that for any vertex x of G, they connect any copy-vertex $(x, a, m, -)$ where a has head x, $m = 1, \ldots, F(a)$, to any copy-vertex $(i, a', m', +)$, where a' has tail x, $m' = 1, \ldots F_{a'}$. We denote by $Router$ the set of all router-arcs created this way, and, for any x, we denote by $Router(x)$ the set of the router-arcs u whose tail may be written $(x, a, m, -)$. Notice that $Router(x)$ defines a complete bipartite digraph on the vertices of $X^*(x)$. For any vertex $p = (x, a, m, +)$, we denote by $RouterIn(p)$ the set of router-arcs u whose head is p. Similarly for any vertex $p = (x, a, m, -)$, we denote by $RouterOut(p)$ the set of router-arcs u whose tail is p.



(a) (b)

Figure 3.2: Constructing the digraph $Strong(G, F)$. (a) A set of arcs $a_1, a_2, a_3, a_4, a_5, a_6$ in a digraph G together with their corresponding flow values F of a PIRP solution. (b) The arcs and vertices in the digraph $Strong(G, F)$ that are created from the arcs, vertices, and flow values in (a). To avoid a cumbersome drawing we have not depicted the router-arcs.

We denote by $Strong(G, F)$ the resulting digraph (see Figure 3.2), which contains $2 \cdot S(F)$ vertices, $S(F)$ copy-arcs, and $Q(F)$ router-arcs.

Example 3.1 - An example of a digraph $Strong(G, F)$
Consider the PIRP solution (F, f) over the digraph $G = (X, A)$ that is depicted in Figure 3.3.

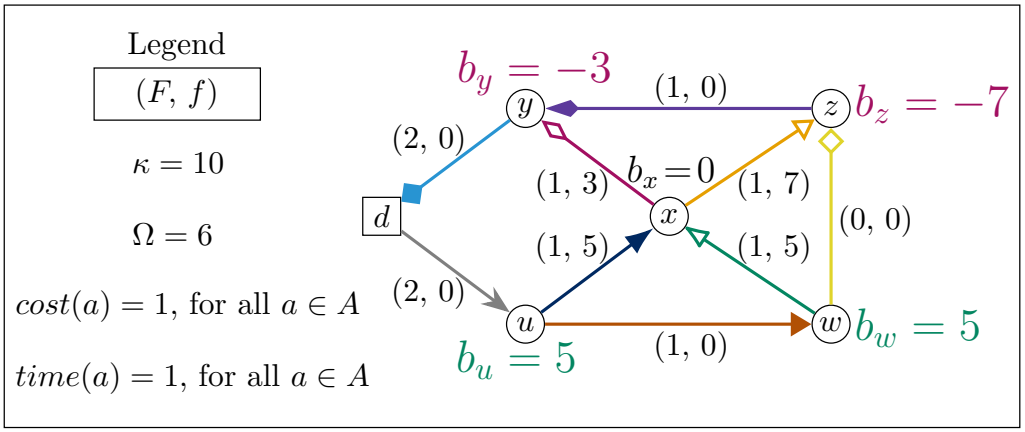


Figure 3.3: An IRP instance on a digraph $G = (X, A)$ together with a PIRP solution (F, f). We have used different arrow tips for distinguishing each arc.

The lift digraph constructed from this solution has 20 vertices and 28 arcs and is shown in Figure 3.4.

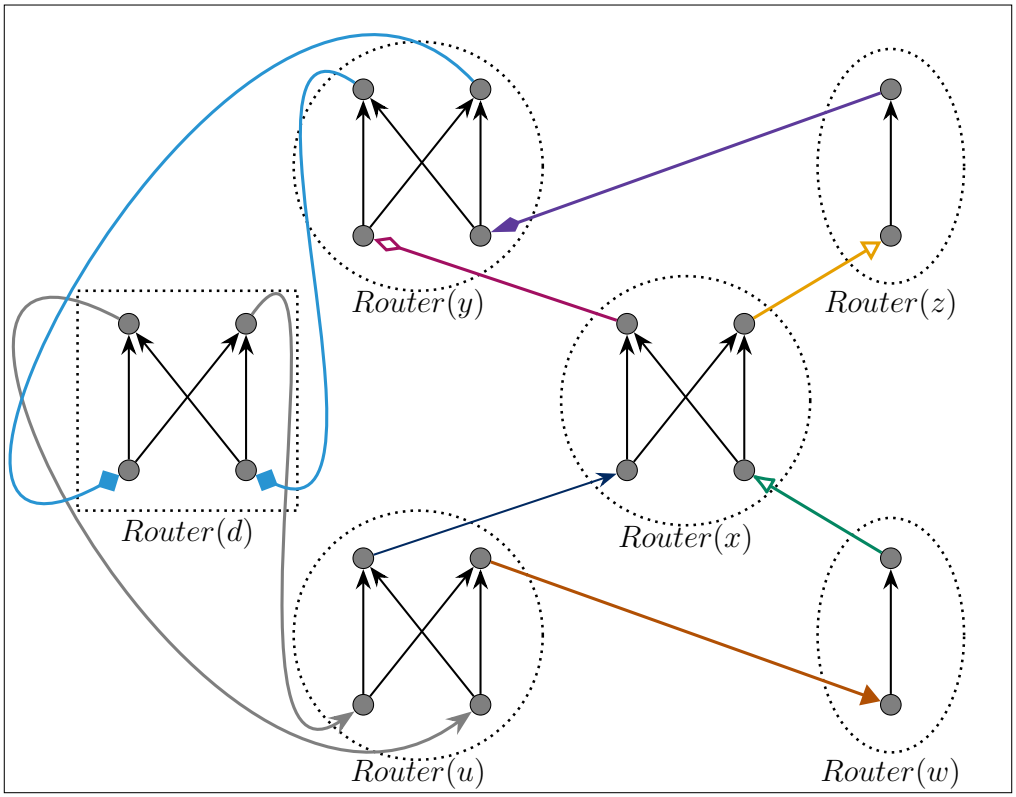


Figure 3.4: The lift digraph constructed from the PIRP solution (F, f) over the digraph $G = (X, A)$ in Figure 3.3. For every vertex $u \in X$, we have surrounded with a dotted convex shape the set of router-arcs in $Router(u)$.

Consider a vertex $x \in V(G)$. The purpose of the digraph $Strong(G, F)$ is to use the router-arcs in $Router(x)$ as "routers" for the vehicles and items entering $X^*(x)$ via $CopyIn(x)$, and going out from $X^*(x)$ via $CopyOut(x)$.

For example, in Figure 3.5, we have used some of those arcs to create a IRP solution, involving a single vehicle route. However, due to the time horizon this is not a feasible solution.

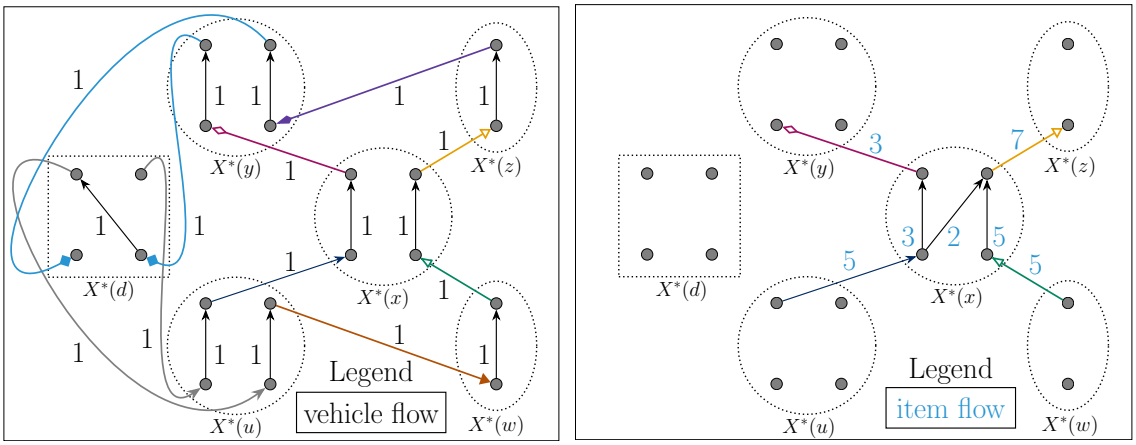


Figure 3.5: Two subgraphs of the lift digraph shown in Figure 3.4. (a) A Hamiltonian path that indicates a single vehicle flow. (b) Item flow.

In contrast, Figure 3.6 shows a feasible solution involving two vehicles. □

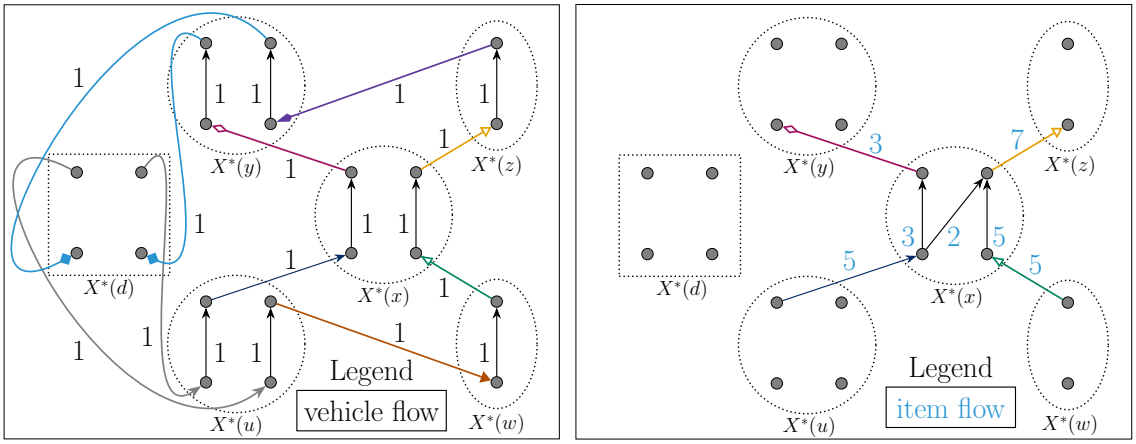


Figure 3.6: Subgraphs of the lift digraph shown in Figure 3.4. (a) Two paths which indicate a vehicle flow involving two vehicles. (b) Item flow.

Now, we can set the following MILP $\mathsf{SLIFT}(G,\, F,\, f)$ Model:

MILP $\mathsf{SLIFT}(G,\, F,\, f)$

Variables

- $\mathbf{Z} = \big(Z_u, u = ((x,a,m,-),(x,a',m',+)) \in \mathit{Router}\big) \in \{0,1\}^{|\mathit{Router}|}$.
- $\mathbf{z} = \big(z_u, u = \big((x,a,m,-),(x,a',m',+)\big) \in \mathit{Router}\big) \in \mathbb{R}_+^{|\mathit{Router}|}$.
- $\boldsymbol{\ell} = \big(\ell_u, u = \big((x,a,m,-),(x,a',m',+)\big) \in \mathit{Router}\big) \in \{0,1\}^{|\mathit{Router}|}$.
- $\mathbf{z}^* = \big(z^*_{a^m}, a^m \in \mathit{Copy}(A)\big) \in \mathbb{R}_+^{|\mathit{Copy}(A)|}$.
- $\mathbf{t} = \big(t_p, p = (x,a,m,\varepsilon) \in X^*\big) \in \mathbb{R}_+^{|X^*|}$.

Constraints

- For any *copy-vertex* $q = (x,a,m,\varepsilon)$ of $\mathit{Strong}(G,\, F)$, with $x \neq d$: $\sum_{u \in \mathit{RouterIn}(q)} Z_u = 1 = \sum_{u \in \mathit{RouterOut}(q)} Z_u$. (E14.1)
- For any *copy-vertex* $q = (d,a,m,+)$ of $\mathit{Strong}(G,\, F)$: $\sum_{u \in \mathit{RouterIn}(q)} Z_u \leq 1$. (E14.2)
- For any *copy-vertex* $p = (d,a,m,-)$ of $\mathit{Strong}(G,\, F)$: $\sum_{u \in \mathit{RouterOut}(p)} Z_u \leq 1$. (E14.3)
- For any *router-arc* u: $z_u \leq \kappa \cdot \ell_u$. (E15.1)
- For any *copy-arc* a^m: $z^*_{a^m} \leq \kappa$. (E15.2)
- For any vertex $p = (x,a,m,-)$ of $\mathit{Strong}(G,\, F)$ with $b_x \geq 0$: $z^*_{a^m} = \sum_{u \in \mathit{RouterOut}(p)} z_u$. (E16.1)
- For any vertex $p = (x,a,m,-)$ of $\mathit{Strong}(G,\, F)$ with $b_x < 0$: $z^*_{a^m} \leq \sum_{u \in \mathit{RouterOut}(p)} z_u$. (E16.2)
- For any vertex $q = (x,a,m,+)$ of $\mathit{Strong}(G,\, F)$ with $b_x \leq 0$: $z^*_{a^m} = \sum_{u \in \mathit{RouterOut}(q)} z_u$. (E17.1)
- For any vertex $q = (x,a,m,+)$ of $\mathit{Strong}(G,\, F)$ with $b_x > 0$: $z^*_{a^m} \geq \sum_{u \in \mathit{RouterOut}(q)} z_u$. (E17.2)
- For any arc $a = (x,y)$ of G: $\sum_{u \in \mathit{Copy}(a)} z^*_u = f(a)$. (E18)

- For any *copy-arc* $(p,q) = \big((x, a = (x,y), m, +), (y, a = (x,y), m, -)\big)$: $t_q \geq t_p + time_G(x,y)$. (E19)
- For any *router-arc* $u = (p = (x,a,m,-), q = (x,a',m',+))$, the implication $\big((Z_u = 1) \vee (\ell_u = 1)\big) \Rightarrow t_p \geq t_q$ holds, and it can be represented by the two linear constraints: $\Omega \cdot Z_u + t_p - t_q \leq \Omega$ and $\Omega \cdot \ell_u + t_p - t_q \leq \Omega$. (E20)

Objective function

Maximize $\sum_{u \in Router(d)} Z_u$. (E21)

The meaning of the above variables is as follows.

- For $u = \big((x,a,m,-),(x,a',m',+)\big) \in Router$:
  - the value $Z_u = 1$ means that the vehicle which arrives at vertex x along the *copy-arc* a^m keeps on along the copy-arc $a'^{m'}$;
  - the value z_u means the number of items which arrive at vertex x along the copy-arc a^m and which are transferred to the copy-arc $a'^{m'}$;
  - $\ell_u = 0$ indicates that $z_u = 0$.
- For $a^m \in Copy(A)$, the value z_{a^m} means the number of items transported along arc a^m.
- For $p = (x,a,m,\varepsilon) \in X^*$ the value t_p means the time when a vehicle arrives (in case $\varepsilon = -$) or leaves (in case $\varepsilon = +$) in x along arc a^m.

Now we explain the meaning of the above constraints (E14.1)-(E21).

- Constraint (E14.1) means that for every $q = (x,a,m,+)$ with $x \neq d$ exactly one vehicle must enter q through a router-arc in $RouterIn(q)$. Symmetrically, for every $q = (x,a,m,-)$ with $x \neq d$ exactly one vehicle must leave q through a router-arc in $RouterOut(q)$.
- Constraint (E14.2) means that if $q = (d,a,m,+)$ (i.e., $q \in X^*Plus(d)$ with d the depot vertex) then at most one vehicle can enter q through a router-arc in $RouterIn(q)$. The case $\sum_{u \in RouterIn(q)} Z_u = 0$ means that q is the starting vertex of a vehicle route, and the case $\sum_{u \in RouterIn(q)} Z_u = 1$ means that q is an intermediate vertex of a vehicle route.

- Constraint (E14.3) means that if $p = (d, a, m, -)$ (i.e., $q \in X^*Minus(d)$ with d the depot vertex) then at most one vehicle can leave p through a router-arc in $RouterOut(p)$. The case $\sum_{u \in Out(p)} Z_u = 0$ means that p is the ending vertex of a vehicle route, and the case $\sum_{u \in Out(p)} Z_u = 1$ means that p is an intermediate vertex of a vehicle route.

- For any router-arc u, the constraint (E15.1) models the logical implications: if $\ell_u = 0$ then $z_u = 0$ (i.e., if $\ell_u = 0$ then the router-arc u cannot transport any item), and if $\ell_u = 1$ then $z_u \leq \kappa$ (i.e., if $\ell_u = 1$ then the router-arc u cannot transport more than κ items).

- Constraint (E15.2) means that every copy-arc a^m cannot transport more than κ items.

- Constraint (E16.1) means that for every $p = (x, a, m, -)$ which is a copy of a neutral or an excess vertex $x \in X$, we have the number of items entering p through a^m is equal to the number of items leaving p through the router-arcs in $RouterOut(p)$ (see Figure 3.7).

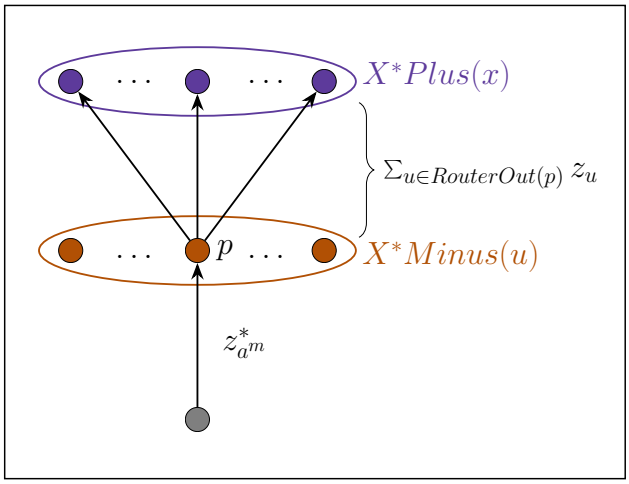


Figure 3.7: Interpretation of the SLIFT(G, F,f) constraints (E16.1). For all $p = (x, a, m, -) \in X^*Minus(x)$ with $x \in X$ and $b_x \geq 0$, the number of items entering p through the copy-arc a^m is equal to the number of items leaving p through the router-arcs in $RouterOut(p)$.

- Constraint (E16.2) means that for every $p = (x, a, m, -)$ which is a copy of a deficit vertex $x \in X$, we have the number of items entering p through a^m is less than or equal to the number of items leaving p through the router-arcs in $RouterOut(p)$. Note the "less than or equal to" symbol of this constraint. That is because we are considering the demand of b_x items in vertex x is satisfied by taking items from vehicles only when they arrive to a vertex in $X^*Minus(x)$ (see Figure 3.8). As a consequence, the total number of items

entering to a vertex $p \in X^*Minus(x)$ is greater than or equal to the total number of items going out from p.

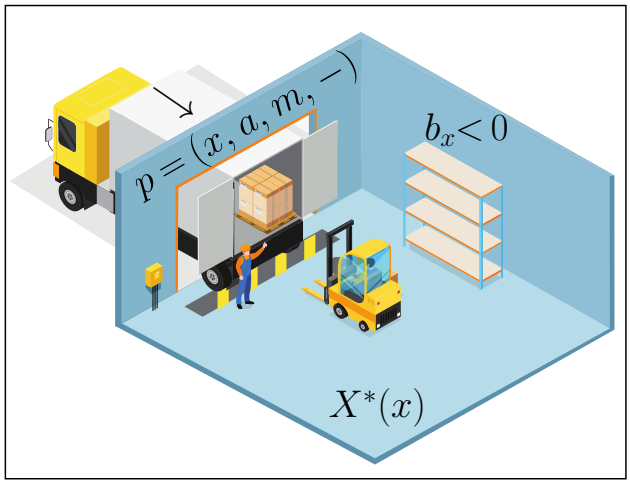


Figure 3.8: Interpretation of the SLIFT(G, F, f) constraints (E16.2). In a set of copy-vertices $X^*(x)$ of a deficit vertex x, the items for satisfying the demand b_x can be taken from the vehicles only when they arrive to some copy-vertex $p = (x, a, m, -) \in X^*Minus(x)$

- Constraint (E17.1) means that for every $q = (x, a, m, +)$ which is a copy of a neutral or a deficit vertex $x \in X$, we have the number of items entering through the router-arcs in $RouterIn(q)$ is equal to the number of items leaving q through the copy-arc a^m (see Figure 3.9).

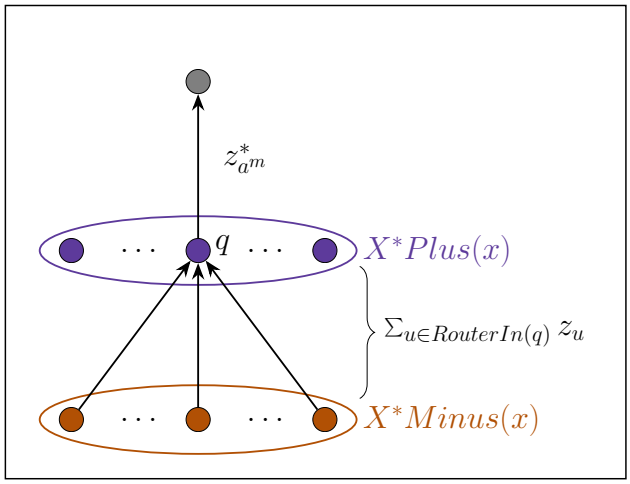


Figure 3.9: Interpretation of the SLIFT(G, F, f) constraints (E17.1). For all $q = (x, a, m, +) \in X^*Plus(x)$ with $x \in X$ and $b_x \leq 0$, the number of items entering through the router-arcs in $RouterOut(p)$ is equal to the number of items leaving p through the copy-arc a^m.

- Constraint (E17.2) means that for every $q = (x, a, m, -)$ which is a copy of an excess vertex $x \in X$, we have the number of items leaving q through a^m is

greater than or equal to the number of items entering q through the router-arcs in $RouterIn(q)$. Note the "greater than or equal to" symbol of this constraint. That is because we are considering the surplus objects in vertex x can be assigned to vehicles only when they arrive to a vertex in $X^*Plus(x)$ (see Figure 3.10). As a consequence, the total number of items entering to a vertex $q \in X^*Plus(x)$ is greater than or equal to the total number of items going out from q.

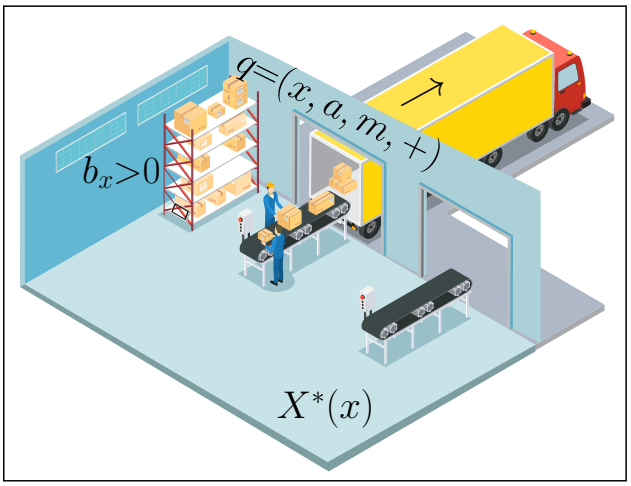


Figure 3.10: Interpretation of the SLIFT(G, F,f) constraints (E17.2). In a set of copy-vertices $X^*(x)$ of an excess vertex x, the surplus items can be assigned to vehicles only when they arrive to some copy-vertex $q = (x, a, m, +) \in X^*Plus(x)$

- Constraint (E18) means that the total quantity of items traversing the copy-arcs in $Copy(a)$ is equal to the number of items traversing a in the PIRP solution (F, f).

- Constraint (E19) means that for any copy-arc $u = (p, q)$ with $p \in X^*Plus(x)$ and $q \in X^*Minus(y)$ the departure time for vertex q is at least the departure time for vertex p plus the time necessary for traversing the copy-arc u (i.e., $time_G(x, y)$).

- In the case of constraints (E20) we have that $time_{Strong(G,F)}(a) = 0$. So, the inequality $\Omega \cdot Z_u + t_p - t_q \leq \Omega$ models the logical implication $(Z_a = 1) \Rightarrow (t_x \leq time_y)$, and the inequality $\Omega \cdot \ell_u + t_p - t_q \leq \Omega$ models the logical implication $(\ell_a = 1) \Rightarrow (t_x \leq t_y)$. Hence, if an arc (x, y) has vehicles or items traversing through it, the departure time of y is at least the departure time of x.

- The objective function (E21) aims to maximize the number of times that the vehicles use the depot vertex as an intermediate vertex of their routes. This reduces the number of vehicles that leaves the depot vertex for the first time, and as a consequence we minimize the number of vehicles used by the solution.

We have the following result about the $\mathsf{SLIFT}(G,\ F,\ f)$ MILP model.

Theorem 3.1 - $\mathsf{SLIFT}(G,\ F,\ f)$ solves the Strong Lift Problem
Solving the above MILP model $\mathsf{SLIFT}(G,\ F,f)$*, which involves* $2 \cdot Q(F)$ *decision variables* Z *and* $\boldsymbol{\ell}$*, together with* $Q(F) + 3 \cdot S(F)$ *rational load and time variables* z*,* z^* *and* t*, also solves the Strong Lift Problem related to* (F,f) *in an exact way.*

Proof. Since the Strong Lift Problem explicitly requires the projection of H onto the digraph G to be equal to F, we see that the routes followed by the vehicles are completely determined by the way we assign a vehicle entering into a vertex x along some copy-arc a^m onto another copy-arc $a'^{m'}$ leaving x (in case $x = d$, we may assign a "null" arc, that means consider that the vehicle ends its trip into d with the arc a^m). Decision vector $\mathbf{Z}$, together with matching constraints (E14.1 - E14.3) express the way vehicle routes distribute themselves inside any vertex x. Then we see that, since any true item move from x to y must be covered by some vehicle, any item arriving to some vertex x_i along some copy-arc a^m will have either to remain in x as part of the negative deficit b_x or keep on along another copy-arc $a'^{m'}$ leaving x. Constraints (E16.1 - E18) express the way items are going to distribute themselves while traversing this vertex x. Deriving a IRP solution $(H,\ h)$ from a PIRP solution $(F,\ f)$ and from vectors $\mathbf{Z}$, $\boldsymbol{\ell}$, $\mathbf{z}$, $\mathbf{z}^*$, that means from such an accurate description of the routes followed by the vehicles and the items, becomes possible if we are able to embed the vertices of the $Strong(F,\ f)$ digraph into the Time-Expanded network G^{Ω}, that means if we can compute a vector $\mathbf{t}$ which meets constraints (E19, E20). It follows that any feasible solution of the Strong Lift Problem may be turned into a feasible solution of $\mathsf{SLIFT}(G,\ F,\ f)$ and conversely. We conclude by noticing that the value of the objective function $\sum_{u \in Router(d)} Z_u$ is merely the difference between the value $\sum_{a \in \partial^+_G(d)} F(a)$ and the number of vehicles, while the other components of the cost IRP function are the same for $(H,\ h)$ and $(F,\ f)$. It comes that solving $\mathsf{SLIFT}(G,\ F,\ f)$ makes us minimize the number of vehicles involved into the lifted solution $(H,\ h)$ derived from $(F,\ f)$. ■

Remark. Though above results may look trivial, they are not. They derive from the fact that we require the projection of H onto digraph G to be exactly equal to F.

Numerical Behavior of SLIFT(G, F, f)

What we want to observe here is the probability that a projected solution (F, f) may be lifted, the related increase in the number of vehicles with respect to the estimation involved into the PIRP model, and also the running times related to the MILP SLIFT(G, F, f) model.

Table 3.1 displays the numerical results of solving the SLIFT(G, F, f) MILP, over the instances of Table 2.1. We have modified the time horizon values of instances 2, 5, and 9, in order to make them feasible. Again, column **G4** corresponds to the optimal value (with respect to the objective function (E9)) of the PIRP solution found by the branch-and-cut algorithm that incorporates the separation procedures described in Algorithms 4-5, and the column **V4** is the estimated number of vehicles in the solution with cost **G4**. The column **SLIFT** displays the cost of the solution of the SLIFT(G, F, f) MILP, **TSLIFT** indicates the running time in seconds that was spent in the computation of **SLIFT**, and **VSLIFT** is the number of vehicles used in the solution with cost **SLIFT**. Missing values are indicated by a hyphen symbol -, and correspond to PIRP solutions (F, f) for which the corresponding SLIFT(G, F, f) MILP is infeasible.

Table 3.1: Strong Lift Numerical Results.

Id	n	m	κ	Ω	λ	α	β	γ	G4	V4	SLIFT	TSLIFT	VSLIFT
1	20	78	2	324	4	304	1.0	1.000	2110.85	3	-	0.01	-
2	20	65	5	400	5	150	0.4	0.500	1196.10	3	-	0.01	-
3	20	77	10	440	4	328	0.2	0.250	854.83	2	-	0.01	-
4	20	75	2	680	8	328	1.0	1.000	3805.81	3	-	6.19	-
5	20	50	5	603	9	392	0.4	0.250	2354.43	3	2474.70	0.01	3
6	20	57	10	840	8	376	0.2	0.250	1532.38	2	-	0.07	-
7	20	62	5	420	6	300	0.4	0.500	2727.30	4	-	0.01	-
8	50	163	2	460	4	170	1.0	1.000	15561.30	17	-	0.56	-
9	50	155	5	390	6	196	0.4	0.500	4326.10	7	-	0.01	-
10	50	149	10	440	4	164	1.0	0.500	7966.03	6	-	0.01	-
11	50	146	20	436	4	312	0.1	0.125	1840.17	3	-	0.01	-
12	50	175	2	728	8	268	1.0	1.000	6976.11	5	-	700.00	-
13	50	217	5	912	8	672	0.4	0.250	1643.76	2	2153.65	0.14	2
14	50	154	10	1040	8	416	0.2	0.125	2643.76	3	-	8.08	-
15	100	363	2	336	4	252	1.0	1.000	17179.00	22	-	0.01	-
16	100	236	5	516	4	188	0.4	0.250	4826.24	8	-	17.38	-
17	100	289	10	432	4	360	0.2	0.250	3272.98	4	-	0.01	-
18	100	419	2	1032	8	412	1.0	1.000	20219.30	10	-	34226.28	-
19	100	327	5	552	8	392	0.5	0.200	5944.23	7	-	1630.87	-
20	100	313	10	712	8	312	0.5	0.500	6091.24	4	-	2.46	-

Comments. Unfortunately, the numerical experiments from Table 3.1 show that, even when f is feasible-path-decomposable, SLIFT(G, F, f) scarcely admits a feasible solution. So we are now going to focus on the Partial Lift Problem.

3.2 Dealing with the Partial Lift Problem

Setting a MILP model for the Partial Lift Problem is not as easy as for the Strong Lift Problem, since we cannot say that the arcs involved into a PIRP solution (F, f) will be enough to ensure the existence of a "lifted" solution (H, h). So we are going to try a decomposition approach which relies on the idea that the quality of final solution (H, h) is mostly driven by the way items are routed and scheduled, and that an algorithmic resolution might work as an iterative loop whose every iteration could be decomposed into two steps as follows.

1st step: "Lift" item flow f into an item flow h on the TEN G^{Ω}, without taking care of the flow F. This initial "Weak-Lift" process is going to rely on the construction of a specific $Weak(G, f)$ digraph, whose construction is going to follow the same logic as the construction of the $Strong(G, F)$ digraph.

2nd step: Once h has been fixed, extend h into a good (best) solution IRP (H, h). Performing this step is just going to be a matter of solving a Minimum Cost Flow problem on an ad hoc digraph. This ad hoc digraph is going to be a subgraph of some $Cover(G, f)$ digraph which will be defined later, and whose construction will allow us to resolve those Minimum Cost Flow problems in a flexible way.

Linking 1st step and 2nd step: A key point is that the quality of resulting solution deeply depends not only on the route followed by the items, but also on the way they are scheduled, that means on the time values of the vertices and arcs which are going to support h in G^{Ω}. So we provide both the digraph $Weak(G, f)$ involved into the 1st step and the digraph $Cover(G, f)$ involved into the 2nd step with some kind of flexibility, which allows to delay the instantiation of the time variables related to the arcs and which support h and to make them evolve until a satisfactory solution is obtained. This flexibility device takes the form of a collection Λ of arcs common to both $Weak(G, f)$ and $Cover(G, f)$, which commands a set of constraints imposed to those time values. This arc set Λ becomes the master object of a bi-level *Weak/Cover* decomposition scheme, which we describe now into details, while starting by the construction of the two digraphs $Weak(G, f)$ and $Cover(G, f)$.

3.2.1 The Digraphs $Weak(G,\ f)$ and $Cover(G,\ f)$

The digraph $Weak(G,\ f)$ aims at providing us with a description of the way items traverse the vertices x of the digraph G. Its construction proceeds the same way as for the digraph $Strong(G,\ f)$.

Vertices of $Weak(G,\ f)$

With any arc $a = (x, y) \in A$ such that $f(a) \geq 1$, we associate $\left\lceil \frac{f(a)}{\kappa} \right\rceil$ *copy-arcs* $a^m, m = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil$, with tail $p = (x, a, m, +)$, head $q = (y, a, m, -)$, a weight $time_{(Weak(G,F))}(a^m) = time_G(a)$, and a weight $cost_{Weak(G,F)}(a^m) = cost_G(a)$, respectively; and we denote by $Copy(a)$ the set of those arcs $a^m, m = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil$. At the same time we create those copy-arcs, we also create *copy-vertices* $p = (x, a, m, +)$ and $q = (y, a, m, -)$, which respectively correspond to the vehicles which leave x with a non-null load and to the vehicles which arrive into y with a non-null load. Like in the case of the digraph $Strong(G,\ F)$, we denote by X^* resulting vertex set which becomes the vertex set of $Weak(G,\ f)$. Also we denote by $Copy(A)$ the set of all those copy-arcs. For any vertex $p = (y, a, m, \varepsilon)$ of X^*, we set $x(p) = y$ and $\varepsilon(p) = \varepsilon$. Also, for any vertex y of G, we define the following subsets of X^*:

- $X^*(y) = \{p \in X^* \text{ such that } x(p) = y\}$;
- $X^*Plus(y) = \{\text{vertices } p,\ x(p) = y,\ \text{and } \varepsilon(p) = +\}$;
- $X^*Minus(y) = \{p \in X^* \text{ such that } x(p) = y,\ \text{and } \varepsilon(p) = -\}$;

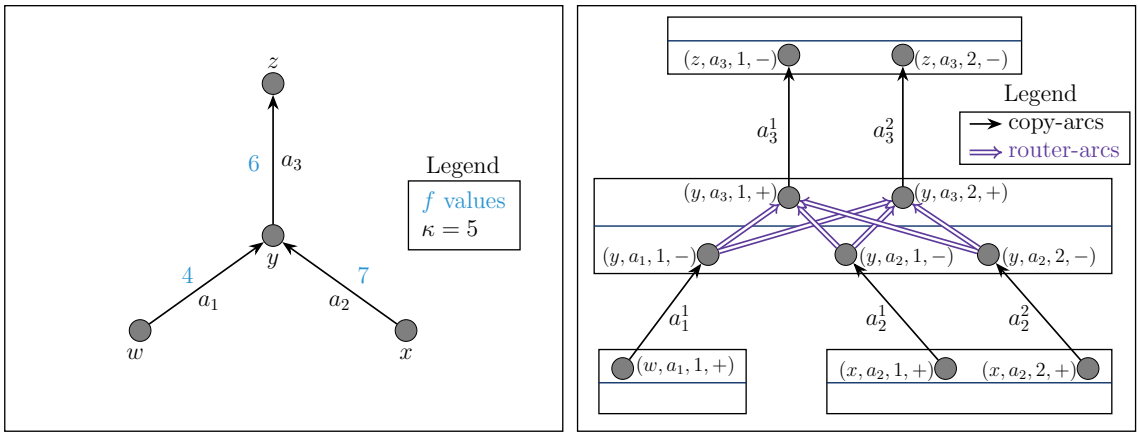
Arcs of $Weak(G,\ f)$

We complete the arc collection $\left\{a^m, a = (x, y),\ \text{such that } f(a) \geq 1, m = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil\right\}$ by *router-arcs* which, for any vertex x of G, connect copy vertex $(x, a, m, -)$ with a arriving into x, $m = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil$, to copy vertex $(x, a', m', +)$ with a' starting from x, $m' = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil$. Once again we denote by $Router$ the set of all router-arcs created this way, and, for any x, we denote by $Router(x)$ the set of all router-arcs u whose tail can be written $(x, a, m, -)$. Notice that $Router(x)$ defines a complete bipartite digraph on the vertices of $X^*(x)$. For any vertex $p = (x, a, m, -)$, we denote by $RouterOut(p)$ the set of router-arcs u whose tail is p. Similarly, for any vertex $q = (x, a, m, +)$, we denote by $RouterIn(q)$ the set of router-arcs u whose head is q.

Finally, for any vertex y of G, we set:

- $CopyIn(y) = \{a \in Copy(A)$ such that a has its head in $X^* Minus(y)\}$;
- $CopyOut(y) = \{a \in Copy(A)$ such that a has its tail in $X^* Plus(y)\}$.

We denote by $Weak(G, f)$ resulting digraph (see Figure 3.11). This digraph can be seen as a subgraph of $Strong(G, F)$, since every vertex (respectively, every arc) of $Weak(G, f)$ is also a vertex (respectively, is also an arc) of $Strong(G, F)$. Furthermore, it does not contain more than $2 \cdot S(F)$ copy-vertices, $S(F)$ copy-arcs, and $Q(F)$ router-arcs.



(a) (b)

Figure 3.11: Constructing the digraph $Weak(G, f)$. (a) Three arcs a_1, a_2, a_3 which are incident with a common vertex y in a digraph $G = (X, A)$ together with their corresponding f values in a PIRP solution (F,f). (b) All the arcs and vertices of $Weak(G, f)$ that are constructed from the vertices, arcs, and f values in (a).

Lemma 3.1. *The digraph $Weak(G, f)$ is acyclic.*

Proof. It directly comes from the fact that if (F, f) is an optimal solution of the PIRP model, then the subgraph induced by the arcs $a = (x, y)$ of G such that $f(a) > 0$ does not contain any circuit. ■

The digraph $Cover(G, f)$ aims at providing us with a flexible reduced version of the Time-Expanded network G^Ω and at helping us in finding the flows H and h. Its construction proceeds as follows.

Vertices of $Cover(G, f)$

They are all the vertices of $Weak(G, f)$, augmented with an auxiliary vertex *source* and an auxiliary vertex *sink*. We denote by $V(Cover)$ the vertex set of $Cover(G, f)$.

Arcs of $Cover(G,\ f)$

They are:

- one arc $u = (sink,\ source)$, provided with a weight $cost_{Cover(G,f)}(u) = \alpha$;
- any arc $u = \big(source,\ p = (x, a, m, \varepsilon)\big)$, $p \in X^*$, provided with a weight $cost_{Cover(G,f)}(u)$ equal to the *cost* weight (i.e., the weight induced by $cost : A \to \mathbb{R}_+$) of a minimum *time* weight path (i.e., minimum with respect to the weight $time : A \to \mathbb{R}_+$) from d to x;
- any arc $u = (p = (x, a, m, \varepsilon), sink)$, $p \in X^*$, provided with a weight $cost_{Cover(G,f)}(u)$ equal to the *cost* weight of a minimum *time* weight path from x to d;
- any arc $u = \big(p = (x, a, m, \varepsilon),\ q = (y, a', m', \varepsilon')\big)$, $p, q \in X^*$, provided with a weight $cost_{Cover(G,f)}(u)$ equal to the *cost* weight of a minimum *time* weight path from x to y. Among those arcs, those which belong to the set $Copy(A)$ of arcs $a^m, a \in A$, $m = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil$, we say they are *marked*.

We denote by $A(Cover)$ the arc set of the digraph $Cover(G,\ f)$.

Those two digraphs allow us to describe the following *Weak/Cover* Decomposition Scheme.

3.2.2 The *Weak/Cover* Decomposition Scheme

As in the Strong Lift Problem, the way item flow values $f(a)$, $a \in \partial_G^-(x)$ will distribute themselves into values $f(a')$, $a' \in \partial_G^+(x)$ while moving through vertex x, is going to be described by two real vectors $\mathbf{z} = \big(z_u,\ u = \big((x, a, m, -), (x, a', m', +)\big) \in Router\big)$, and $\mathbf{z}^* = \big(z^*_{a^m},\ a^m \in Copy(A)\big)$. Those two vectors will provide us with "lifted" flow h once we assign time values t_p to the vertices p of the digraph $Weak(G,\ f)$. But conversely, the way we assign those time values is going to have an impact on the computation of vectors $\mathbf{z}$, and $\mathbf{z}^*$. So, as told at the beginning of Section 3.2, we consider as the master object of our decomposition scheme a set Λ of arcs (p, q), $p \in X^* \cup \{source,\ sink\} = V(Cover)$. This active arc subset of $A(Cover)$ will provide us with the arcs which can be used for item transfer or vehicle moves. It must contain all the marked arcs, and induce the following constraints.

- On vector $\mathbf{Z}$ which will describe vehicle routes on the digraph $Cover(G,\ f)$: for every arc $u = (p, q)$ which is not in Λ, we set $Z_u = 0$.

- On vector $\mathbf{z}$: for every router-arc $u = (p, q)$ which is not in Λ, we set $z_u = 0$;
- On time vector $\mathbf{t} = (t_p, p \in V(Cover))$: for every arc (p, q) in Λ, we have that $t_q \geq t_p + time_G(x(p), x(q))$.

That means that once Λ has been determined, vector $\mathbf{t}$ becomes constrained by the following linear programming constraint system $\mathsf{CTIME}(\Lambda)$.

CTIME(Λ) constraint system on the time vector $\mathbf{t} = (t_p,\ p \in V(Cover)) \geq 0$:

- $t_{source} = 0$; (E22)
- $t_{sink} \leq \Omega$; (E23)
- for any arc $u = (source, p)$, $p \in X^*$: $t_p \geq time_G(d, x(p))$; (E24)
- for any arc $u = (p, sink)$, $p \in X^*$: $t_p + time_G(x(p), d) \leq \Omega$; (E25)
- for any arc (p, q) in Λ, $p \in X^*, q \in X^*$: $t_q \geq t_p + time_G(x(p), x(q))$. (E26)

By the same way, vectors $(\mathbf{z},\ \mathbf{z}^*)$ become constrained by the following constraint system $\mathsf{WEAK}(G,\ f,\ \Lambda)$, which is nothing more than a linear program.

WEAK$(G,\ f,\ \Lambda)$ constraint system on vectors $\mathbf{z}$, $\mathbf{z}^*$:

- for any copy-arc a^m: $z^*_{a^m} \leq \kappa$; (E27)
- for any copy-vertex $q = (x, a, m, -)$ of $Weak(G,\ f)$:
  $z^*_{a^m} \leq \sum_{u \in RouterOut(q)} z_u$; (E28)
- for any copy-vertex $p = (x, a, m, +)$ of $Weak(G,\ f)$:
  $z^*_{a^m} \geq \sum_{u \in RouterIn(p)} z_u$; (E29)
- for any vertex x of G:
  $\sum_{u \in CopyIn(x)} z^*_u = \sum_{u \in RouterOut(x)} z_u + \max(-b_x, 0)$; (E30)
- for any vertex x of G:
  $\sum_{u \in CopyOut(x)} z^*_u = \sum_{u \in RouterIn(x)} z_u + \max(b_x, 0)$; (E31)
- for any router-arc u which is not in Λ: $z_u = 0$. (E32)

Finally, the flow vector $\mathbf{Z}$ with indexation on the arcs of $Cover(G, f)$ and which is going to represent the activity of the vehicles becomes constrained by the following constraint system $\mathsf{COVER}(G, f, \Lambda)$, which is nothing more than a standard Minimum Cost Flow LP model.

$\mathsf{COVER}(G, f, \Lambda)$ constraint system on vector $\mathbf{Z}$:

- for every vertex p of $Cover(G, f)$: (E33)

$$\sum_{u \in \partial^-_{Cover(G,f)}(p)} Z_u = \sum_{u \in \partial^+_{Cover(G,f)}(p)} Z_u;$$

- for any marked arc u: $Z_u = 1$; (E34)
- for any unmarked arc $u \in X^* \times X^*$ which is not in Λ: $Z_u = 0$; (E35)
- the cost (E36)

$$\sum_{u \in A(Cover)} cost_{Cover(G,f)}(u) \cdot Z_u$$

  is the smallest possible.

In the rest of this chapter, we will denote by $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ an optimal dual solution of $\mathsf{COVER}(G, f, \Lambda)$, with vector $\boldsymbol{\lambda} = \big(\lambda_p, p \in V(Cover)\big)$ corresponding to constraints (E33) and vector $\boldsymbol{\mu} = \big(\mu_u, u \text{ is a marked arc of } A(Cover)\big)$ corresponding to constraints (E34).

So, we may now reformulate our Partial Lift Problem according to the following bi-level setting.

Weak/Cover Reformulation of the Partial Lift Problem: Compute arc collection $\Lambda \subseteq A(Cover)$ which contains the marked arcs and is such that

- $\mathsf{CTIME}(\Lambda)$ admits a feasible solution;
- $\mathsf{WEAK}(G, f, \Lambda)$ system admits a feasible solution $(\mathbf{z}, \mathbf{z}^*)$;
- the optimal value $\sum_{u \in A(Cover)} cost_{Cover(G,f)}(u) \cdot Z_u$ of $\mathsf{COVER}(G, f, \Lambda)$ is the smallest possible.

3.2.3 Weak-Lift-Consistency

Before keeping on, we must ask ourselves about which kind of restriction are imposed to the solutions computed according to above *Weak/Cover* decomposition scheme.

We say that vector h is *weak-lift-consistent* with f if the projection of h onto G is equal to f, and for any arc $a = (x, y)$ of G we have that, the transportation of $f(a)$ items from x to y can be decomposed into the transportation along m arcs $\big((x, t^m), (y, t^m + time_G(x, y)\big)$, $m = 1, \ldots, \left\lceil \frac{f(a)}{\kappa} \right\rceil$ of the Time-Expanded network G^Ω (some of them possibly identical), and each one transporting no more than κ items.

Then we have the following result.

Theorem 3.2 - Optimal Weak-Lift-Consistent solutions
Solving above Weak/Cover Reformulation yields the best feasible weak-lift-consistent solution of the Partial Lift Problem.

Proof. Let us first consider some feasible weak-lift-consistent solution (H, h) of the Partial Lift Problem. We split any arc $u = ((x, t), (y, t+time_G(x, y))$ of G^Ω such that related value $h(u)$ is larger than κ into $\left\lceil \frac{h(u)}{\kappa} \right\rceil$ arcs with pairwise distinct vertices.

Then we consider the *skeleton* digraph $Sk(G^\Omega, H, h)$ obtained from G^Ω by keeping only the arcs $u = \big((x, t), (y, t + time_G(x, y)\big)$ and vertices (x, t) obtained this way, by adding an arc between any two consecutive vertices (x, t) and (x, t'), $t \leq t'$, and by adding vertices *source* and *sink* together with arc $(sink, source)$ and all arcs $(source, (x, t))$ and $((x, t), sink)$. Since (H, h) is weak-lift-consistent we can say that, for any arc $a = (x, y)$ in G, no more than $\left\lceil \frac{h(u)}{\kappa} \right\rceil$ arcs $u = \big((x, t), (y, t + time_G(x, y)\big)$ are involved into $Sk(G^\Omega, H, h)$. It comes that we may associate in a bijective way, with any vertex (x, t) of $Sk(G^\Omega, H, h)$, a vertex $p = (x, a, m, \varepsilon)$ of $Weak(G, f)$ and extend this correspondence to vertices *source* and *sink*. For any vertex $p = (x, a, m, \varepsilon)$ obtained through this construction, we set $x(p) = x$ and $t(p) = t$. We define the arc collection Λ as $\Lambda = \{(p, q)$ such that $t(q) - t(p) \geq time_G(x(p), x(q))\}$. Then we easily check that h may be turned into a feasible solution of $\mathsf{WEAK}(G, f, \Lambda)$, that H may be turned into a feasible solution of $\mathsf{COVER}(G, f, \Lambda)$ with unchanged cost value, and that values $t_p = t(p)$ meet the constraints of $\mathsf{CTIME}(\Lambda)$.

Conversely, let us consider a solution $(\Lambda, \mathbf{z}, \mathbf{z}^*, \mathbf{Z})$ of above Weak/Cover Decomposition Scheme, together with some feasible solution $\mathbf{t}$ of the $\mathsf{CTIME}(\Lambda)$ constraint system. We see that $\mathbf{t}$ may be chosen in such a way that, for any arc marked arc (p, q) in Λ, we get $t_q = t_p + time_G(x(p), x(q))$. Then we perform the following construction of a digraph $Aux(G, f, \mathbf{t})$. This construction is illustrated in Figure 3.12.

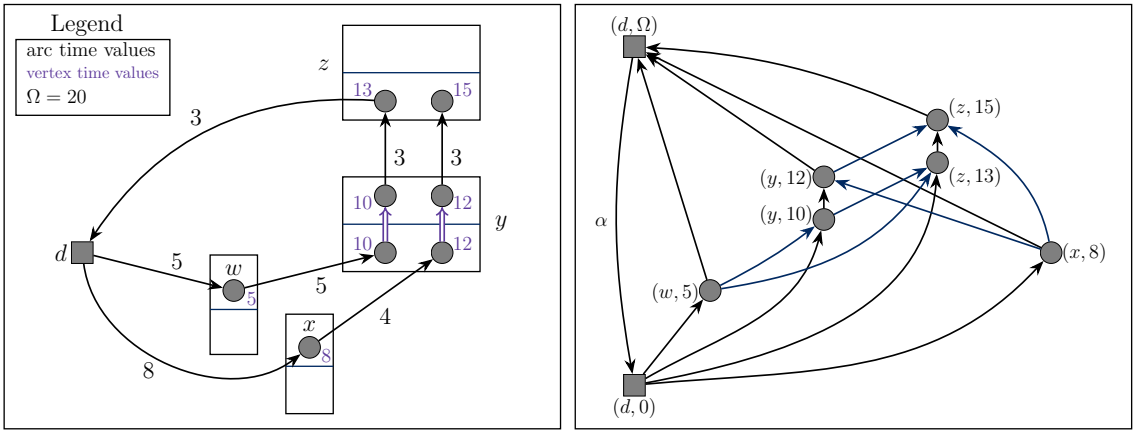


Figure 3.12: Constructing the digraph $Aux(G,\ f,\ \mathbf{t})$. (a) A set of arcs in a digraph $Weak(G,\ f)$ together with their corresponding vertex time values. (b) The digraph $Aux(G,\ f,\ \mathbf{t})$ constructed from the digraph in (a).

Vertices of $Aux(G,\ f,\ \mathbf{t})$. They are all the vertices of $Cover(G,\ f)$. For any x, the vertices of $X^*(x)$ may be ordered according to a linear order σ_x, in such a way that for any $p, q \in X^*(x)$ the following implication holds: $p\,\sigma_x q \Rightarrow t_p \leq t_q$.

Arcs of $Aux(G,\ f,\ \mathbf{t})$. They are as follows.

- One arc $u = (sink,\ source)$, provided with a weight $cost_{Aux(G,f,\mathbf{t})}(u) = \alpha$.

- Any arc $u = (source,\ p = (x, a, m, \varepsilon)), p \in X^*$, such that t_p is at least equal to the minimum *time* weight of a path from d to $x(p)$ in G, and which is minimal for the ordering σ_x with this property. Such an arc u is provided with a weight $cost_{Aux(G,f,\mathbf{t})}(u)$ equal to the *cost* weight of a minimum *time* weight path from d to $x(p)$ in G.

- Any arc $u = (p,\ sink)$, $p \in X^*$, such that $\Omega - t_p$ is at least equal to the minimum *time* weight of a path from $x(p)$ to d in G, and which is maximal for the ordering σ_x with this property. Such an arc u is provided with a weight $cost_{Aux(G,f,\mathbf{t})}(u)$ equal to the *cost* weight of a minimum *time* weight path from $x(p)$ to d in G.

- Any arc $u = (p = (x, a, m, \varepsilon), q = (x', a', m', \varepsilon'))$, $x \neq x'$, such that $t_q - t_p$ is at least equal to the minimum *time* weight of a path, from x to x' in G and that p is maximal for σ_x with this property and q is minimal for $\sigma_{x'}$ with this property. Such an arc u is provided with a weight $cost_{Aux(G,f,\mathbf{t})}(u)$ equal to the *cost* weight of a minimum *time* weight path from $x(p)$ to $x(q)$ in G. If $(a, m) = (a', m')$, then this arc is marked, and provided with a capacity M_u equal to $\left\lceil \frac{z^*_{am}}{\kappa} \right\rceil$.

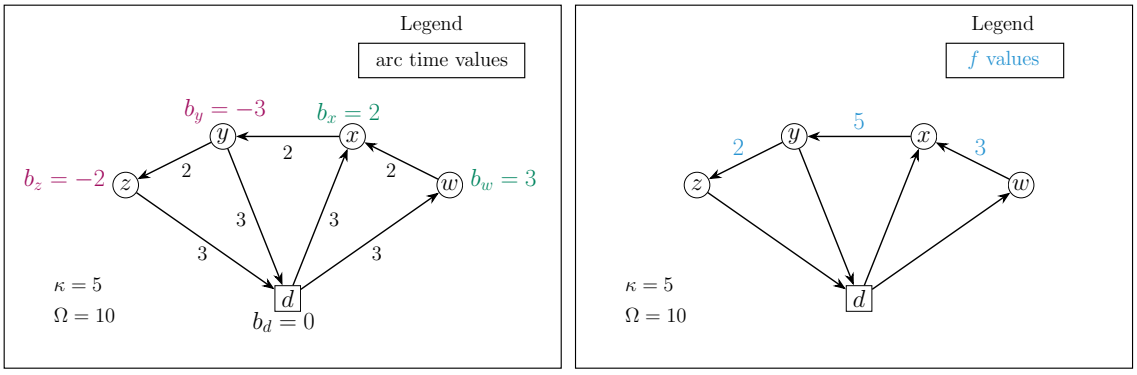
- Any arc $u = (p = (x, a, m, \varepsilon), p' = (x, a', m', \varepsilon'))$, where p and p' are consecutive according to the ordering σ_x. Such an arc u is provided with a weight $cost_{Aux(G,f,\mathbf{t})}(u) = 0$.

Vectors $\mathbf{z}$ and $\mathbf{z}^*$ provide us with a flow h defined on the digraph $Aux(G, f, \mathbf{t})$. In order to get H, we proceed as follows. For any arc $u = \big(p = (x, a, m, \varepsilon), q = (y, a', m', \varepsilon')\big)$, $x \neq y$, which is not a marked arc and is such that $t_q > t_p + time_G(x, y)$, we introduce an auxiliary vertex $q' = \big(y, t_p + time_G(x, y)\big)$ (if it does not already exists), and deviate the flow along the path (p, q', q). Every time we do it, we update accordingly h and σ_x. This process is finite, and when it ends, flow vector $\mathbf{Z}$ and flow h provide us with a feasible IRP solution (H, h), with unchanged cost. ■

Existence of Weak-Lift-Consistent Solutions

Still, solving the Partial Lift Problem according to previously described decomposition scheme remains heuristic, in the sense that not all IRP feasible solutions (H, h) are weak-lift-consistent, as it is shown by the following example.

Example 3.2 - A feasible-path-decomposable flow which is not weak lift-consistent In the digraph $G = (X, A)$ depicted in Figure 3.13, the item flow vector $\mathbf{f} = (f(a), a \in A)$ can be decomposed into two feasible-paths (w, x, y) and (x, y, z). Theorem 2.5 tells us that f can be lifted into a feasible IRP solution (H, h). However we need two vehicles in order to transfer three items from w to y, and two items from x to z, which will traverse arc (x, y) at distinct times. We deduce that digraph $Weak(G, f)$, which allows only one arc from x to y, forbids the existence of a weak-lift-consistent IRP solution (H, h). □



(a) (b)

Figure 3.13: An example of feasible-path-decomposable item flow f that is not weak-lift-consistent. (a) An IRP instance over a digraph $G = (X, A)$. (b) A feasible-path-decomposable item flow f on G that does not allow a weak-lift-consistent IRP solution (H, h).

Checking the existence of weak-lift-consistent IRP solutions can be done by solving the following MILP WLIFT$(G,\ f)$ constraint system.

MILP WLIFT$(G,\ f)$ (constraint system)

Variables

- $\mathbf{z} = (z_u, u = ((x, y, m, -), (x, y', m', +)) \in Router)$, with rational values. Value z_u means the number of items which arrive at vertex x along arc a^m and which are transferred to arc $a'^{m'}$.
- $\boldsymbol{\ell} = (\ell_u, u = ((x, a, m, -), (x, a', m', +)) \in Router)$, with $\{0, 1\}$ values. Value $\ell_u = 1$ means that $z_u \neq 0$.
- $\mathbf{z}^* = (z^*_{a^m}, a^m \in Copy(A))$ with rational values. Value z_u means the number of items which are transported along arc a^m;
- $\mathbf{t} = (t_p, p = (x, a, m, \varepsilon) \in X^*)$ with rational nonnegative values. Value t_p means the time when a vehicle arrives at (in case $\varepsilon = -$) or leaves (in case $\varepsilon = +$) x along arc a^m.

Constraints

- For any router-arc u: $z_u \leq \kappa \cdot \ell_u$. (E37)
- For any copy-arc a^m: $z^*_{a^m} \leq \kappa$. (E38)
- For any vertex $q = (y, a, m, -)$ of $Weak(G, F)$: $z^*_{a^m} \geq \sum_{u \in RouterOut(q)} z_u$. (E39)
- For any vertex $p = (x, a, m, +)$ of $Weak(G, F)$: $z^*_{a^m} \geq \sum_{u \in In(p)} z_u$. (E40)
- For any vertex x of G: $\sum_{v \in CopyIn(x)} z^*_u = \sum_{u \in RouterOut(x)} z_u + \max(-b_x, 0)$. (E41)
- For any vertex x of G: $\sum_{v \in CopyOut(x)} z^*_u = \sum_{u \in RouterIn(x)} z_u + \max(b_x, 0)$. (E42)
- For any arc $a = (x, y)$ of G: $\sum_{u \in Copy(a)} z^*_u = f(a)$. (E43)
- For any copy-arc $(p, q) = \big((x, a = (x, y), m, +), (y, a = (x, y), m, -)\big)$: $t_q \geq t_p + time_G(x, y)$. (E44)

- For any router-arc $u = (p = (x, a, m, -), q = (x, a', m', +))$, the logical implication $(\ell_u = 1) \Rightarrow (t_q \geq t_p)$ holds, which can be represented by the constraint: $\Omega \cdot \ell_u + t_p - t_q \leq \Omega$. (E45)

Theorem 3.3 - Existence of Weak-Lift-Consistent Solutions
A feasible weak-lift-consistent solution $(H,\ h)$ *of the Partial Lift Problem exists if and only if above constraint system* $\mathsf{WLIFT}(G,\ f)$ *admits some feasible solution.*

Proof. Deriving a solution $(\mathbf{z},\ \mathbf{z}^*,\ \boldsymbol{\ell},\ \mathbf{t})$ from a feasible IRP solution is pure routine. Conversely, let us suppose that we are provided with a feasible solution $(\mathbf{z},\ \mathbf{z}^*,\ \boldsymbol{\ell},\ \mathbf{t})$ of $\mathsf{WLIFT}(G,\ f)$. Then we build digraph $Aux(G,\ f,\ \mathbf{t})$ exactly as in the proof of Theorem 3.2. In order to get vector H, we solve the following Minimum Cost Flow Problem $\mathsf{MCF}(G,\ f,\ \mathbf{t})$.

$\mathsf{MCF}(G,\ f,\ \mathbf{t})$: Compute an integral flow vector $\mathbf{Z}$ on the digraph $Aux(G,\ f,\ \mathbf{t})$ such that

- $\mathbf{Z}$ satisfies the flow conservation condition at every vertex;
- for any marked arc u, $Z_u \geq M_u$, where M_u is the capacity defined in the setting of $Aux(G,\ f,\ \mathbf{t})$.
- $\mathbf{Z}$ minimizes the cost $\sum_{u \in A(Aux(G,f,\mathbf{t}))} cost_{Aux(G,f,\mathbf{t})} \cdot Z_u$.

Then we derive H from $\mathbf{Z}$ the same way we did in the last part of Theorem 3.2 and we conclude. ■

In the next section we present some numerical results (see Table 3.2) of solving the $\mathsf{WLIFT}(G,\ f)$ MILP, over the same instances in Table 3.1. Those numerical results allow us to check that in most cases, a feasible-path-decomposable flow f induces a feasible $\mathsf{WLIFT}(G,\ f)$ MILP. In such a case, intuition tells us that we should not need to involve, while dealing with the Partial Lift Problem, flow vectors h which are not weak-lift-consistent.

This intuition leads us to set the following conjecture.

Conjecture 1 - Exactness of Weak/Cover Decomposition Scheme for Weak-Lift-Feasible Solutions
If above constraint system $\mathsf{WLIFT}(G,\ f)$ is feasible, then the Weak/Cover decomposition scheme solves the Partial Lift Problem in an exact way.

3.3 Handling the Weak/Cover Decomposition

In this last section, we briefly address the issue of the practical design of an algorithm for the Partial Lift Problem. Previously described Weak/Cover decomposition scheme involves, as its master object, a collection Λ of arcs of the $Cover(G,\ f)$ digraph.

This suggests us to design algorithms which focus on computing "good" collections Λ. As it usually happens when it comes to the design of algorithm for multi-level models, the main difficulty here is to derive from the slave $\mathsf{CTIME}(\Lambda)$, $\mathsf{WEAK}(G,\ f,\ \Lambda)$ and $\mathsf{COVER}(G,\ f,\ \Lambda)$ levels sensitivity information which will allow us to efficiently drive master collection Λ.

We notice that $\mathsf{WEAK}(G,\ f,\ \Lambda)$ is a simple rational linear program, with constraints related to main vector $\mathbf{z}$ which are transportation constraints set on bipartite digraphs, that $\mathsf{COVER}(G,\ f,\ \Lambda)$ is a simple Minimum Cost Flow LP model and that $\mathsf{CTIME}(\Lambda)$ is about the computation of a maximum weight path in an acyclic digraph.

A consequence is that we may rely on linear programming duality in order to retrieve information from the resolution of the slave levels, and drive the master object Λ accordingly.

First, in Section 3.3.1 we compact our Weak/Cover formulation in a way which skips slave levels $\mathsf{CTIME}(\Lambda)$, $\mathsf{WEAK}(G,\ f,\ \Lambda)$ and $\mathsf{COVER}(G,\ f,\ \Lambda)$ and only keeps constraints related to master object Λ. Next (Section 3.3.2-19) we derive a simple algorithmic handling of Λ which we apply to the instances of Table 3.1.

3.3.1 An Exact MILP Weak/Cover Reformulation

We turn the previous Weak/Cover decomposition scheme into a $\mathsf{WPLIFT}(G,\ f)$ MILP model to search for an optimal weak-lift-feasible solution of the Partial Lift Problem.

We consider vectors $\mathbf{t}$, $\mathbf{z}$, $\mathbf{z}^*$, and $\mathbf{Z}$ as in Section 3.2.2 and introduce an additional $\{0,1\}$-valued vector $\boldsymbol{\chi} = \big(\chi_u, u \in A(Cover)\big)$. We merge the programs $\mathsf{CTIME}(\Lambda)$, $\mathsf{WEAK}(G,\ f,\ \Lambda)$, and $\mathsf{COVER}(G,\ f,\ \Lambda)$ into a single MILP, while discarding constraints (E26), (E32), and (E35). Next, we add the constraints (E46)-(E49) and the objective function (E50) which are described below.

MILP $\mathsf{WPLIFT}(G,\ f)$

Variables

- $\mathbf{t} = (t_p,\ p \in V(Cover)) \in \mathbb{R}_+$.
- $\mathbf{z} = (z_u,\ u \in Router) \in \mathbb{R}_+$.
- $\mathbf{z}^* = (z^*_u,\ u \in Copy(A)) \in \mathbb{R}_+$.
- $\mathbf{Z} = (Z_u,\ u \in A(Cover)) \in \mathbb{R}_+$.
- $\boldsymbol{\chi} = (\chi_u,\ u \in A(Cover)) \in \{0,\ 1\}^{|A(Cover)|}$.

Constraints

- Include constraints (E22)-(E25), (E27)-(E31), and (E33)-(E34).
- For any marked arc u: $\chi_u = 1$. (E46)
- For any arc $u = (p, q) \in X^* \times X^*$: (E47)
  if $\chi_u = 1$ then $t_p + time_G(x(p),\ x(q)) \le t_q$.
- For any router-arc u: if $\chi_u = 0$ then $z_u = 0$. (E48)
- For any unmarked arc $u \in X^* \times X^*$: if if $\chi_u = 0$ then $Z_u = 0$. (E49)

Objective Function

The cost (E50)

$$\alpha \cdot Z_{(sink,\,source)} + \beta \cdot \sum_{u \in X^* \times X^*} cost_{Cover(G,\,f)}(u) \cdot Z_u + \gamma \cdot \sum_{a \in A} time(a) \cdot f(a)$$

is smallest possible.

3.3.2 A Simple Monotonic Cover Algorithm

There are clearly many ways to turn previously described Weak/Cover Decomposition Scheme and above $\mathsf{WPLIFT}(f)$ reformulation into an algorithm. We propose, implement and test here a very simple "Monotonic Cover" algorithm which basically works in two steps, without any feedback loop.

- 1st step: Solve the $\mathsf{WLIFT}(G, f)$ model and derive vectors $\mathbf{z}^*$ and $\mathbf{z}$ and $\boldsymbol{\ell}$, and initialize collection Λ with the arcs of $Cover(G, f)$ which support $\mathbf{z}^*$ and $\mathbf{z}$.
- 2nd step: While it is possible to maintain the feasibility of constraint system $\mathsf{CTIME}(\Lambda)$, make Λ evolve in a monotonic way by first, at any iteration, solving the $\mathsf{COVER}(G, f, \Lambda)$ Minimum Cost Flow model, and next inserting into Λ a non-active arc (p, q) which would induce a new constraint on the dual of $\mathsf{COVER}(G, f, \Lambda)$ that makes infeasible the current dual optimal solution of $\mathsf{COVER}(G, f, \Lambda)$. At any iteration, retrieve some IRP solution and update the best solution ever obtained.

Algorithm 6: Monotonic Cover algorithm.

input : A parameter $\varepsilon > 0$, and an item flow f such that $\mathsf{WLIFT}(G, f)$ is feasible.
output: A TEN IRP solution (H^{cur}, h^{cur}) over the digraph $Aux(G, f, \mathbf{t})$ defined in the proof of Theorem 3.2.

Solve $\mathsf{WLIFT}(G, f)$ MILP and retrieve vectors $\mathbf{z}^*$ and $\boldsymbol{\ell}$;
Set $\Lambda \leftarrow Copy(A) \cup \{u \in Router : \ell_u = 1\}$ and construct digraph $Cover(G, f)$;
Solve $\mathsf{COVER}(G, f, \Lambda)$ and $\mathsf{CTIME}(\Lambda)$;
Retrieve a primal solution $(\mathbf{H}^{cur}, \mathbf{h}^{cur})$ and a dual solution $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ of $\mathsf{COVER}(G, f, \Lambda)$;
Set `CurCost` $\leftarrow cost(\mathbf{H}^{cur}\ \mathbf{h}^{cur})$ and `stop` $\leftarrow$ `False`;
while not `stop` **do**
    Search for an arc $u = (p, q)$ such that $\lambda_p - \lambda_q + \mu_u - cost_{Cover(G,f)}(u) > \varepsilon$ and such that $\mathsf{CTIME}(\Lambda \cup \{u\})$ remains feasible;
    if no arc u was found at previous step **then**
        Take $\mathbf{H}^{cur} = (H^{cur}_a, a \in A(Cover))$;
        Take $\mathbf{h}^{cur} = (h^{cur}_a, a \in A(Cover)))$;
        `stop` $\leftarrow$ `True` and **return** $(\mathbf{H}^{cur}, \mathbf{h}^{cur})$;
    else
        $\Lambda \leftarrow \Lambda \cup \{u\}$;
        Retrieve a feasible vector solution $\mathbf{t}$ of $\mathsf{CTIME}(\Lambda)$ and take $\Lambda(\mathbf{t}) = \{(p, q) \in Cover(G, f) : t_q \geq t_p + time_G(x(p), x(q))\}$;
        Solve the Minimum Cost Flow problem $\mathsf{COVER}(G, f, \Lambda(\mathbf{t}))$ and retrieve a primal solution $(\mathbf{H}, \mathbf{h})$ and a dual solution $(\boldsymbol{\lambda}, \boldsymbol{\mu})$;
        If $cost(\mathbf{H}, \mathbf{h}) <$ `CurCost`, then update $(\mathbf{H}^{cur}, \mathbf{h}^{cur})$ to $(\mathbf{H}, \mathbf{h})$, and `CurCost` to $cost(\mathbf{H}, \mathbf{h})$;

Implementation Details

Although the construction of the MILP models from Sections 3.1-3.2.3 can be performed without much difficulty, the implementation of Algorithm 6 it is a little more delicate because it involves the interaction and updating of several LP models. Furthermore, Algorithm 6 uses some LP dual solutions, which in turn depend on the particular implementation of their LP primal counterparts. Therefore, in this subsection we describe a way of implementing Algorithm 6.

Throughout this subsection, the *identification* of the elements of a finite set X with index values $x = 1, \dots, n$, means that we have established a bijective function $\phi : X \to \{1, \dots, n\}$, and we are using the symbol x to denote both an element of X, and the index $\phi(x)$. The particular interpretation of x should be clear from context so confusion should not result.

Input. Consider an IRP instance consisting of a digraph $G = (X, A)$, a vector $\mathbf{b}$ of vertex balance coefficients, a time horizon Ω, a fleet of vehicles with finite capacity $\kappa > 0$, a weight function $cost : A \to \mathbb{R}_+$, a weight function $time : A \to \mathbb{R}_+$, and non-negative cost parameters α, β, and γ. Let us suppose, also, that we have a solution (F, f) of the corresponding PIRP model.

If we consider that any vertex in X is identified with an index value $x = 1, \dots, n$, and that any arc in A is identified with an index value $a = 1, \dots, m$. Then, we should be provided with an array `balance`, indexed over $x = 1, \dots, n$, and with an array `item_flow`, indexed over $a = 1, \dots, m$. For every index x, the entry `balance[`x`]` is the balance coefficient b_x of the corresponding vertex x; and for every index a, the entry `item_flow[`a`]` is the value of the items flow f passing through the corresponding arc a.

We also should be provided with an array `arc`, with indexation over $a = 1, \dots, m$. The entry `arc[`a`]` codifies an arc $a = (x, y) \in A$ and contains the following four records:

1. a record `arc[`a`].tail` containing the tail x of arc a;
2. a record `arc[`a`].head` containing the head y of arc a;
3. a record `arc[`a`].weight_cost` equal to $cost(a)$;
4. a record `arc[`a`].weight_time` equal to $time(a)$.

We consider that we have a digraph data structure based on adjacency lists that encodes the digraph G. This data structure consists of two bidimensional arrays `adj_arc_out`, and `adj_arcs_in`, both with the first dimension indexed by $x = 1, \ldots, n$. The structure `adj_arcs_out[`x`]` is a ragged array that contains the indices a of all the arcs with tail x; similarly `adj_arcs_in[`x`]` is a ragged array that contains the indices a of all the arcs with head x. In other words, `adj_arcs_out[`x`]` (respectively, `adj_arcs_in[`x`]`) contains all the elements of $\partial^+_G(x)$ (respectively, all the elements of $\partial^-_G(x)$).

Also, we should be provided with two bidimensional arrays `cost_path`, and `time_path`; both indexed over $\{1,\ldots,n\} \times \{1,\ldots,n\}$. For all (x, y) in $\{1,\ldots,n\} \times \{1,\ldots,n\}$, the entry `cost_path[`$x$`][`$y$`]` is equal to $\text{dist}_{(G,cost)}(x, y)$, i.e., the minimum weight of a path from vertex x to vertex y in the digraph G (with respect to the weight function *cost*); analogously, the entry `time_path[`x`][`y`]` is equal to $\text{dist}_{(G,time)}(x, y)$, i.e., the minimum weight of an (x, y)-path in G (with respect to the weight function *time*).

Encoding sets $Copy(A)$ and X^* of digraph $Weak(G, f)$. We start by defining $I = \sum_{a\in A}\lceil \frac{f(a)}{\kappa} \rceil$. Then, we scan the entries of array `item_flow`, and for every $a = 1, \ldots, m$, such that `item_flow[`a`]`> 0, we create $\lceil$`item_flow[`a`]`$/\kappa\rceil$ new index values i; every such value will correspond to a copy-arc of $Copy(A)$ in the digraph $Weak(G,f)$. So, we identify any copy-arc in $Copy(A)$ with an index $i = 1, \ldots, I$, and we create an array `copy_arc` indexed over $i = 1, \ldots, I$. We fill entry `copy_arc`$[i]$ with the following five records initialized in the following way:

1. `copy_arc[`i`].arc` $= a$,
2. `copy_arc[`i`].head_in_G` $=$ `arc[`a`].head`,
3. `copy_arc[`i`].tail_in_G` $=$ `arc[`a`].tail`,
4. `copy_arc[`i`].weight_cost` $=$ `arc[`a`].weight_cost`,
5. `copy_arc[`i`].weight_time` $=$ `arc[`a`].weight_time`.

In a similar way, at the same time we are dealing with an arc $a = (x, y)$ we fill two bidimensional arrays `adj_copy_arcs_out`, and `adj_copy_arcs_in`, by adding the current index i to `adj_copy_arcs_out[`x`]` and `adj_copy_arcs_in[`y`]`.

Hence at the very end, we have a bidimensional array `adj_copy_arcs_out` that provide us, for every $x \in X$ with a ragged array `adj_copy_arcs_out[`x`]` containing the indices i of all the copy-arcs with `copy_arc[`i`].tail_in_G` = x; similarly, we have a bidimensional array `adj_copy_arcs_in` that provide us, for every $x \in X$ with a ragged array `adj_copy_arcs_in[`x`]` containing the indices i of all the copy-arcs with `copy_arc[`i`].head_in_G` = x.

Next, in the digraph $Weak(G, f)$ we can identify the copy-vertex head of copy-arc i with the index value i, and the copy-vertex tail of copy-arc i with the index value $I + i$. Proceeding this way, we identify every copy-vertex in X^* with an index value $i = 1, \ldots, 2I$. Then, we create an array `lift_vertex` indexed from 1 to $2I$. For i from 1 to I we set `lift_vertex[`i`]` = `copy_arc[`i`].head_in_G`, and `lift_vertex[`$I + i$`]` = `copy_arc[`i`].tail_in_G`. Note that, by using this indexation, we can determine easily that, in the digraph $Weak(G, f)$, the copy-arc encoded by `copy_arc[`i`]` has a copy-vertex head with index value i, and a copy-vertex tail with index value $I + i$. We also know that any copy-vertex i corresponds to a copy of the vertex `lift_vertex[`i`]` in digraph G.

Encoding set $Router$ **of digraph** $Weak(G, f)$**.** For all vertex $x \in X$, for all copy-arc i in `adj_copy_arcs_in[`x`]`, and for all copy-arc i' in `adj_copy_arcs_out[`x`]` we create a a triple (x, i, i') and a new index value j. Then, we identify every one of those triples with an index value $j = 1, \ldots, J$. For every index value j, we store the triple (x, i, i') in an array `router_arc`. Note that, such a triple (x, i, i') represents a router-arc in $Router(x)$ with copy-vertex head i and copy-tail $I + i'$.

Conversely, we introduce two bidimensional arrays `adj_router_arcs_out` and `adj_router_arcs_in`. For every i in $\{1, \ldots, I\}$, `adj_router_arcs_in[`i`]` is a ragged array containing the indices j such that `router_arc[`j`]` has its second coordinate equal to i; similarly `adj_router_arcs_out[`i`]` is a ragged array containing the indices j such that `router_arc[`j`]` has its third coordinate equal to i. In other words, for every $x \in X$, and every copy-vertex $i \in X^*(x)$, the array `adj_router_arcs_in[`i`]` contains the indices j of all the router-arcs in $Router(x)$ with copy-vertex head i (i.e., the router-arcs in $RouterIn(i)$); similarly, the array `adj_router_arcs_out[`i`]` contains the indices j of all the router-arcs in $Router(x)$ with copy-vertex tail $I + i$ (i.e., the router-arcs in $RouterOut(I + i)$).

Encoding WLIFT(G,f) MILP. We can use the above-defined arrays `lift_vertex`, `copy_arc`, and `router_arc` to guide the construction of the MILP WLIFT(G,f). For all $a^m \in Copy(A)$, the variables $z^*_{a^m}$ can be indexed from 1 to I; whilst, for $u \in Router$, variables z_u and ℓ_u can be indexed from $I + 1$ to $I + J$.

On the other hand, for $p \in X^*$ the variables t_p can be indexed from 1 to $2I$ in the following way. For every index i from 1 to I, we create one variable t_i corresponding to the copy-vertex head (in the digraph $Weak(G,\, f)$) of copy-arc i, and one variable t_{I+i} corresponding to the copy-vertex tail (in the digraph $Weak(G,\, f)$) of copy-arc i. Proceeding this way, we can see that for i in $\{1,\, \ldots\, ,\, 2I\}$, the real variable t_i corresponds to the time value associated with copy-vertex i in X^*.

Encoding vertices of digraph $Aux(G,f,\, \mathbf{t})$. We suppose here that we have a feasible solution $\zeta = (\mathbf{z}, \mathbf{z}^*,\, \boldsymbol{\ell},\, \mathbf{t})$ of the MILP $\mathsf{WLIFT}(G,\, f)$. Vertices of $Aux(G,f,\, \mathbf{t})$ are all the pairs (i, t_i) with i from 1 to $2I$, together with a new pair $source = (d, 0)$ with an index value $2I+1$, and a new pair $sink = (d, \Omega)$ with an index value $2I+2$. For encoding those vertices, we create an array `aux_vertex` indexed from 1 to $2I+2$. For any such a value i in $\{1, \ldots, 2I\}$ we store in `aux_vertex[`i`]` the following two records initialized in the following way:

1. `aux_vertex[`i`].vertex_in_G` = `lift_vertex[`i`]`,
2. `aux_vertex[`i`].time` = t_i.

We also define:

1. `aux_vertex[`$2I+1$`].vertex_in_G` = d,
2. `aux_vertex[`$2I+1$`].time` = 0;
3. `aux_vertex[`$2I+2$`].vertex_in_G` = d,
4. `aux_vertex[`$2I+2$`].time` = Ω.

Conversely, we create a bidimensional array `aux_vertex_back` indexed by X; and for $x = 1,\, \ldots\, ,\, n$, we store in the array `aux_vertex_back[`x`]` all the indices i such that `aux_vertex[`i`].vertex_in_G` = x. In other words, `aux_vertex_back[`x`]` contains the indices of all the vertices in $X^*(x)$.

Encoding arcs of digraph $Aux(G,f,\, \mathbf{t})$. We take all the pairs (i, i') with $i \in \{1, \ldots, 2I, 2I+1\}$, and $i' \in \{1, \ldots, 2I, 2I+2\}$, together with the pair $(2I+2, 2I+1)$. Any pair (i, i') represents an arc $\big((i, t_i), (i', t_{i'})\big)$ of $Aux(G,f,\, \mathbf{t})$, and can be identified with an index value $k = 1,\, \ldots\, ,\, K$. For every such an index value k representing a pair (i, i'), we store the following records on an array `covering_arc`:

- `aux_arc[k].tail`=i,
- `aux_arc[`k`].head`=i',

- a weight *cost* `aux_arc[`k`].weight_cost`, which is equal to $\beta \cdot$ `cost_path[`x`][`y`]` if i is related to vertex x (i.e., if `lift_vertex[`i`]` = x), and i' is related to vertex y (i.e., if `lift_vertex[`i'`]` = y); but when k represents the pair $(i, i') = (2I + 2, 2I + 1)$, we set `aux_arc[`$k$`].weight_cost` = α.

- a weight *time* `aux_arc[`k`].weight_time`, which is equal to $\gamma \cdot$ `time_path[`x`][`y`]`, if i is related to vertex x, and i' is related to vertex y.

- a flag value `aux_arc[`k`].mark`, which is equal to 1 if $1 \leq i' \leq I$, $I + 1 \leq i \leq 2I$, and $i + i' = 2I$ (i.e. if i' and i are, respectively, the head and tail of the copy-arc stored at `copy_arc[`i'`]`). In such a case, we store a record value `aux_arc[`k`].item_flow` equals to $z^*_{i'}$. Otherwise, we set `aux_arc[`k`].mark` = 0, and `aux_arc[`k`].item_flow` = 0.

- a flag value `aux_arc[`k`].active`, which tell us whether arc (i, i') is active: any arc k such that `aux_arc[`k`].mark` = 1 is active, as well as the arc $(2I+2, 2I+1)$ (i.e. the arc $(sink, source)$ of $Aux(G,f,\ \mathbf{t})$). As for the other arcs k connecting vertex (`aux_vertex[`i'`].vertex_in_G`, `aux_vertex[`i'`].time`) to vertex (`aux_vertex[`i`].vertex_in_G`, `aux_vertex[`i`].time`) we set `aux_arc[`k`].active` = 1, if

  Case 1: `aux_vertex[`i'`].vertex_in_G` = `aux_vertex[`i`].vertex_in_G`, and `aux_vertex[`i'`].time` ≤ `aux_vertex[`i`].time`

  Case 2: `aux_vertex[`i'`].vertex_in_G` ≠ `aux_vertex[`i`].vertex_in_G`, `aux_vertex[`i'`].time` + `aux_arc[`k`].weight_time` ≤ `aux_vertex[`i`].time`, and there does not exist another vertex $(i'',\ t_{i''})$ of $Aux(G,f,\ \mathbf{t})$ with

  * `aux_vertex[`i''`].vertex_in_G` = `aux_vertex[`i`].vertex_in_G`,
  * `aux_vertex[`i'`].time` < `aux_vertex[`i''`].time`, and
  * `aux_vertex[`i''`].time` < `aux_vertex[`i`].time`.

Note that, we can use the array `aux_vertex_back` to initialize this last record.

As usual, while building the arc set $\{1, \ldots, K\}$ of $Aux(G,f,\ \mathbf{t})$, we also build two arrays `adj_aux_arcs_out`, and `adj_aux_arcs_in`, both with indexation on $1, \ldots, 2I+2$, with the meaning:

- `adj_aux_arcs_out[`i`]` provide us with the list of arcs k with tail i, and

- `adj_aux_arcs_in[`i`]` provide us with the list of arcs k with head i.

Encoding MCF(G, f, $\mathbf{t}$). The linear program MCF(G, f, $\mathbf{t}$) aims to compute an integral flow vector $\mathbf{Z} = (Z_k,\, k = 1, \ldots, K)$ with $Z_k = 0$ for all $k \in \{1, \ldots, K\}$ such that `aux_arc[`k`].active` $= 0$, and that minimizes $\sum_{k \in K}$ (`aux_arc[`k`].weight_cost`) $\cdot\, Z_k$ + $\sum_{k \in K}$ (`aux_arc[`k`].time_cost`) $\cdot$ (`aux_arc[`k`].item_flow`).

Note the second sum is a constant that corresponds to the cost c_3 related to the item ride time. The term is constant because we search for solutions (H,h) such that f is the projection of h.

The following constraints are added to the linear program in the order below:

1. Z satisfies flow conservation at any vertex of $Aux(G,f,\, \mathbf{t})$, so, for any $i = 1, \ldots, 2I + 2$ we add the constraint

$$\sum_{k \in \partial^-_{Aux(G,f,\mathbf{t})}(i)} Z_k = \sum_{k \in \partial^+_{Aux(G,f,\mathbf{t})}(i)} Z_k\,,$$

   notice we can use previous arrays `adj_aux_arcs_in` and `adj_aux_arcs_out` in order to set those constraints.

2. For any $k = 1, \ldots, K$, if `aux_arc[`k`].mark` $= 1$, we add the constraint:

$$Z_k \geq \left\lceil \frac{\texttt{aux\_arc[}k\texttt{].item\_flow}}{\kappa} \right\rceil,$$

   but if `aux_arc[`k`].mark` $= 0$, we simply add the constraint

$$Z_k \geq 0.$$

The dual solution vector of MCF(G, f, $\mathbf{t}$) is a vector (u, y) with indexation $\mathbf{u} = (u_i, i = 1, \ldots, 2I + 2)$, and $\mathbf{y} = (y_k, k = 1, \ldots, K,$ with `aux_arc[`k`].active` $= 1)$, and the constraints in the dual of MCF(G, f, $\mathbf{t}$) are as follows.

For all $k = (i, i')$, such that `aux_arc[`k`].active` $= 1$:

- if `aux_arc[`k`].mark` $= 1$, we have a constraint

$$u_i - u_{i'} + y_k \leq \texttt{aux\_arc[k].weight\_cost};$$

- if `aux_arc[`k`].mark` $= 0$, we have a constraint

$$u_i - u_{i'} \leq \texttt{aux\_arc[k].weigth\_cost}.$$

A LP implementation of CTIME(Λ). The first time we solve the WLIFT(G,f) MILP, we are provided with a solution ζ and a collection Λ consisting of all the arcs carrying at least one item.

We can obtain a LP implementation of CTIME(Λ) from the WLIFT(G,f) MILP by simply fixing the values of variables in $\mathbf{z}$, $\boldsymbol{\ell}$, and $\mathbf{z}^*$ to the corresponding values computed in the solution ζ. Proceeding that way, adding a new arc (i, i') to Λ is equivalent to the introduction of a constraint $t_{i'} \geq t_i + cost_{Aux(G,f,\mathbf{t})}((i, i'))$ to that implementation of CTIME(Λ).

Additional comments. If $\mathbf{Z}$= $(Z_k,\, k \in A(Aux(G, f, \mathbf{t})))$ is an optimal solution of current MCF(G,f, $\mathbf{t}$) model and $(\mathbf{u}, \mathbf{y})$ is an optimal LP dual solution, the only way to possibly reduce the cost below the optimal value, it could be to allow the use of an arc $k = \big((i,\, t_i),\, (i',\, t_{i'})\big) \in Aux(G, f, \mathbf{t})$ with `aux_arc[`k`].active`= 0.

Now we may note that, given an arc $k = \big((i,\, t_i),\, (i',\, t_{i'})\big) \in Aux(G,\, f,\, \mathbf{t})$ with `aux_arc[`k`].active`= 0, the change of a constraint $Z_k = 0$ in MCF(G,f,$\mathbf{t}$) by a constraint $Z_k \geq 0$, would be equivalent to add a constraint $u_{(i,t_i)} - u_{(i',t_{i'})} \leq cost_{Aux(G,f,\mathbf{t})}(k)$ to the LP dual problem of MCF(G, f, $\mathbf{t}$). If this constraint is already satisfied by the current LP dual solution $(\mathbf{u}, \mathbf{y})$, then the optimal value would not change. So, we are interested in finding an arc $k = \big((i,\, t_i),\, (i',\, t_{i'})\big)$ with `aux_arc[`k`].active`= 0, and such that $u_{(i,t_i)} - u_{(i',t_{i'})} > cost_{Aux(G,f,\mathbf{t})}(k)$, because the incorporation of the opposite constraint $u_{(i,t_i)} - u_{(i',t_{i'})} \leq cost_{Aux(G,f,\mathbf{t})}(k)$ into the LP dual model, makes infeasible the current dual optimal solution $(\mathbf{u}, \mathbf{y})$.

We will not add this constraint directly into the dual of MCF(G,f,$\mathbf{t}$). Instead we will include such an arc k into the collection Λ, and that means the addition of a constraint $t_{i'} \geq t_i + cost_{Aux(G,f,\mathbf{t})}(k)$ into the above described LP implementation of CTIME(Λ).

Now we provide a complete example of the Monotonic Cover algorithm.

Example 3.3 - Lifting a solution with Algorithm 6

Figure 3.14 shows an IRP instance over a digraph $G = (X, A)$ with a depot vertex 0. Vertices 1, 3, and 5 are excess vertices with $b_1 = 1$, $b_3 = 10$, and $b_5 = 1$. Vertices 2, 4, and 6 are deficit vertices with $b_2 = -1$, $b_4 = -10$, and $b_6 = -1$. Vehicles have an item capacity $\kappa = 10$ and the time horizon for the relocation process is $\Omega = 56$.

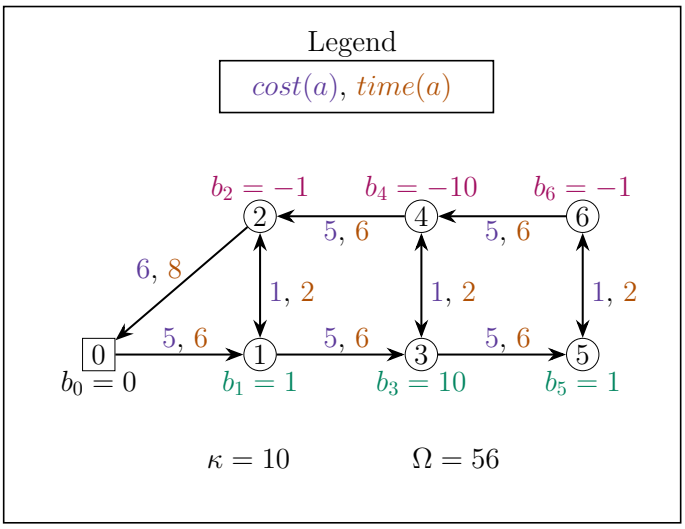


Figure 3.14: An example of IRP instance over a digraph $G = (X, A)$ with $X = \{0, 1, 2, 3, 4, 5, 6\}$, depot vertex 0, and $A = \{(0, 1), (1, 2), (1, 3), (2, 0), (2, 1), (3, 4), (3, 5), (4, 2), (4, 3), (5, 6), (6, 4), (6, 5)\}$.

Figure 3.15 shows an item flow f on the same digraph G that allows to transport the items from excess vertices to deficit vertices. Note that f is feasible-path-decomposable for the given time horizon $\Omega = 56$.

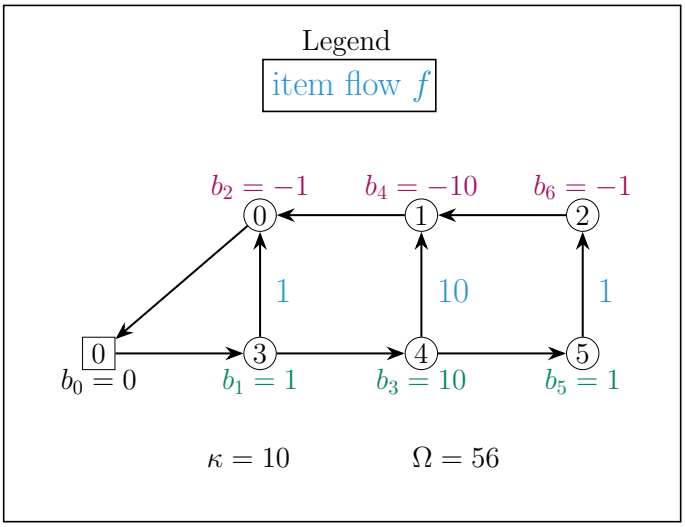


Figure 3.15: An example of item flow f over the same instance in Figure 3.14. Note that f allows to transport the items from excess vertices to deficit vertices, and is feasible-path-decomposable for the time horizon $\Omega = 56$.

For the instance in Figure 3.14 and the item flow f in Figure 3.15, we formulate and solve the MILP WLIFT(G,f). Figure 3.16 shows a WLIFT(G,f) solution over the digraph $Weak(G,f)$. Note the set of router-arcs is empty and the set of copy-arcs consists of the three arcs $(3,0)$, $(4,1)$, and $(5,2)$.

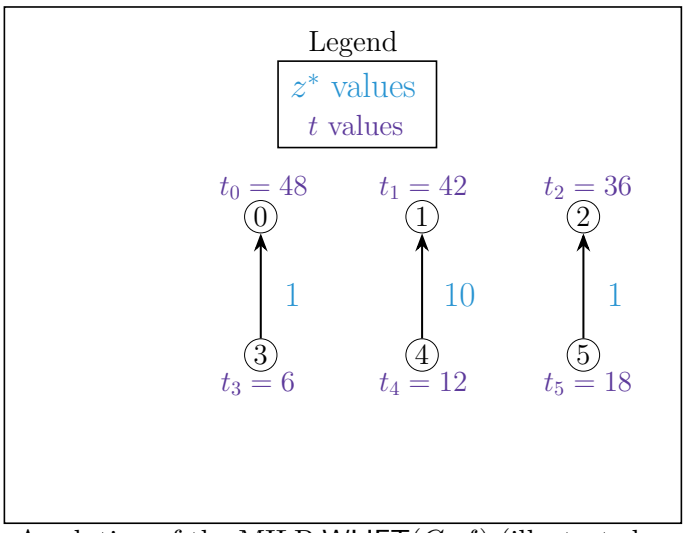


Figure 3.16: A solution of the MILP WLIFT(G, f) (illustrated over the digraph $Weak(G,f)$) for the digraph G in Figure 3.14 and the item flow in Figure 3.15.

Now, we construct the digraph $Aux(G, f, \mathbf{t})$ from the WLIFT(G,f) solution depicted in Figure 3.16. The digraph $Aux(G, f, \mathbf{t})$ is depicted in Figure 3.17. In the Figures 3.16-3.21 of this example, we have drawn each vertex p of $Aux(G, f, \mathbf{t})$, according to the corresponding t_p value: a vertex p is drawn at a lower position than a vertex q if and only if $t_p \leq t_q$.

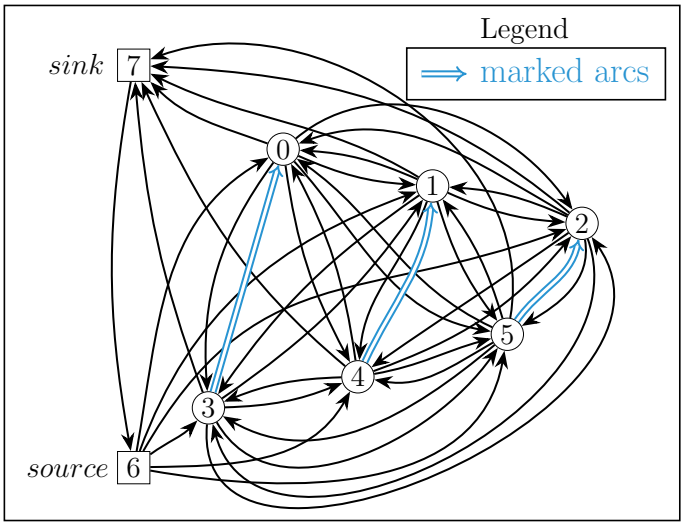


Figure 3.17: The digraph $Aux(G, f, \mathbf{t})$ constructed from the WLIFT(G, f) solution in Figure 3.16.

We solve the Minimum Cost Flow $\mathsf{MCF}(G, f, \mathbf{t})$ for the vector $\mathbf{t}$=(t_0 = 48, t_1 = 42, t_2 = 36, t_3 = 6, t_4 = 12, t_5 = 18) and we obtain the solution depicted in Figure 3.18 which involves three vehicles.

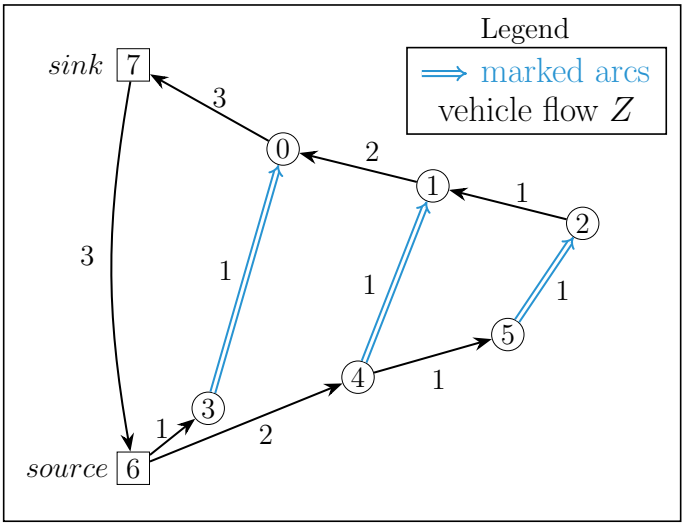


Figure 3.18: A $\mathsf{MCF}(G,f,\mathbf{t})$ solution over a subgraph of $Aux(G, f, \mathbf{t})$. We have considered the vector $\mathbf{t}$ from the $\mathsf{WLIFT}(G,f)$ solution in Figure 3.16.

Now, we retrieve a dual solution $(\mathbf{y},\mathbf{u})$ of $\mathsf{MCF}(G, f, \mathbf{t})$ and we identify the non-active arc $(0,4)$ such that $y_0 - y_4 - cost_{Aux(G,f,\mathbf{t})}((0,4)) = -322 < 0$. We add the arc $(0,4)$ to Λ, that means a new constraint $t_4 \geq t_0 + cost_{Weak(G,f)}((0,4))$ in $\mathsf{CTIME}(G,\Lambda)$. We solve $\mathsf{CTIME}(G,\Lambda)$ and retrieve a solution $\mathbf{t}'$ = (t_0 = 8, t_1 = 42, t_2 = 36, t_3 = 6, t_4 = 16, t_5 = 18). Then we solve $\mathsf{MCF}(G, f, \mathbf{t}')$ and we obtain the solution depicted in Figure 3.19 which involves two vehicles.

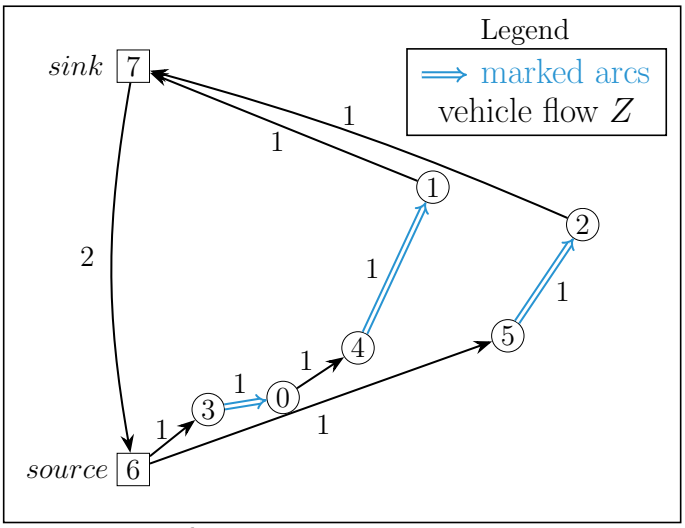


Figure 3.19: A $\mathsf{MCF}(G,f,\mathbf{t}')$ solution over a subgraph of $Aux(G, f, \mathbf{t})$. We have taken the vector $\mathbf{t}'$ = (t_0 = 8, t_1 = 42, t_2 = 36, t_3 = 6, t_4 = 16, t_5 = 18).

Again, we retrieve a dual solution $(\mathbf{y},\mathbf{u})$ of $\mathsf{MCF}(G,\ f,\ \mathbf{t})$ and we identify the non-active arc $(0,5)$ such that $y_0 - y_5 - cost_{Aux(G,f,\mathbf{t})}((0,5)) = -332 < 0$. We add the arc $(0,5)$ to Λ, that means a new constraint $t_5 \geq t_0 + cost_{Weak(G,f)}((0,5))$ in $\mathsf{CTIME}(G,\Lambda)$. We solve $\mathsf{CTIME}(G,\Lambda)$ and retrieve a solution $\mathbf{t}' = (t_0 = 8,\ t_1 = 42,\ t_2 = 36,\ t_3 = 6,\ t_4 = 16,\ t_5 = 22)$. Then we solve $\mathsf{MCF}(G,\ f,\ \mathbf{t}')$ and we obtain the solution depicted in Figure 3.20 which still involves two vehicles.

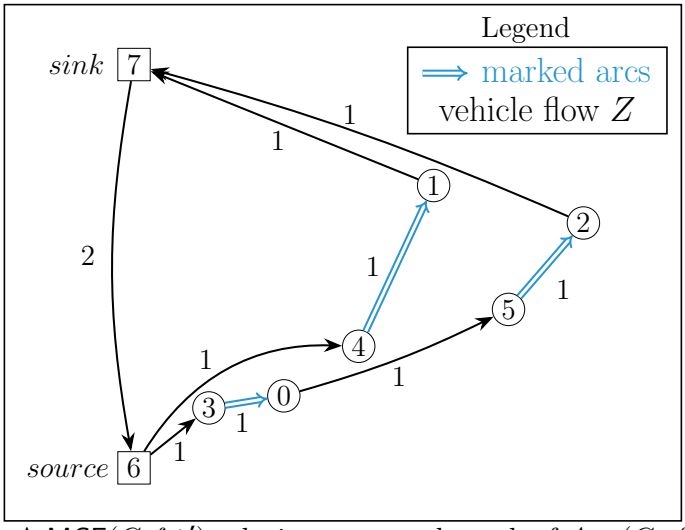


Figure 3.20: A $\mathsf{MCF}(G,f,\mathbf{t}')$ solution over a subgraph of $Aux(G,\ f,\ \mathbf{t})$. We have taken the vector $\mathbf{t}' = (t_0 = 8,\ t_1 = 42,\ t_2 = 36,\ t_3 = 6,\ t_4 = 16,\ t_5 = 22)$.

We repeat the process and add the non-active arc $(1,5)$ to Λ. We solve $\mathsf{CTIME}(G,\Lambda)$ and retrieve a solution $\mathbf{t}' = (t_0 = 8,\ t_1 = 18,\ t_2 = 36,\ t_3 = 6,\ t_4 = 16,\ t_5 = 26)$. Then we solve $\mathsf{MCF}(G,\ f,\ \mathbf{t}')$ and we obtain the solution depicted in Figure 3.21 which involves a single vehicle.

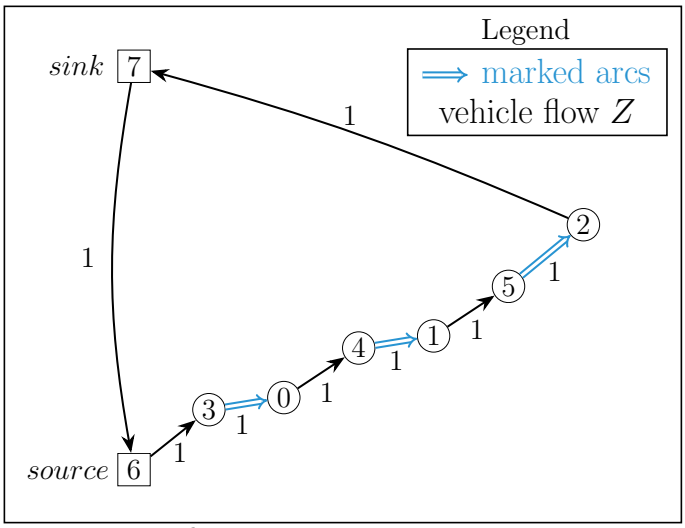


Figure 3.21: A $\mathsf{MCF}(G,f,\mathbf{t}')$ solution over a subgraph of $Aux(G,\ f,\ \mathbf{t})$. We have taken the vector $\mathbf{t}' = (t_0 = 8,\ t_1 = 18,\ t_2 = 36,\ t_3 = 6,\ t_4 = 16,\ t_5 = 26)$.

Finally, we arrive at the end of the process because it is not possible to add more non-active arcs to Λ and keep a feasible constraint system $\mathsf{CTIME}(G,\Lambda)$. □

Numerical Experiments

We performed experiments with the purpose of getting an evaluation of the difference, namely with regard to the number of vehicles, between the value provided by the projected PIRP model and the IRP solution obtained after performing the "lift" process. So, Table 3.2 provides us, for the instances described in Table 3.1, with the values **G4**, **T4**, and **V4** computed with the PIRP model. Column **MC** displays the cost of the solution computed with Algorithm 6, **TMC** indicates the running time in seconds that was spent in the computation of **MC**, **VMC** is the number of vehicles used in the solution with cost **MC**, and column **ITER** shows the number of iterations in the main loop of Algorithm 6 to compute the value **MC**. Missing values are indicated by a hyphen symbol -, and correspond to PIRP instances for which the WLIFT(G, f) MILP is unfeasible.

Table 3.2: Numerical results for the WLIFT(G, f) MILP, and the Monotonic Cover algorithm.

Id	n	m	κ	Ω	G4	V4	WLIFT	TWLIFT	MC	TMC	VMC	ITER
1	20	78	2	324	2110.85	3	1	0.01	3097.00	0.213	5	10
2	20	65	5	400	1196.10	3	1	0.01	1294.70	0.131	3	5
3	20	77	10	440	854.83	2	1	0.01	1504.25	0.425	3	19
4	20	75	2	680	3805.81	3	1	0.03	5958.00	2.651	7	43
5	20	50	5	603	2354.43	3	1	0.06	3074.30	2.176	4	33
6	20	57	10	840	1532.38	2	1	0.01	2655.40	0.321	4	17
7	20	62	5	420	2727.30	4	1	0.03	3963.70	1.623	7	39
8	50	163	2	460	15561.30	17	1	0.15	20952.00	811.539	31	294
9	50	155	5	390	4326.10	7	0	0.01	-	-	-	-
10	50	149	10	440	7966.03	6	0	0.03	-	-	-	-
11	50	146	20	436	1840.17	3	0	0.01	-	-	-	-
12	50	175	2	728	6976.11	5	1	0.03	11745.00	38.117	17	133
13	50	217	5	912	1643.76	2	1	0.01	3619.25	2.901	4	48
14	50	154	10	1040	2643.76	3	1	0.03	6134.03	21.542	10	108
15	100	363	2	336	17179.00	22	1	0.59	23469.00	271.212	37	187
16	100	236	5	516	4826.24	8	1	0.29	9190.75	340.001	24	244
17	100	289	10	432	3272.98	4	0	0.01	-	-	-	-
18	100	419	2	1032	20219.30	10	1	0.14	54691.00	4495.43	70	977
19	100	327	5	552	5944.23	7	1	0.01	17026.20	484.001	32	357
20	100	313	10	712	6091.24	4	1	0.01	11809.50	62.124	18	139

Comments. From Table 3.2, we can check above Monotonic Cover algorithm finds feasible IRP solutions for all the weak-lift-consistent instances (i.e., instances with 1 in the **WLIFT** column). However, by comparing values in columns **V4** and **VMC**, we can see that most of the time the solutions found by the Monotonic Cover algorithm involve a bigger number of vehicles than the estimated number by the PIRP model.

3.3.3 A More Efficient Path-Concatenate Algorithm

If we analyze previous Algorithm 6, we can observe that choosing a non-active arc with a negative reduced cost in the $\mathsf{COVER}(G,\ f,\ \Lambda)$ model is not enough to guarantee an improvement in the cost solution, because the inclusion of a new arc a in the collection Λ may give a different $\mathsf{CTIME}(\Lambda)$ solution, and this in turn may yield a $\mathsf{COVER}(G,\ f,\ \Lambda \cup \{a\})$ model with a worse cost optimal solution. We may try to avoid this problem by replacing the step 7 of Algorithm 6 in the following way:

Step 7′: Search for an arc $u = (p, q)$ such that $\lambda_p - \lambda_1 + \mu_u - cost_{Cover(G,f)}(u) > \varepsilon$, CTIME($\Lambda \cup \{u\}$) remains feasible, and $\mathsf{COVER}(G,\ f,\Lambda \cup \{u\})$ has and optimal solution with cost better than or equal to the cost of $\mathsf{COVER}(G,\ f,\Lambda)$.

Although such a a modification avoids previous issue, it has the following two drawbacks.

- The number of non-active arcs u such that $\lambda_p - \lambda_1 + \mu_u - cost_{Cover(G,f)}(u) > \varepsilon$ may be huge (see column **ITER** of Table 3.2), and for each of those arcs u we would need to solve not only the corresponding linear program $\mathsf{CTIME}(\Lambda \cup \{u\})$ but also the linear program $\mathsf{COVER}(G,\ f,\Lambda \cup \{u\})$. As a result, we obtain a slower Monotonic Cover algorithm.
- If current collection Λ is a local optimal collection (i.e., if for all $u \in Cover(G, f) \setminus \Lambda$, such that $\lambda_p - \lambda_1 + \mu_u - cost_{Cover(G,f)} > \varepsilon$ and $\mathsf{CTIME}(\Lambda \cup \{u\})$ is feasible, we have that $\mathsf{COVER}(G,\ f,\Lambda \cup \{u\})$ has a worse optimal solution cost than $\mathsf{COVER}(G,\ f,\Lambda)$), then we would end solving all of the related linear programs $\mathsf{COVER}(G,\ f,\Lambda \cup \{u\})$ only to find that the cost has not been improved.

The algorithm which we are going to present here works in a more empirical way, and uses a resolution of the $\mathsf{CTIME}(\Lambda)$ constraint system through Bellman algorithm in order to deduce which arc (p, q), with $p = (x_1, a_1, m_1, -)$ and $q = (x_2, a_2, m_2, +)$ has to be inserted in Λ. Its intuition is very simple: we do as if we were dealing with a collection of paths, supported by the arcs of current collection Λ, which are going to be followed by the vehicles between the time when they start loading some item and the time when they come back with empty load to the depot vertex d. Then we try to concatenate two among those paths, in such a way that it will be possible for a same vehicle to follow those two paths and that resulting path is going to be the shortest possible.

More precisely, we proceed as in Section 6 while following two steps:

- The **first step** works exactly as in Section 3.3.2: we solve $\mathsf{WLIFT}(G,f)$ and derive vectors $\mathbf{z}$, $\mathbf{z}^*$ which will remain unchanged during all the process. We initialize Λ with the arcs of $Cover(G,\ f)$ which correspond to nun-null $\mathbf{z}$, $\mathbf{z}^*$ values.

- Then the **second step** works as follows:

  - Current collection Λ being given, we solve $\mathsf{CTIME}(\Lambda)$ through Bellman algorithm for the search of a longest path in an acyclic digraph, and get, for any vertex $p = (x, a, m, \varepsilon) \in X^*$ a time window $[Min_p, Max_p]$, with:

    * $Min_p \geq time(d, x)$;
    * $Max_p \geq \Omega - time(x, d)$.

  - Then we identify vertices $p_1 = (x_1, a_1, m_1, +)$ such that $Min_{p_1} = time(d, x_1)$ and denote by $StartPath$ resulting vertex set. By the same way, we identify vertices $p_2 = (x_2, a_2, m_2, -)$ such that $Max_{p_2} = \Omega - time(x_2, d)$ and denote by $EndPath$ resulting vertex set. The meaning of those two vertex subsets is that they identify the vertices which may respectively correspond to the first time when a vehicle loads some item and the last time when it is unloaded some item.

  - Then we select $p_2 = (x_2, a_2, m_2, -) \in EndPath$ and $p_1 = (x_1, a_1, m_1, +) \in StartPath$ such that $Min_{p_2} + \Omega - Max_{p_1}$ does not exceed Ω, which also means $Min_{p_2} \leq Max_{p_1}$, and such that $Max_{p_1} - Min_{p_2}$ is the largest possible.

  - We insert arc (p_2, p_1) into Λ, update the time windows induced by the constraint system $\mathsf{CTIME}(\Lambda)$, and get some feasible solution $\mathbf{t}$ of $\mathsf{CTIME}(\Lambda)$. We set $\Lambda(t) = \{(p, q) \in A(Cover) : t_q \geq t_p + time(x(p, x(q)))\}$.

  - Finally we solve the $\mathsf{COVER}(G,\ f,\ \mathbf{t})$ Minimum Cost Flow problem, and update accordingly the best solution $\mathbf{H}$ ever found.

  We stop when it is not possible to perform above process, that means when it is not possible to compute p_1 and p_2.

The above process may be set in an algorithmic format as follows.

Algorithm 7: Path-Concatenate algorithm.

input : An item flow f such that WLIFT(G, f) is feasible.

output: A TEN IRP solution ($\mathbf{H}_{cur}$, $\mathbf{h}_{cur}$) over the digraph $Aux(G,f,\mathbf{t})$ defined in the proof of Theorem 3.2.

1 Solve WLIFT(G, f) MILP and retrieve vectors $\mathbf{z}$, $\mathbf{z}^*$ and $\boldsymbol{\ell}$ which will remain unchanged during all the process

2 Λ is the set of all arcs of $Cover(G, f)$ which correspond to non null $\mathbf{z}$, $\mathbf{z}^*$ values.

3 Compute time windows $[Min_p, Max_p]$ for $p \in X^*$

4 Solve the Minimum Cost Flow model COVER(G, f, Λ) and derive an initial solution ($\mathbf{H}_{cur}$, $\mathbf{h}_{cur}$).

5 `stop` ← `False`;

6 **while not** `stop` **do**

7 Compute the $StartPath$ and $EndPath$ subsets;

8 **if** one of those subsets is empty **then**

9 `stop` ← `True`;

10 **else**

11 Search for $p_2 = (x_2, a_2, m_2, -) \in EndPath$ and $p_1 = (x_1, a_1, m_1, +) \in StartPath$ such that $Min_{p_2} \leq Max_{p_1}$ and $Max_{p_1} - Min_{p_2}$ is the largest possible ;

12 **if** Fail(Search) **then**

13 `stop` ← `True`;

14 **else**

15 $\Lambda \leftarrow \Lambda + \{(p_2, p_1)\}$;

16 Update the time windows $[Min_p, Max_p]$, $p \in X^*$ and retrieve some feasible solution $\mathbf{t}$ of CTIME(Λ);

17 $\Lambda(\mathbf{t}) \leftarrow \{(p,q) \in A(Cover) : t_q \geq t_p + time(x(p), x(q))\}$;

18 Solve COVER(G, f, $\Lambda(\mathbf{t})$): if the value of resulting solution $\mathbf{Z}$ is better than the cost of ($\mathbf{H}_{cur}$, $\mathbf{h}_{cur}$) then update ($\mathbf{H}_{cur}$, $\mathbf{h}_{cur}$).

19 Return ($\mathbf{H}_{cur}$, $\mathbf{h}_{cur}$);

As the numerical test are going to make appear, this algorithm is significantly more efficient than the previous one of Section 3.3.2.

Implementation Details

Of course, there are different ways of implementing the heuristic described in Algorithm 7. In this section we describe a four-step implementation for constructing vehicle paths that are compatible with a computed time-feasible items flow f. The proposed implementation starts by associating with any vertex u of $Weak(G, f)$ a time window $[Min_u, Max_u]$, and uses an initial collection Γ of non-incident arcs as "seeds" for constructing vehicle paths. At the general step, it considers arc costs, time window values, and uses constraint propagation to select an arc $u = (p, q)$ of $Cover(G, f)$ and merge two paths of Γ into a single path. The selected arc $u = (p, q)$ is added to current collection Λ, and we solve the resulting $\mathsf{COVER}(G, f, \Lambda \cup \{u\})$. At the end, we otput the best cost $\mathsf{COVER}(G, f, \Lambda)$ solution found. We describe next the initialization and the four stages of the proposed implementation.

Initialization. We start from a $\mathsf{WLIFT}(G, f)$ constraint system set on a digraph $Weak(G, f)$ corresponding to a PIRP solution (F, f) and with a feasible solution $(\mathbf{z}, \boldsymbol{\ell}, \mathbf{z}^*, \mathbf{t})$. We construct digraph $Aux(G, f, \mathbf{t})$ and an auxiliary digraph G_{time} with vertex set $X_{time} = X^*$, and arc set $A_{time} = \{a \in Copy(A) : z^*_a > 0\} \cup \{a \in Router : z_a > 0\}$. This digraph G_{time} will be used to keep track of the time constraints related to the constructed solutions. Also, given $x, y \in X$, for every $x' \in X^*(x)$, and for every $y' \in X^*(y)$, we define $time(x', y') = time(x', y) = time(x, y') = time(x, y)$. This function will be used at the fourth stage of the heuristic.

For every $x \in X_{time}$ we associate a time window $[Min_x, Max_x]$, where Min_x (respectively Max_x) is the minimum (respectively the maximum) departure time from x that is compatible with $(\mathbf{z}, \boldsymbol{\ell}, \mathbf{z}^*, \mathbf{t})$; that is, Min_x, Max_x are values such that, if we fix the t_x variable value in the initial $\mathsf{WLIFT}(G, f)$ constraint system to a value in the interval $[Min_x, Max_x]$, resulting constraint system remains feasible, and $[Min_x, Max_x]$ is maximal with respect to set inclusion. We also define an initial collection of one-arc paths $\Gamma = \{(x, y) \in A_{time},$ such that $(x, y) \in Copy(A)$ and $z^*_{(x,y)} > 0\}$, and a boolean array `used_vertices` indexed over X_{time} and with all entries initialized to `false`.

First stage. We search for an arc $a = (x, y) \in Router$ with `used_vertices`[x] = `false`, and `used_vertices`[y] = `false`, and such that a is the unique arc with $z_a > 0$ connecting x to y. We merge the path $\Gamma_1 \in \Gamma$ ending at x with the path $\Gamma_2 \in \Gamma$ starting at y. Also, we set used `vertices`$[x]$ = `true`, and `used_vertices`$[y]$ = `true`. We repeat this process until no more of such arcs a can be found.

Second stage. We put all the arcs $a = (x, y) \in Router$ with $z_a > 0$ in a list `L`, and we sort those arcs (x, y) according to their corresponding $Min_x - Max_y$ values. Then, for every a in `L` (in the ordering given by `L`) with `used_vertices`[x] = `false`, and `used_vertices`[y] = `false` we merge the path $\Gamma_1 \in \Gamma$ ending at x with the path $\Gamma_2 \in \Gamma$ starting at y, and we set `used_vertices`[x] = `true`, and `used_vertices`[y] = `true`.

Third stage. We put all the arcs $a = (x, y) \in Router$ with $z_a = 0$ in a list `L`, and we sort those arcs (x, y) according to their corresponding $Min_x - Max_y$ values. Then, for every a in `L` (in the ordering given by `L`) with `used_vertices`[x] = `false`, and `used_vertices`[y] = `false` we add a to A_{time}, and propagate time constraints (i.e., we check the acyclicity of $A_{time} \cup \{a\}$, and update time window values). Let $\Gamma_1 \in \Gamma$ be the path ending at x, and $\Gamma_2 \in \Gamma$ be the path starting at y. Let x' be the first vertex of Γ_1, and y' be the last vertex of Γ_2. We have the two following cases.

- If G_{time} is acyclic, and for all $x \in X_{time}$ we have that $Min_x \leq Max_x$, and also we have that $time(d, x') + time(\Gamma_1) + time(x, y) + time(\Gamma_2) + time(y', d) \leq \Omega$, then we merge the path $\Gamma_1 \in \Gamma$ ending at x with the path $\Gamma_2 \in \Gamma$ starting at y, and we set `used_vertices`[x] = `true`, and `used_vertices`[y] = `true`. For all $x \in X^*$, we replace t_x by Min_x. Then we update $Aux(G, f, \mathbf{t})$, and we solve $\mathsf{MCF}(G, f, \mathbf{t})$.
- If G_{time} contains a cycle, or exists $x \in X_{time}$ such that $Min_x > Max_x$, or we have $time(d, x') + time(\Gamma_1) + time(x, y) + time(\Gamma_2) + time(y', d) > \Omega$, then the corresponding constraint system is infeasible. We remove a from A_{time}, and restore time window values.

Fourth stage. For every $x \in X_{time}$ such that x is the ending vertex of a path in Γ, and for every y such that y is the starting vertex of a path in Γ, we create a pair (x, y). We put those pairs in a list `L` and we sort those pairs (x, y) according to their corresponding $time(x, y)$ values. Then, for every pair (x, y) in `L` (in the ordering given by `L`) with `used_vertices`[x] = `false`, and `used_vertices`[y] = `false` we add (x, y) to A_{time}, and propagate time constraints (i.e., we check the acyclicity of $A_{time} \cup \{a\}$, and update time window values). Let $\Gamma_1 \in \Gamma$ be the path ending at x, and $\Gamma_2 \in \Gamma$ be the path starting at y. Let x' be the first vertex of Γ_1, and y' be the last vertex of $\Gamma 2$. We have the two following cases.

- If G_{time} is acyclic, and for all $x \in X_{time}$ we have that $Min_x \leq Max_x$, and also we have that $time(d, x') + time(\Gamma_1) + time(x, y) + time(\Gamma_2) + time(y', d) \leq \Omega$, then we merge the path $\Gamma_1 \in \Gamma$ ending at x with the path $\Gamma_2 \in \Gamma$ starting at y, and we set `used_vertices`$[x]$ = `true`, and `used_vertices`$[y]$ = `true`. For all $x \in X^*$, we replace t_x by Min_x. Then we update $Aux(G, f, \mathbf{t})$, and we solve $\mathsf{MCF}(G, f, \mathbf{t})$.

- If G_{time} contains a cycle, or exists $x \in X_{time}$ such that $Min_x > Max_x$, or we have $time(d, x') + time(\Gamma_1) + time(x, y) + time(\Gamma_2) + time(y', d) > \Omega$, then the corresponding constraint system is infeasible. We remove a from A_{time}, and restore time window values.

Output. Return the best cost $\mathsf{MCF}(G, f, \mathbf{t})$ solution found.

Note that, once we have checked the acyclicity of G_{time}, the above constraint propagation process for updating time windows can be performed in time $O\big(\text{size}(G_{time})\big)$ in the following way.

Propagation of Min values. Sort X_{time} in a topological ordering of G_{time}. For every $x \in X_{time}$, and every $a = (x, y) \in \partial^+_{G_{time}}(x)$, if $Min_y < Min_x + time(x, y)$ then set $Min_y = Min_x + time(x, y)$.

Propagation of Max values. Sort X_{time} in a reverse topological ordering of G_{time}. For every $y \in X_{time}$, and every $a = (x, y) \in \partial^-_{G_{time}}(y)$, if $Max_x > Max_y - time(x, y)$ then set $Max_x = Max_y - time(x, y)$.

Numerical Experiments

Table 3.3 provides us, for the instances described in Table 3.1, with the values **G4**, **T4**, and **V4** computed with the PIRP model; and with the values **MC**, **TMC**, and **VMC** computed with Algorithm 6. Column **PC** displays the cost of the solution computed with the above-described Path-Concatenate algorithm, **TPC** indicates the running time (in seconds) that was spent in the computation of **PC**, and **VPC** is the number of vehicles used in the solution with cost **PC**. Columns **TEN** and **V** contain reference values for the cost and number of vehicles of the TEN IRP model computed by CPLEX 12.10 in one hour. Again, missing values are indicated by a hyphen symbol - and correspond, either to PIRP instances for which the $\mathsf{WLIFT}(G, f)$ MILP is

unfeasible or TEN IRP instances for which the solver could not find any feasible solution within the time limit.

Table 3.3: Numerical results for the Monotonic Cover algorithm, and the Path-Concatenate algorithm.

Id	n	m	κ	Ω	G4	V4	MC	TMC	VMC	PC	TPC	VPC	TEN	V
1	20	78	2	324	2110.85	3	3097.00	0.213	5	3097.00	0.003	5	2633.00	4
2	20	65	5	400	1196.10	3	1294.70	0.131	3	1294.70	0.009	3	1282.70	3
3	20	77	10	440	854.83	2	1504.25	0.425	3	1504.85	0.045	3	1123.25	2
4	20	75	2	680	3805.81	3	5958.00	2.651	7	4537.00	0.224	4	4215.00	3
5	20	50	5	603	2354.43	3	3074.3	2.176	4	2474.70	0.163	3	2469.00	3
6	20	57	10	840	1532.38	2	2655.40	0.321	4	1822.80	0.039	2	1823.70	2
7	20	62	5	420	2727.30	4	3963.70	1.623	7	3266.50	0.142	5	2898.10	4
8	50	163	2	460	15561.30	17	20952.00	811.539	31	18435.00	18.622	24	17006.00	20
9	50	155	5	390	4326.10	7	-	-	-	-	-	-	5023.70	9
10	50	149	10	440	7966.03	6	-	-	-	-	-	-	8820.50	8
11	50	146	20	436	1840.17	3	-	-	-	-	-	-	2670.93	6
12	50	175	2	728	6976.11	5	11745.00	38.117	17	7589.00	4.622	6	-	-
13	50	217	5	912	1643.76	2	3619.25	2.901	4	2234.25	0.189	2	-	-
14	50	154	10	1040	2643.76	3	6134.03	21.542	10	3425.83	1.807	4	-	-
15	100	363	2	336	17179.00	22	23469.00	271.212	37	21623.00	9.476	32	19383.00	27
16	100	236	5	516	4826.24	8	9190.75	340.001	24	6656.35	20.588	14	31881.35	86
17	100	289	10	432	3272.98	4	-	-	-	-	-	-	3808.15	5
18	100	419	2	1032	20219.30	10	54691.00	4495.430	70	22692.00	217.848	13	-	-
19	100	327	5	552	5944.23	7	17026.20	484.001	32	7556.70	33.846	10	-	-
20	100	313	10	712	6091.24	4	11809.50	62.124	18	7548.00	6.679	7	-	-

Comments: In Table 3.3, we can see that the Path Concatenate heuristic presented in this section finds feasible IRP solutions for all the weak-lift-consistent instances. Also, by comparing values in columns **MC** and **PC**, **TMC** and **TPC**, and **VMC** and **VPC**, we can see that most of the time the solutions found by the Path Concatenate heuristic have lower costs, involve less vehicles, and have required lower running times to be computed than the solutions found by the Monotonic Cover algorithm.

3.4 Conclusion

In this chapter we have proposed several algorithms for handling the two Lift problems introduced in Section 2.5.1.

We have proposed a MILP formulation for solving the Strong Lift Problem in an exact way. However, the numerical experiments have suggested that even when we consider both the Extended-Subtour constraint and the Feasible-Path constraints, the computed PIRP solutions usually yield an infeasible Strong Lift Problem. Then, we have either to search for additional constraints that may increase the probability of obtaining PIRP solutions which yield feasible Strong Lift Problems, or we have to accept a deterioration of the cost estimated by the PIRP model.

Also, we have studied the Partial Lift Problem and we have proposed a Weak/ Cover decomposition approach for handling it in a flexible way. We have introduced the concept of weak-lift-consistence and provided two heuristic algorithms for lifting item flows which are weak-lift-consistent.

Finally, the resolution approach described here remains somewhat heavy. It requires an extended use of the mixed integer linear programming machinery, and so is hardly fitted to situations when the size of the base digraph G is large. It follows that a last issue is about the way our almost exact algorithms may be turned into fast running heuristic algorithms, which can run without the help of any specific and possibly expensive library.

Part III

The Pickup-and-Delivery Problem with Transfers and Time Horizon

Chapter 4

The Virtual Path Problem: Application to the PDPT

Let G be an acyclic digraph with an underlying constraint system. Suppose we are allowed to introduce new arcs between some pairs of vertices of G at the expense of introducing new constraints in the constraint system. In this chapter we introduce the Virtual Path Problem, which is about the construction of a directed path between two given vertices of G, while optimizing some objective function and while maintaining the feasibility of the underlying constraint system related to G.

In Section 4.2 we describe formally the Virtual Path Problem and we present the Virtual A* algorithm for solving it in an exact way.

Then, in Section 4.3 we introduce the 1-Request Insertion PDPT which is about the way that one additional request can be inserted into the current solution of a Pickup-and-Delivery Problem with Transfers (PDPT) instance. We show that this insertion problem can be seen as particular case of the Virtual Path Problem and then we propose the Virtual A* algorithm as an exact algorithm for solving the problem. We also propose a heuristic based on Dijkstra's algorithm and constraint propagation. We compare the quality of the found solutions and the performance of both algorithms over several data sets of pseudorandom instances.

In Section 4.4 we show how to combine the single insertion algorithms developed in Section 4.3 with some classical metaheuristics, to insert multiple requests into a PDPT schedule.

use those single insertion algorithms to handle PDPT instances with multiple requests. We also present some common algorithms based on random/local search to explore the solution space and we conclude with some numerical experiments.

4.1 Introduction

A standard Pickup-and-Delivery Problem (PDP) most often involves a finite set V of identical vehicles $v_1, v_2, \ldots, v_{|V|}$ with capacity κ, which must be scheduled in order to perform a set of pickup-and-delivery tasks inside some transit network G. Time windows may be involved, but in most cases, a *time horizon* $[0, \Omega]$ is imposed for the whole schedule. Vehicles must meet a set R of requests: a request $r \in R$ consists in an origin o_r, a destination d_r, a load ℓ_r, and has to be served by exactly one vehicle.

In the PDP with transfers (PDPT), a request may be served by several vehicles, in the sense that load ℓ_r may start from origin o_r with some vehicle v_1 and next shift to some vehicle v_2 at some relay vertex x, and so on, until reaching destination d_r into some vehicle v_p. Depending on the context, a transfer from v_1 to v_2 may take different forms: one may impose either vehicles to meet (*strong synchronization constraint*) at relay vertex x or only forbid vehicle v_2 from leaving x before the arrival of v_1 (*weak synchronization constraint*). In other words, a weak synchronization constraint implies that v_1 and v_2 are obliged to meet only when v_2 arrives first to the relay vertex x.

Due to those synchronization constraints, a convenient way to model PDPT is through the use of Time-Expanded networks (see Section 1.2.3, or for example Godinho et al. (2014) [106] and Bsaybes et al. (2019) [41]). Given a digraph $G = (X, A)$ and a time horizon $[0, \Omega]$, we construct a Time-Expanded network $G^\Omega = (X^\Omega, A^\Omega)$ (see Figure 4.1) whose vertex set X^Ω consists of an auxiliary *source* vertex $\hat{s}$, an auxiliary *sink* vertex $\hat{p}$, and a copy (x, t_i) for every vertex x of G and for any time value $t_i \in [0, \Omega]$ (note that in practice, we usually consider only a finite set of relevant time values t_i). We represent any feasible move along an arc of G, from a vertex x to a vertex y between time t_i and time $t_i + \delta$ (where $\delta \geq 0$ is the time necessary for moving from x to y), by some arc $\big((x, t_i), (y, t_i + \delta)\big)$. If $x = y$ then such a move becomes a *waiting* move. Then, we represent vehicle circulation by a unique integral flow vector $\mathbf{H}$ indexed over the set A^Ω of the Time-Expanded network arcs, and request circulation comes as a multicommodity flow $\{h_r : A^\Omega \to \mathbb{R}, r \in R\}$ such that, for every arc $a \in A^\Omega$ the sum $\Sigma_{r \in R} h_r(a)$ is less than or equal to the value $\kappa \cdot H_a$, where H_a is the entry of vector $\mathbf{H}$ corresponding to arc a.

However, resulting models are not very well-fitted to numerical handling, both because of their size and because the PDPT tends to arise in dynamic contexts, with requests not completely known in advance and which must be managed in a flexible way.

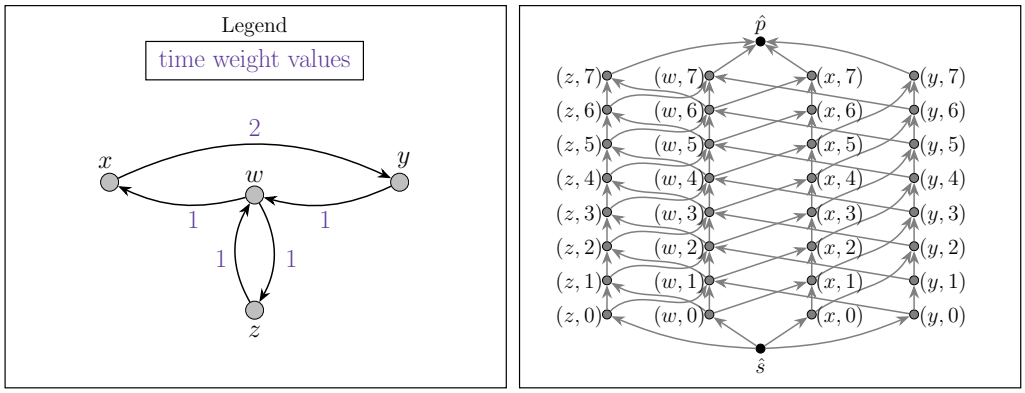


Figure 4.1: Construction of a Time-Expanded network. (a) A digraph $G = (X = \{w, x, y, z\}, A = \{(z, w), (w, x), (x, y), (y, w), (w, z)\})$. Arc labels indicate the time necessary for traversing each arc. (b) The Time-Expanded network G^{Ω} with time horizon $[0, \Omega = 7]$ constructed from (a). Because the time weights are integral, we only need to consider the integral values t_i in the interval $[0, 7]$.

It results that most often, the PDPT is handled in a heuristic way and a common strategy is to rely on an insertion (or build and destroy) approach: requests are successively inserted into some current schedule and possibly removed and reinserted for trying to improve the cost.

According to this paradigm, the key issue becomes the related insertion process. In case no transfer is allowed, the insertion is trivial in the sense that it can be performed through enumeration. It is not the case when transfers are allowed. As a matter of fact, inserting a request r becomes then significantly more difficult than simply searching for some path from an origin o_r to a destination d_r inside some ad hoc network (or even Time-Expanded network) because synchronization (strong or weak) constraints requirements tend to impact the whole current schedule.

So, the main purpose of this chapter is to thoroughly study the one request insertion problem that occurs as a part of the Pickup-and-Delivery Problem with Transfers (with time horizon), and that we will denote by 1-Request Insertion PDPT.

For this, we introduce a more general problematic that we call the Virtual Path Problem and we give a generic framework for handling it through a combination of constraint propagation techniques and an A*-like algorithm (see Section 1.2.1, or for example, Nilsson (1980) [166] and Russell and Norvig (2021) [188]).

Then, we adapt the algorithm for the Virtual Path Problem to derive an algorithm for solving the 1-Request Insertion PDPT in an exact way, without imposing any restriction neither on the number of transfers nor on the characteristics on the transfer parameters. In order to fit with the purpose of practical use in realistic dynamic contexts, we also describe a heuristic algorithm, which relies on Dijkstra's algorithm (augmented with a "closure" mechanism) and constraint propagation.

Finally, we show how to insert multiple requests into a PDPT schedule by combining some classical metaheuristics with the single insertion algorithms that we have developed.

Relation with the Existent Literature

Among the papers mentioned in Section 1.1, the works of Bouros et al. (2011) [36] and Lehuédé et al. (2013) [156] are the closest ones to the contributions of this chapter. Both rely on the construction of auxiliary graphs, which represent the current state of vehicle paths together with some specific constraints related to transfers. Still [36] does not care of time-feasibility (i.e., time windows) nor the impact of restrictions imposed to transfers. At the opposite, [156] is a theoretical contribution which focuses on the complexity of testing the feasibility of an insertion (with one transfer at most), after some preprocessing that allows the management of slack time variables. The work we present here it is in the middle of those works: while we also rely on precomputation of weight path values, we relax the restriction on the number of relay vertices used in a transfer, impose no restriction on the number of transfers, and propose both exact and heuristic algorithms for the computation of a best feasible insertion for a given request. Also, we analyze through numerical experiments, the impact on both the behavior of the algorithms and the nature of the solutions for PDPT instances with single/multiple requests.

4.2 The Virtual Path Problem

We start from an acyclic digraph $H = (X, A)$ with vertex set X and arc set A. We denote by $<<^A$ the partial ordering over X that is induced by the arcs of H, and we suppose that we are provided with a unique minimal vertex $x^{\min}$ and a unique maximal vertex $x^{\max}$, both with respect to $<<^A$. A weight function $time : A(H) \to \mathbb{R}_+$ is associated with H. In addition, we are provided with a time horizon $[0, \Omega]$ and so, function $time$ induces the following system (CS) of constraints that involves a nonnegative time vector $\mathbf{t} = (t_x,\ x \in X)$.

$$\left.\begin{array}{ll} \bullet \text{ For any } a = (x_1, x_2) \in A, \text{ we have that } t_{x_2} \geq t_{x_1} + time(a). & \text{(CS1)} \\ \bullet \text{ For any } x \in X, \text{ we have that } t_x \leq \Omega. & \text{(CS2)} \end{array}\right\} \text{(CS)}$$

We suppose that (CS) is feasible and for any $x \in X$, we denote by $[Min_x, Max_x]$ the time window of feasible values associated to x.

We also are provided with two specific additional vertices $\hat{o}$ (origin) and $\hat{d}$ (destination) which are not in X, and with a collection Λ of paths Λ_k, $k = 1, \ldots, K$ in H. For $k = 1, \ldots, K$ the path Λ_k has a weight $time(\Lambda_k)$ which is induced by the function $time$, that is $time(\Lambda_k) := \sum_{a \in A(\Lambda_k)} time(a)$.

An example of the above-defined situation is illustrated in Figure 4.2.

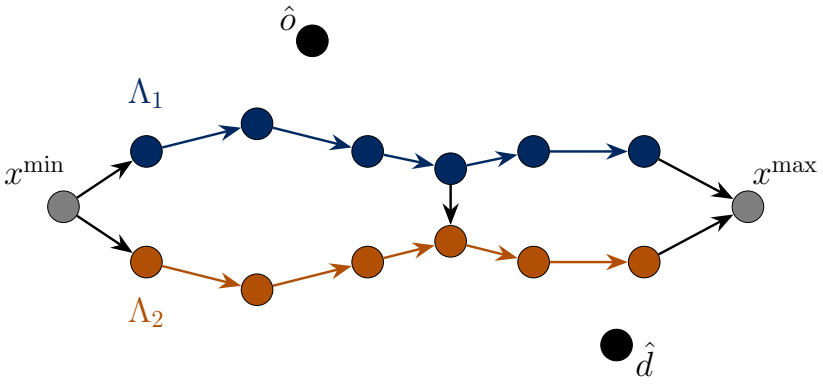


Figure 4.2: Two isolated vertices $\hat{o}$ and $\hat{d}$, and a connected digraph $H = (X, A)$ involving paths Λ_1 and Λ_2. Note that $\hat{o}$ and $\hat{d}$ are not vertices of H.

Our purpose is the computation of some kind of path from $\hat{o}$ to $\hat{d}$ which meets some constraints and minimizes some cost involving the weight of the path, the route collection Λ, and the time windows $[Min_x, Max_x]$, $x \in X$. Moving from $\hat{o}$ to $\hat{d}$ will involve arcs of A, which now are dubbed *real-arcs* and additional arcs which will be called virtual-arcs and which will behave as vertex transitions that impact the constraint system (CS). We describe now the construction of those virtual-arcs.

We start by defining $X^* = X \cup \{\hat{o}, \hat{d}\}$. Then, we define a *virtual-arc* as a 4-tuple $a^{virt} = \big(x, y, a_1 = (x_1, y_1), a_2 = (x_2, y_2)\big)$, where $(x, y) \in (X \cup \{\hat{o}\}) \times (X \cup \{\hat{d}\})$ is such that $(x, y) \notin A$, and $a_1, a_2 \in A \cup \{\texttt{Nil}\}$, (here, `Nil` is a new auxiliary arc). Any virtual-arc is provided with a nonnegative "length" 3-tuple $(\Delta, \Delta_1, \Delta_2)$ in such a way that:

- if $x \in X$, then $x = x_1$; otherwise if $x = \hat{o}$ then we are provided with an auxiliary deviation cost δ_2;
- if $y \in X$, then $y = y_2$; otherwise if $y = \hat{d}$ we are provided with an auxiliary deviation cost δ_1;
- if $a_1 = \texttt{Nil}$ then $a_2 \neq \texttt{Nil}$, $x = \hat{o}$, $y \neq \hat{d}$, and $\Delta = \Delta_1$;
- if $a_2 = \texttt{Nil}$ then $a_1 \neq \texttt{Nil}$, $y = \hat{d}$, $x \neq \hat{o}$, and $\Delta = \Delta_2$.

We say that x is the tail vertex of a^{virt}, that y is its head, that Δ is its length, and that Δ_1, Δ_2 are its *deviation coefficients*.

Remark 1. This notion of virtual-arc formalizes the notion of transfer, i.e., of "bridges" in digraph H from the tail of one arc to the head of another one, or from the source vertex $\hat{o}$ to the head of some arc, or from the tail of some arc to the sink vertex $\hat{d}$. A virtual-arc (x, y, a_1, a_2) represents a movement from x to y that introduces new constraints to (CS).

Then we consider a set A^{virt} of virtual-arcs and denote by $H^* = (X^*, A^* = A^{virt} \cup A)$ the structure which results from augmenting real-arc set A with virtual-arc set A^{virt}. We call H^* a *virtual digraph*. Please notice that this virtual digraph has to be understood more as a kind of transition system that impacts the constraint system (CS). Also note that this virtual digraph may contain cycles.

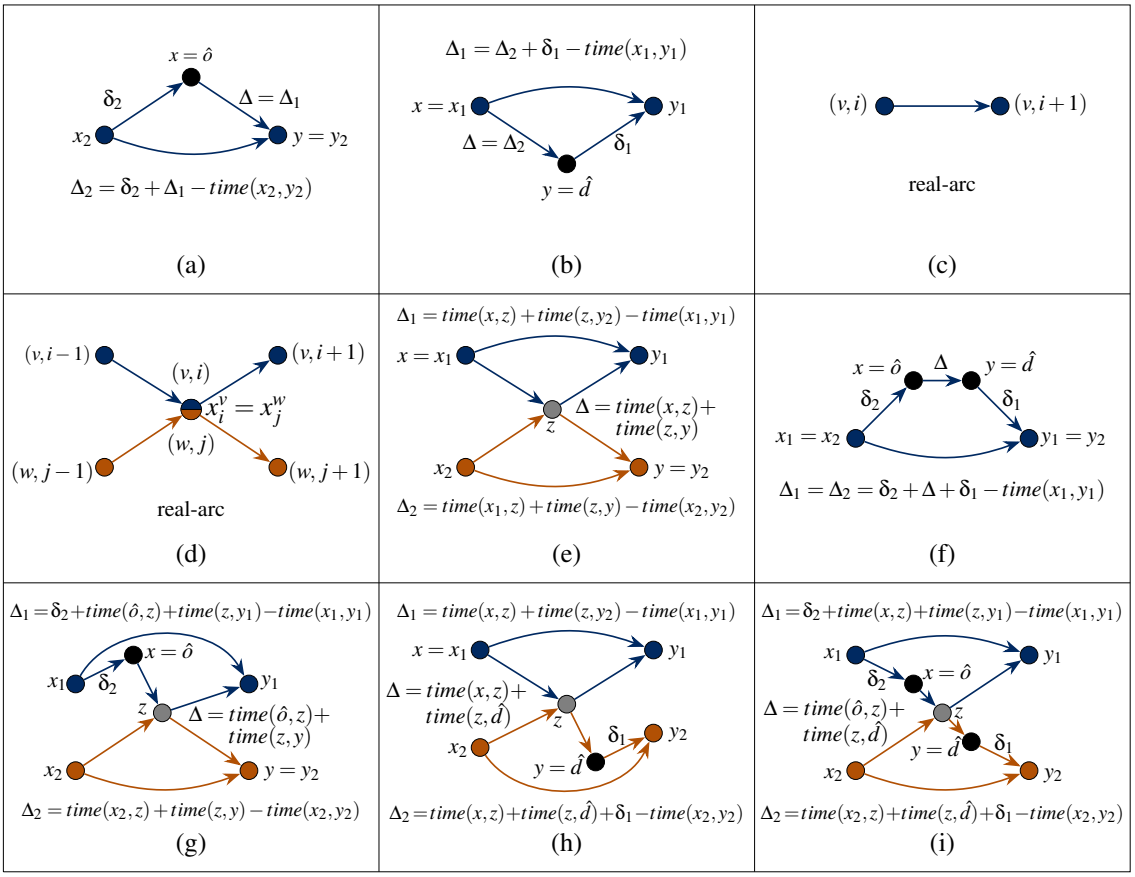


Figure 4.3: Deviation coefficients to construct the virtual graph arcs.

Running along virtual-arc (x, y, a_1, a_2) while moving from $\hat{o}$ to $\hat{d}$ means inserting the arc (x, y) as a real-arc of H^* with an impact on the constraint system (CS). Let us describe this impact.

Insertion of a virtual-arc $(x, y, a_1 = (x_1, y_1), a_2 = (x_2, y_2))$: (V1)

- It increases the values $time(a_1)$ and $time(a_2)$ respectively by Δ_1, and Δ_2. (see Figures 4.4-4.7). Note this can be understood intuitively as the introduction of two new constraints $t_{x_1} + time(a_1) + \Delta_1 \leq t_{y_1}$ and $t_{x_2} + time(a_2) + \Delta_2 \leq t_{y_2}$.
- It introduces a new real-arc (x, y) with $time((x, y)) = \Delta$ (see Figures 4.4-4.7). Again, this can be understood intuitively as the introduction of a new constraint $t_y \geq t_x + \Delta$.

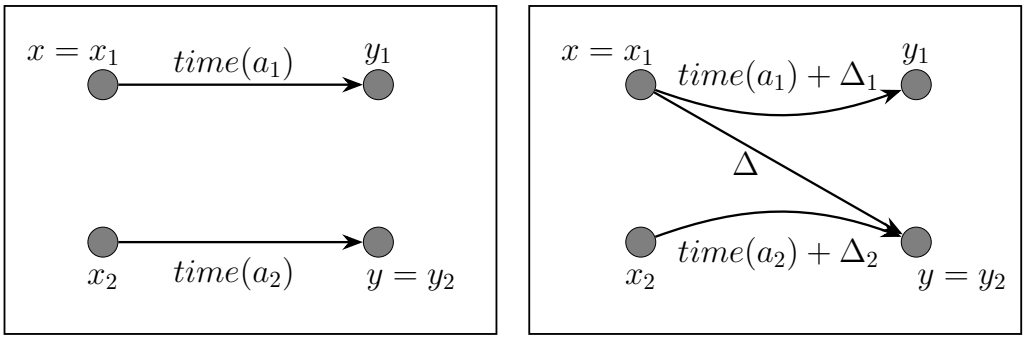


Figure 4.4: A virtual-arc $a^{virt} = (x, y, a_1, a_2)$ with $x, y \in X$ and $a_1, a_2 \in A$. (a) Components of a^{virt}. (b) The effect of adding the arc a^{virt} in (a) to the digraph H. It increases the values $time(a_1)$ and $time(a_2)$, by Δ_1 and Δ_2, respectively. It also creates a new real-arc (x, y) with $time((x, y)) = \Delta$.

- In case $a_1 = \texttt{Nil}$, then $x = \hat{o}$, $y \neq \hat{d}$, and it also creates a new real-arc from x_2 to $\hat{o}$ with $time((x_2, \hat{o})) = \delta_2$ (See Figure 4.5). Note this can be understood intuitively as the introduction of a new constraint $t_{x_2} + \delta_2 \leq t_{\hat{o}}$.

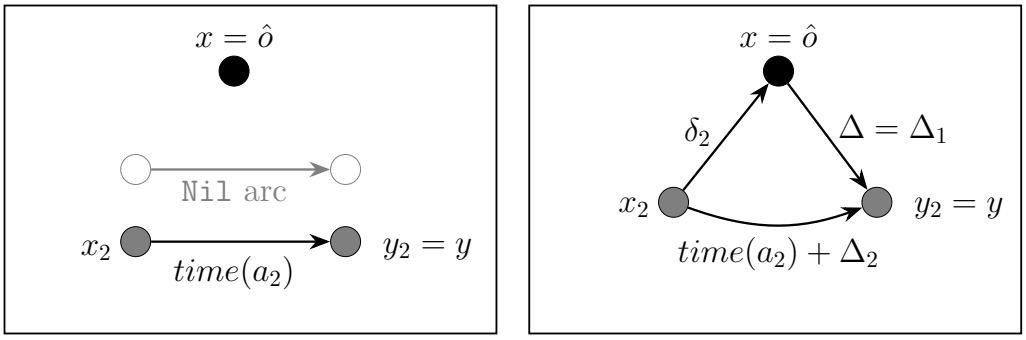
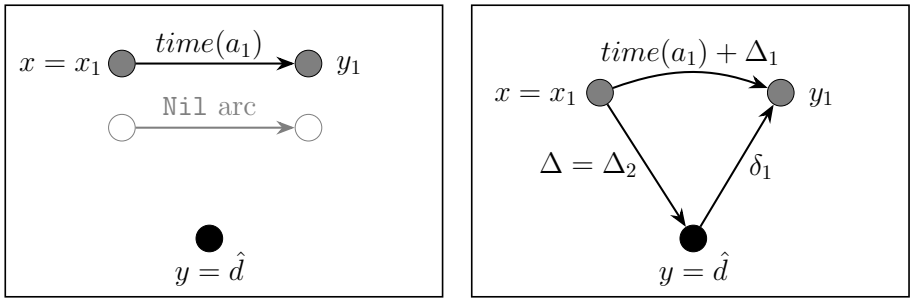


Figure 4.5: A virtual-arc $a^{virt} = (x, y, a_1, a_2)$ with $a_1 = \texttt{Nil}$ and $x = \hat{o}$. (a) Components of a^{virt}. (b) The effect of adding the arc a^{virt} in (a) to the digraph H. It increases the value $time(a_2)$ by Δ_2, creates a new real-arc $(\hat{o}, y)$ with $time((\hat{o}, y)) = \Delta = \Delta_1$, and creates a new real-arc $(x_2, \hat{o})$ with $time((x_2, \hat{o})) = \delta_2$.

- In case $x = \hat{o}$, $y \neq \hat{d}$, and $a_1 \neq \texttt{Nil}$, then it also creates a new real-arc from x_1 to $\hat{o}$, with $time((x_1, \hat{o})) = \delta_2$ (see Figure 4.3 (g)). Note this can be understood intuitively as the introduction of a new constraint $t_{x_1} + \delta_2 \leq t_{\hat{o}}$.

- In case $a_2 = \texttt{Nil}$, then $y = \hat{d}$, $x \neq \hat{o}$, and it also creates a new real-arc from $\hat{d}$ to y_1 with $time((\hat{d}, y_1)) = \delta_1$ (See Figure 4.6). This can be interpreted intuitively as the introduction of a constraint $t_{\hat{d}} + \delta_1 \leq t_{y_1}$.

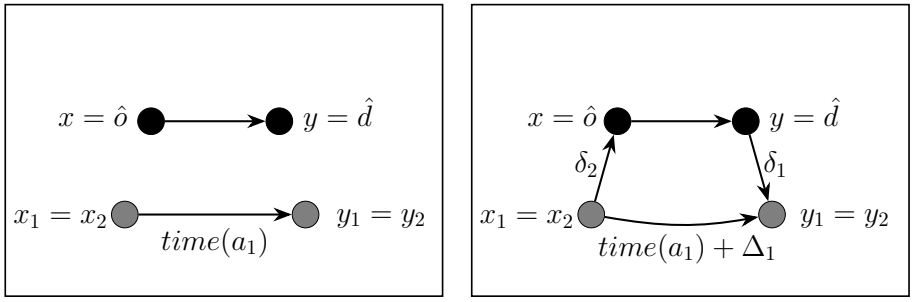


(a) (b)

Figure 4.6: A virtual-arc $a^{virt} = (x, y, a_1, a_2)$ with $a_2 = \texttt{Nil}$, $y = \hat{d}$, and $x \neq \hat{o}$. (a) Components of a^{virt}. (b) The effect of adding the arc a^{virt} in (a) to digraph H. It increases the value $time(a_1)$ by Δ_1, creates a new real-arc $(x_1, \hat{d})$ with $time((x_1, \hat{d})) = \Delta = \Delta_2$, and creates a new real-arc $(\hat{d}, y_1)$ with $time((\hat{d}, y_1)) = \delta_1$.

- In case $y = \hat{d}$, $x \neq \hat{o}$, and $a_2 \neq \texttt{Nil}$, then it also creates a new real-arc from $\hat{d}$ to y_2, with $time((\hat{d}, y_2)) = \delta_1$ (See Figure 4.3 (h)). Note this can be understood intuitively as the introduction of a new constraint $t_{\hat{d}} + \delta_1 \leq t_{y_2}$.

- In case $x = \hat{o}$, $y = \hat{d}$, and $a_1 = a_2 \in A$, then it also creates a new real-arc $(x_1, \hat{o})$ with $time((x_1, \hat{o})) = \delta_2$, a real-arc $(\hat{d}, y_1)$ with $time((\hat{d}, y_1)) = \delta_1$ (See Figure 4.7). This can be interpreted intuitively as the introduction of the two constraints $t_{x_1} + \delta_2 \leq t_{\hat{o}}$, and $t_{\hat{d}} + \delta_1 \leq t_{y_1}$.



(a) (b)

Figure 4.7: A virtual-arc $a^{virt} = (x, y, a_1, a_2)$ with $x = \hat{o}$, $y = \hat{d}$, and $a_1 = a_2 \in A$. (a) Components of a^{virt}. (b) The effect of adding the arc a^{virt} in (a) to digraph H. It increases the value $time(a_1)$ by Δ_1, creates a new real-arc $(\hat{o}, \hat{d})$ with $time((\hat{o}, \hat{d})) = \Delta$, creates a new real-arc $(x_2, \hat{o})$ with $time((x_2, \hat{o})) = \delta_2$, and creates a new real-arc $(\hat{d}, y_1)$ with $time((\hat{d}, y_1)) = \delta_1$.

- In case $x = \hat{o}$, $y = \hat{d}$, and $a_1, a_2 \in A$ with $a_1 \neq a_2$, then it also creates a new real-arc $(x_1, \hat{o})$ with $time((x_1, \hat{o})) = \delta_2$ and a real-arc $(\hat{d}, y_2)$ with $time((\hat{d}, y_2)) = \delta_1$ (See Figure 4.3 (i)). This can be interpreted intuitively as the introduction of the two constraints $t_{x_1} + \delta_2 \leq t_{\hat{o}}$, and $t_{\hat{d}} + \delta_1 \leq t_{y_2}$.

As we can see, such an insertion impacts both the time windows $[Min_x, Max_x]$, $x \in X$ and the weights $time(\Lambda_k)$ of paths Λ_k, $k = 1, \ldots, K$. In order to make (CS) consistent, this leads us to impose to any virtual-arc $(n, m, a_1 = (x_1, y_1), a_2 = (x_2, y_2))$:

- no path from y_2 to x_1 exists in H. (J1)

Now, we proceed to define our Virtual Path Problem. We want to compute a virtual path π which connects $\hat{o}$ to $\hat{d}$ and involves real-arcs and virtual-arcs, while maintaining the feasibility of constraint system (CS). This imposes that the new real-arcs resulting from the insertion of the virtual-arcs of π does not create any cycle. More precisely, let π be a virtual path starting at $\hat{o}$, if we denote by $A^{add}(\pi)$ the set of real-arcs which were created by considering the virtual-arcs of π, then the set $A \cup A^{add}(\pi)$ must contain no cycle. In case the resulting constraint system CS(π) is feasible, then for all $x \in X^*$ we denote by $[Min_x^\pi, Max_x^\pi]$ the resulting time windows and we denote by Λ_k^π the path Λ_k after being modified by the creation of the virtual path π.

As for the cost of virtual path π, it may involve the weight $time(\pi) = \sum_{a \in A(\pi)} time(a)$ (i.e., the "distance" run by the request), the weights $time(\Lambda_k^\pi)$, $k = 1, \ldots, K$ (vehicle running costs), and the time when all the served requests may be achieved (user ride time).

Hence we define $cost(\pi) := \alpha \cdot time(\pi) + \sum_{k=1}^{K} \beta_k \cdot time(\Lambda_k^\pi) + \sum_{x \in X^*} \gamma_x \cdot Min_x^\pi$, where α, β_k, γ_n are nonnegative scaling coefficients for $k = 1, \ldots, K$, and $x \in X^*$. We denote by X^{user} the subset of X defined by the nodes x such that $\gamma_x > 0$. We summarize this as below.

***Virtual Path Problem*:** Compute a virtual path π from $\hat{o}$ to $\hat{d}$ in the virtual digraph H^* so that

- resulting real-arc set $A \cup A^{add}(\pi)$ does not contain any cycle;
- resulting constraint system CS(π) remains feasible;

- the cost of path π:

$$cost(\pi) := \alpha \cdot time(\pi) + \sum_{k=1}^{K} \beta_k \cdot time(\Lambda_k^\pi) + \sum_{x \in X^*} \gamma_x \cdot Min_x^\pi,$$

is minimal, where α, β_k, γ_x are nonnegative scaling coefficients for $k = 1, \ldots, K$, and $x \in X^*$.

An Algorithm for the Virtual Path Problem

Let us recall that the A* algorithm was introduced in [115] as an adaptation of Dijkstra's algorithm in order to deal with the path search in very large state networks, like those which may be involved in robotics. It fits our purpose here since at any time during the resolution problem, we shall deal not only with some current vertex x, but also with the full sequence of all virtual-arcs which have been previously inserted to reach x. We are going to describe an algorithm that we call Virtual A* and which will perform an enumeration of the sequences of virtual-arcs likely to be part of virtual path π, while relying on the preprocess that we describe next.

Preprocess for the Virtual A* algorithm:

1. Compute a table DLARGE and a table DSHORT which provide us, for any pair $x_1, x_2 \in X$, with the weight according to the function *time*; of respectively a maximum weight path and a minimum weight path from x_1 to x_2. If no path exists from x_1 to x_2 in the original digraph G then we set $\texttt{DLARGE}[x_1, x_2] = -\infty$ and $\texttt{DSHORT}[x_1, x_2] = +\infty$. Clearly (CS) constraints imply that:
   - for any x_1, $x_2 \in X^*$, we have that $t_{x_2} \geq t_{x_1} + \texttt{DLARGE}[x_1, x_2]$;
   - for any $x \in X$, $Min_x = \texttt{DLARGE}[x^{\min}, x]$, and $Max_x = \Omega - \texttt{DLARGE}[x, x^{\max}]$. The construction of DSHORT can be performed together with a complementary data structure allowing us to retrieve, for any x_1, x_2 in X, a minimum weight path π_{x_1,x_2} of H (with respect to the weight function *time*) from x_1 to x_2.

   Note this part of the preprocess helps us in performing both constraint propagation and no cycle checking during the resolution process.

2. For any pair (x, y) in $X^* \times X^*$ such that $(x, y) \notin A$, set an arc (x, y) if there exists some virtual-arc a^{virt} from x to y and provide it with a weight $D^1_{(x,y)}$

equals to $\min\{\Delta = time(a^{virt}) : a^{virt}$ has tail x and head $y\}$. Let A^1 be the set of arcs obtained that way. Then $(X^*, A \cup A^1)$ defines an oriented digraph structure $H1$. By setting $D^1_a = time(a)$, for all $a \in A$; we provide any arc a with a weight D^1_a. Next, for any $x \in X^*$, we compute (through Dijkstra algorithm) the weight $\texttt{W1}[x]$ of a minimum weight path (according to $\mathbf{D}^1 = (D^1_a, a \in A)$) from x to $\hat{d}$. Clearly $\texttt{W1}[x]$ provides us with a lower bound for the weight of a minimum cost virtual path from x to $\hat{d}$, since we do not care here with the feasibility of such a path. Then we set $\texttt{LB} = \alpha \cdot \texttt{W1}[\hat{o}] + \sum_{k=1}^{K} \beta_k \cdot L_k + \sum_{x \in X^*} \gamma_x \cdot Min_x$. By the same way, LB provides us with a lower bound for the cost $cost(\pi)$ of any virtual path π which connects $\hat{o}$ to $\hat{d}$ in the virtual digraph H^* and which does not consist of a single virtual-arc from $\hat{o}$ to $\hat{d}$.

3. Compute a value $val^*_{\hat{o},\hat{d}}$ equals to the minimum possible cost of a virtual path consisting of a single virtual-arc $a^{virt}_{\hat{o},\hat{d}}$ with tail $\hat{o}$ and head $\hat{d}$. Let us denote by $\pi^{\hat{o},\hat{d}}$ the resulting virtual path. It follows that $val^*_{\hat{o},\hat{d}} = cost(\pi^{\hat{o},\hat{d}})$. If $\pi^{\hat{o},\hat{d}}$ exists as a feasible solution of our problem, then set $\texttt{UB} = val^*_{\hat{o},\hat{d}}$, otherwise set $\texttt{UB} = +\infty$. The computation of $val^*_{\hat{o},\hat{d}}$ can be achieved according to the following lemma.

Lemma 4.1. *Given a path π' consisting of a single virtual-arc $a^{virt}_{\hat{o},\hat{d}} = \big(\hat{o}, \hat{d}, a_1 = (x_1, y_1), a_2 = (x_2, y_2)\big)$. We can compute, for any path Λ_k the weight $time(\Lambda^{\pi'}_k)$ by adding Δ_1 to $time(\Lambda_k)$ if $a_1 \in \Lambda_k$ and adding Δ_2 to $time(\Lambda_k)$ if $a_2 \in \Lambda_k$. Similarly, if we define $X' = \{\hat{o}, \hat{d}, y_1, x_2, y_2\}$ and $X^{user} = \{x \in X^* : \gamma_x \neq 0\}$, then for all $x \in X^{user}$ we can compute the values $Min^{\pi'}_x$ by propagating first the constraints* (V1) *and* (C1) *for the vertices in X' and then by taking $Min^{\pi'}_x$ as* $\sup(Min_x, \sup_{x' \in X'}(Min_{x'} + \texttt{DLARGE}[x', x]))$. *It follows that $cost(\pi')$ can be computed in $O(|X^{user}|)$.*

Proof. The first part of above statement follows from the updating process described in (V1). For the second part, we sort X' in a topological ordering with respect of $<<^{A \cup \{(x_1,\hat{o}),(\hat{o},\hat{d}),(\hat{d},y_2)\}}$, and we use such an ordering to apply the formulas described in (V1) to propagate the constraints (V1) and (C1) on vertices $\hat{o}$, $\hat{d}$, y_1, x_2, and y_2 in constant time. Then, for $x \in X^{user}$ we have that $Min^{\pi'}_x = \sup(Min_x, \sup_{x' \in X'}(Min_{x'} + \texttt{DLARGE}[x', x]))$. ∎

The virtual-arcs of H^* may be provided with an oriented digraph relation Σ as follows. Virtual-arcs $a^{virt}_1 = (x_1, y_2, a_1, a_2)$ and $a^{virt}_2 = (x'_1, y'_2, a'_1, a'_2)$ define an arc (a^{virt}_1, a^{virt}_2) in Σ if and only if $\texttt{DSHORT}[y_2, x'_1] \neq +\infty$. We provide any such an arc (a^{virt}_1, a^{virt}_2) in Σ with a weight $\Phi(a^{virt}_1, a^{virt}_2)$ equals to the number of x-y-paths $\pi_{x,y}$ of real-arcs, such that $\texttt{DSHORT}[y_2, x] + \sum_{a \in A(\pi_{x,y})} time(a) + \texttt{DSHORT}[y, x'_1] = \texttt{DSHORT}[y_2, x'_1]$, that means the number of paths of real-arcs likely to be involved in a virtual path π which would connect a^{virt}_1 to a^{virt}_2 according to Σ. Note that Σ may contain cycles.

Let us suppose that π is some virtual path which is a feasible solution of our Virtual Path problem and that $(a_1^{virt}, a_2^{virt}, \ldots, a_\nu^{virt})$ is the sequence of virtual-arcs involved in π. Then we have the following.

Lemma 4.2. *The sequence $(a_1^{virt}, a_2^{virt}, \ldots, a_\nu^{virt})$ defines a chain (i.e., an elementary path) according to Σ, which does not induce any cycle (that means, which is such that, for any x_1, $x_2 \in X$ with $x_1 <<^A x_2$, $\texttt{DSHORT}[x_2, x_1] = +\infty$). Besides, if π is optimal, then for any consecutive elements $a_i^{virt} = (x_1^i, y_2^i, a_1^i, a_2^i)$, $a_{i+1}^{virt} = (x_1^{i+1}, y_2^{i+1}, a_1^{i+1}, a_2^{i+1})$ in $(a_1^{virt}, a_2^{virt}, \ldots, a_\nu^{virt})$, the virtual path π follows a path of G with minimum time weight equal to $\texttt{DSHORT}[y_2^i, x_1^{i+1}]$.*

Proof. The first part from the proof derives from the fact we forbid arc set $A \cup A^{add}(\pi)$ from containing any cycle. The second part is due to the fact that, if the subpath of real-arcs followed by π from y_2^i to x_1^{i+1} is not a minimum weight path in the sense of weight function *time* (see Figure 4.8), then we may improve π with respect of the first term of $cost(\pi) = \alpha \cdot time(\pi) + \sum_{k=1}^{K} \beta_k \cdot time(\Lambda_k^\pi) + \sum_{x \in X^*} \gamma_x \cdot Min_x^\pi$, without modifying any of the two other terms. ■

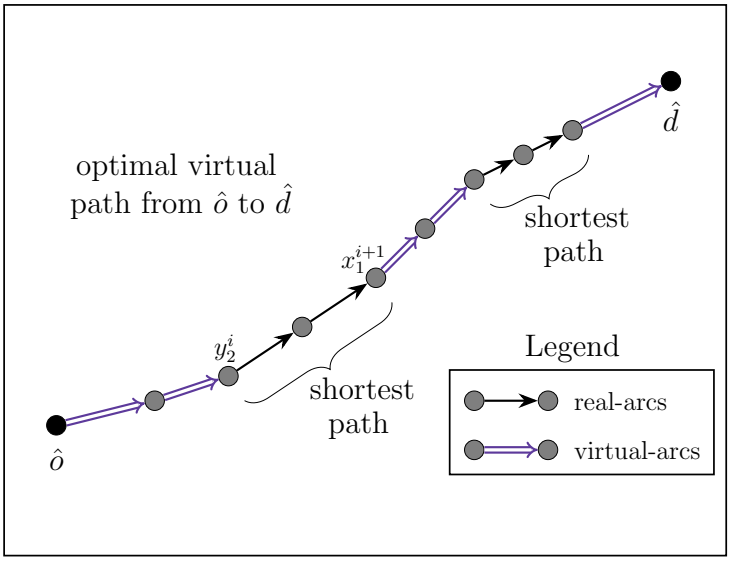


Figure 4.8: Representation of an optimal virtual path from $\hat{o}$ to $\hat{d}$.

Now, we describe the Virtual A* algorithm, which performs a breadth-first search (BFS) process inside a tree of states; where the BFS is filtered by a lower/upper bound device.

- A state s consists of a pivot vertex $x \in X^*$ and a sequence $virt(s) = (a_1^{virt}, \ldots, a_\nu^{virt})$ (possibly empty) of virtual-arcs. Any state $s = (x, virt(s))$ is provided with the following additional information.

- For any $a_i^{virt} = (x_1^i, y_2^i, a_1^i = (x_1^i, y_1^i), a_2^i = (x_2^i, y_2^i))$, $i = 1, \ldots, \nu$, its current values $Min_{x_1^i}$, $Min_{y_1^i}$, $Min_{x_2^i}$, and $Min_{y_2^i}$, related to the time windows $[Min_y, Max_y]$ for $y = x_1^i, y_1^i, x_2^i, y_2^i$.
- The value Min_x (note that here, x is the pivot vertex).
- The weight $\lambda(s)$ (with respect to the weight function *time*) of the path $\pi(s)$ in H^* from $\hat{o}$ to x and which is involved in state s.
- The subset X_{act}^{user} of user nodes y whose value Min_y has not been fully determined yet.
- A value $val(s)$ which reflects the expected quality of state s and decomposes itself into a current cost $val1(s)$ and an estimation value $val2(s)$ of the cost which remains to be run in order to arrive to $\hat{d}$.

- We use an expansion list `LS` of states s, sorted in a nondecreasing order of $val(s)$ values.

At a given time during the process, we are provided with a state $(x, virt(s))$ and we try to expand it along an arc a^* with tail x. Before doing it, we first check that:

- if a^* is a real-arc, then it keeps along with Lemma 4.2;
- if a^* is a virtual-arc, then it does not induce the creation of any cycle.

Lemma 4.3. *In case a^* is a real-arc (x, y), then a^* meets* Lemma 4.2 *if and only if $\lambda(s) + time(a^*) = \texttt{DSHORT}[\hat{o}, y]$. In case a^* is a virtual-arc (x, y, a_1, a_2), then a^* does not create any cycle if and only if $\texttt{DSHORT}[y, x_1^i] = +\infty$ for any $i = 1, \ldots, \nu$.*

Proof. The first part expresses the fact that the path from $\hat{o}$ to $\hat{d}$ which results from the expansion of x through a real-arc (x, y) must be a minimum weight path.

As for the second statement, let us first recall that we only consider virtual-arcs which meet (J1). Then we see through induction that, if the insertion of a virtual-arc a^* creates a cycle, then such a cycle cannot involve only the virtual-arc $a^* = (x, y, a_1, a_2)$ (because $\texttt{DSHORT}[y, x] = +\infty$) and so the cycle must involve some previously created real arc (x_1^i, y_2^i), and so that we must have $\texttt{DSHORT}[y, x_1^i] \neq +\infty$, for some i (see Figure 4.9). In case $x = \hat{o}$ (respectively $y = \hat{d}$), then we adapt our notations, since a^* has tail $\hat{o}$ (respectively a^* has head $\hat{d}$). ■

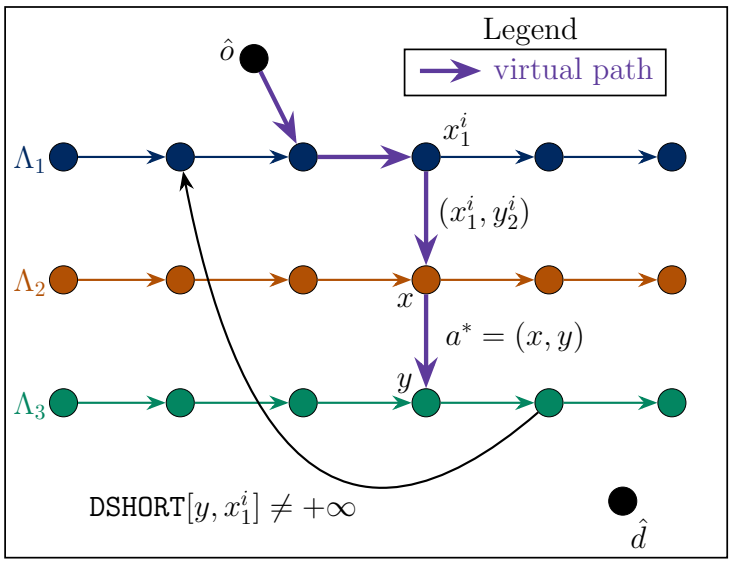


Figure 4.9: The figure used in the proof of Lemma 4.3. Given a state $s = (y, virt(s))$ with $virt(s) = (a_1^{virt} = (x_1^1, y_1^1, a_1^1, a_2^2), \ldots, a_\nu^{virt} = (x_1^\nu, y_1^\nu, a_1^\nu, a_2^\nu))$, we can verify that the virtual path from $\hat{o}$ to y codified by s induces a cycle in $A \cup A^{add}(\pi)$ by checking if $\texttt{DSHORT}[y, x_1^i] \neq \infty$ for some $i = 1, \ldots, \nu$.

Let us consider a state $s = (x, virt(s))$ with $virt(s) = (a_1^{virt}, a_2^{virt}, \ldots, a_\nu^{virt})$ and involving a path π. Let us suppose that $a_i^{virt} = (x_1^i, y_2^i, a_1 = (x_1^i, y_1^i), a_2 = (x_2^i, y_2^i))$ for $i = 1, \ldots, \nu$. Next, we must compute the state $s' = (y, virt(s'))$ which results from s when we extend π by an arc a^* and its impact on the constraints system (CS).

The case when a^* is a real-arc (x, y) is easy since (CS) remains unchanged and we have the following updates: (V2)

- $Min_y^{\pi \cup \{a^*\}} = \sup(\, Min_y^\pi, \sup(\, Min_{x_1^\nu}^\pi + \texttt{DLARGE}[x_1^\nu, y], Min_{x_2^\nu}^\pi + \texttt{DLARGE}[x_2^\nu, y], Min_{y_1^\nu}^\pi + \texttt{DLARGE}[y_1^\nu, y], Min_{y_2^\nu}^\pi + \texttt{DLARGE}[y_2^\nu, y]))$;
- $\lambda(s') = \lambda(s) + time(a^*)$;
- $val1(s') = val1(s) + \alpha \cdot time(a^*)$;
- $val2(s') = \texttt{W1}[y]$.

Things are more complicated when a^* is a virtual-arc $a^* = (x, y, a_1 = (x_1, y_1), a_2 = (x_2, y_2))$ with coefficients $(\Delta, \Delta_1, \Delta_2)$. We have that: (V3)

- $\lambda(s') = \lambda(s) + \Delta$;
- $val2(s') = \alpha \cdot \texttt{W1}[y]$;
- values $Min_{x_2}^{\pi \cup \{a^*\}}$, $Min_{y_1}^{\pi \cup \{a^*\}}$, $Min_{y_2}^{\pi \cup \{a^*\}}$ are computed according to Lemma 4.4, while $Min_{x_1}^{\pi \cup \{a^*\}} = Min_{x_1}^\pi$;

- we have $virt(s') = (virt(s), a^*)$;
- $val1$ is increased by $\alpha \cdot \Delta + \Delta_1 \cdot (\sum_{k \in K_1} \beta_k) + \Delta_2 \cdot (\sum_{k \in K_2} \beta_k) + \Theta$, where:
  - $K_i = \{k \in \{1, \ldots, K\} : a_i \in \Lambda_k\}$ for $i = 1, 2$;
  - Θ comes according to Lemma 4.5.
- X^{user}_{act} is updated by withdrawing $X^o(virt(s))$ computed as in Lemma 4.5;
- $s' = (y, virt(s'))$ with $virt(s')$ above defined.

Lemma 4.4. *The values $Min_{x_2}^{\pi \cup \{a^*\}}$, $Min_{y_1}^{\pi \cup \{a^*\}}$, $Min_{y_2}^{\pi \cup \{a^*\}}$ can be obtained in $O(\nu)$ by propagating constraints* (C1) *and* (V1) *from current value Min_x^π.*

Proof. We proceed as in Lemma 4.1. We first modify every value Min_{x^o}, with $x^o = x_2, y_1, y_2$ according to the formula $Min_{x^o}^{\pi \cup \{a^*\}} = \sup\big(Min_{x^o}^\pi, \sup_{i=1,\ldots,\nu}(Min_{x_1^i}^\pi + \texttt{DLARGE}[x_1^i, x^o], Min_{x_2^i}^\pi + \texttt{DLARGE}[x_2^i, x^o], Min_{y_1^i}^\pi + \texttt{DLARGE}[y_1^i, x^o], Min_{y_2^i}^\pi + \texttt{DLARGE}[y_2^i, x^o])\big)$. Then we propagate constraints (V1) on the set $\{x, x_2, y_1, y_2\}$ (completed by $\hat{d}$ in case the head of a^* is $\hat{d}$) and we check that resulting values $Min_y^{\pi \cup \{a^*\}}$ for $y \in X^*$ are stable under (V1) and (C1). ■

Lemma 4.5. *Value Θ can be computed in $O(\nu \cdot |X^{user}|)$ as follows.*

- *If a^* has tail $\hat{o}$, then $\Theta = \gamma_{\hat{o}} Min_{\hat{o}}^{\{a^*\}} + \gamma_{y_2} Min_{y_2}^{\{a^*\}} + \gamma_{x_2} Min_{x_2}^{\{a^*\}}$ such that it derives from* Lemma 4.4.
- *If a^* does not have tail $\hat{o}$ and does not have head $\hat{d}$, then set:*
  - *$X^o(virt(s)) = \{x^o \in X^{user} - \{\hat{o}, \hat{d}\} :$ for all $k \in K$, and for all $i \in \{1, \ldots, \nu\}$, x^o is not a predecessor of x_1^i or y_2^i in $\Lambda_k\}$;*
  - *for any $x^o \in X^o(virt(s))$, we define $B_{x^o} = \sup\big(Min_{x^o}^\pi, \sup_{i=1,\ldots,\nu}(Min_{x_1^i}^\pi + \texttt{DLARGE}[x_1^i, x^o], Min_{x_2^i}^\pi + \texttt{DLARGE}[x_2^i, x^o], Min_{y_1^i}^\pi + \texttt{DLARGE}[y_1^i, x^o], Min_{y_2^i}^\pi + \texttt{DLARGE}[y_2^i, x^o])\big)$.*
  - *$\Theta = \sum_{x^o \in X^o(virt(s))} \gamma_{x^o}(B_{x^o} - Min_{x^o}^\pi)$.*
- *If a^* has head $\hat{d}$, then set:*
  - *$X^o(virt(s)) = \{x^o \in X^{user} - \{\hat{o}, \hat{d}\} :$ for all $k \in K$, and for all $i \in \{1, \ldots, \nu\}$, x^o is not a predecessor of x_1^i or y_2^i in $\Lambda_k\}$;*
  - *for any $x^o \in X^o(virt(s))$, we define $B_{x^o} = \sup\big(Min_{x^o}^\pi, \sup_{i=1,\ldots,\nu}(Min_{x_1^i}^\pi + \texttt{DLARGE}[x_1^i, x^o]), Min_{x_2^i}^\pi + \texttt{DLARGE}[x_2^i, x^o], Min_{y_1^i}^\pi + \texttt{DLARGE}[y_1^i, x^o], Min_{y_2^i}^\pi + \texttt{DLARGE}[y_2^i, x^o])\big)$.*
  - *$\Theta = \sum_{x^o \in X^o(virt(s))} \gamma_{x^o}(B_{x^o} - Min_{x^o}^\pi) + \gamma_{\hat{d}} Min_{\hat{d}}^{\pi \cup \{a^*\}}$.*

Proof. The first statement of above list expresses the fact that only values Min_x, which may be affected by the insertion of a virtual-arc with origin $\hat{o}$, and that we are sure that they are not going to be modified anymore (Lemma 4.4), correspond to $x = \hat{o}$, $x = x_2$, and $x = y_2$. As for the second statement, we deal as in Lemmas 4.1 and 4.4 in order to update target values $Min^{\pi}_{x^o}$, with the difference that we focus on $x^o \in X^o(virt(s))$. This subset involves the vertices x^o whose values Min_{x^o} have not been updated yet and the vertices that we are sure (because of Lemmas 4.1 and 4.4) that they are not going to be impacted by the insertion of additional virtual-arcs.

The third statement can be handled as the second one, with the difference that we know that we become sure that no value Min_x which has not been modified yet, is going to be modified later. ■

Algorithm 8 shows a pseudocode for the Virtual A* algorithm. Note that, $\partial^+_{H^*}(x)$ denotes the set of arcs (real-arcs or virtual-arcs) in H^* with tail x. Also we denote by Head(LS) the first element of the list LS.

Algorithm 8: Virtual A* algorithm

Input : Digraph G, and virtual digraph H^*. Tables DLARGE, and DSHORT; value $val^*_{\hat{o},\hat{d}}$.
Output: A feasible virtual path (if there exists one), or a failure message.

```
1  LS ← {(ô, Nil)}, val ← LB, val1 ← 0, val2 ← LB, iterate ← true
2  if π^op = ((ô, d̂, a1, a2)) is feasible then
3  |  path ← ((ô, d̂, a1, a2)), UB ← val*_{ô,d̂}
4  else
5  |  path ← Nil, UB ← +∞
6  while (LS ≠ ∅ ) and (iterate = true) do                    [Main loop]
7  |  s = (x, virt(s)) ← Head(LS)
8  |  Remove Head(LS) from LS
9  |  if (val(s) ≥ UB then
10 |  |  iterate ← false
11 |  else if (n = d̂) then
12 |  |  path ← virt(s), iterate ← false
13 |  for a* ∈ ∂+_{H*}(x) satisfying Lemma 4.3 do
14 |  |  s' ← Expand(s, a*)
15 |  |  if (sup_{y∈X*}(Min_y^{π(s')}) ≤ Ω) and s' is not in LS then
16 |  |  |  Push s' into LS according to val(s')
17 if path ≠ Nil then                                         [If any solution]
18 |  return virtual path retrieved from variable path
19 else                                                      [If no solutions]
20 |  Print "No path found"
```

To conclude this section, we specify the retrieve instruction for retrieving a virtual path from a sequence of virtual-arcs $virt(s)$ and the Expand function.

Instruction retrieve. Let $virt(s)$ be the sequence of virtual-arcs which we have got at the end of the Virtual A* algorithm. If $virt(s) = ((\hat{o}, \hat{d}, a_1, a_2))$ then we take $\pi = ((\hat{o}, \hat{d}, a_1, a_2))$. In other case, we should have $virt(s) = (a_1^{virt}, \ldots, a_\nu^{virt})$ with $a_i^{virt} = (x_1^i, y_2^i, (x_1^i, y_1^i), (x_2^i, y_2^i))$ for $i = 1, \ldots, \nu$; $x_1^1 = \hat{o}$, $y_2^\nu = \hat{d}$, and $\nu \geq 2$. We get path π by setting, for any $i = 1, \ldots, \nu - 1$:

- π_i = shortest path in G from y_2^i to x_1^{i+1};
- $\pi = (a_1^{virt}, \pi_1, a_2^{virt}, \ldots, a_1^{virt}, \pi_{\nu-1}, a_\nu^{virt})$.

Expand **function.** We must consider the following two cases.

1. If a^* is a real-arc (x, y), then the Expand function works according to (V2).
2. If a^* is a virtual-arc $(x, y, a_1 = (x_1, y_1), a_2 = (x_2, y_2))$ then the Expand function works according to (V3), Lemmas 4.4-4.5.

We can adapt the Virtual A* algorithm, in order to impose an upper bound `VMAX` on the number of virtual-arcs involved into virtual path π, by using a counter variable `vcount` inside the state variables of Virtual A* to reject any virtual-arc a^* as soon as `vcount` is equal to `VMAX`.

We see that Virtual A* works by visiting the *Virtual Tree* Υ whose vertex set $V(\Upsilon)$ consists of all finite chains $virt(s) = (a_1^{virt}, a_2^{virt}, \ldots)$ and whose arcs set $A(\Upsilon)$ comes in a natural way: $virt(s) = (a_1^{virt}, \ldots, a_\nu^{virt})$ is the parent of any $virt(s') = (a_1^{virt}, \ldots, a_\nu^{virt}, a_{\nu+1}^{virt})$. Related arc is provided with a weight $\Phi^*(virt(s), virt(s')) = \Phi(a_\nu^{virt}, a_{\nu+1}^{virt})$. Also, for any vertex $virt(s) = (a_1^{virt}, \ldots, a_\nu^{virt}) \in V(\Upsilon)$, we can associate a weight $\Xi(virt(s)) = \nu \cdot |X^o(virt(s))|$, where $X^o(virt(s))$ is the set defined in Lemma 4.5. We have the following statement.

Theorem 4.1 - Complexity of the Virtual A* Algorithm
Above Virtual A algorithm solves the Virtual Path Problem in an exact way, with a complexity of* $O\big(\sum_{virt(s)\in V(\Upsilon)} \Xi(virt(s)) + \sum_{(virt(s),virt(s'))\in A(\Upsilon)} \Phi^*(virt(s), virt(s'))\big)$. *It becomes time-polynomial if we impose an upper bound* `VMAX` *on the number of virtual-arcs involved in the searched path* π.

Proof. For the complexity part, we see that the number of possible states (i.e., $|V(\Upsilon)|$) is bounded by the number of chains $virt(s)$ in the digraph involved in Lemma 4.2. Lemmas 4.3-4.5 tell us that for $virt(s) = (a_1^{virt}, \ldots, a_\nu^{virt})$ the Expand procedure works in $O(\nu \cdot |X^o(virt(s))|)$.

As for the exactness of the algorithm, it derives from the fact that the quality of a virtual path π entirely depends on the chain induced by its virtual-arcs. Since the Virtual A* algorithm enumerates these chains while filtering the search process through branch-and-bound and constraint propagation, we get our result. ■

4.3 Application to the PDPT

In this section we introduce the 1-Request Insertion PDPT problem and we show that can be seen as a particular case of the Virtual Path Problem. Then, we propose the virtual A* algorithm for solving the problem in an exact way and a heuristic Dijkstra-like algorithm. We also present some numerical results.

4.3.1 The PDPT Problem

We first suppose that we are provided with a finite set X of "points" representing physical locations, given together with the following two functions.

- A function $time : X \times X \to \mathbb{R}_+$, such that for any $x, y \in X$, the value $time(x, y)$ means the time required for a vehicle to move from x to y according to a minimum weight path strategy (with respect to the weight *time*).
- A function $relay : X \times X \to X$ is going to be involved in order to determine "relay vertices", that means the transfer points where two vehicles v_1, v_2 will meet in order to perform some transfer. Intuitively, for any $(x_1, x_2) \in X \times X$, the element $u = relay(x_1, x_2)$ corresponds to a place that is more or less at the same distance from x to y and minimizes the sum $time(x_1, u) + time(u, x_2)$. For example, if X contains points from a Euclidean plane, we can define the function *time* to be the Euclidean distance, and then $u = relay(x_1, x_2)$ would be an element in X with minimal Euclidean distance to the midpoint of the segment from x_1 to x_2.

A PDPT instance consists of a finite set of points X together with the two functions $time : X \times X \to \mathbb{R}_+$ and $relay : X \times X \to X$ that we have just defined, a finite set of vehicles V with a common capacity $\kappa \in \mathbb{R}_+$, a *time horizon* $[0, \Omega]$, and a finite set $R \subset X \times X \times \mathbb{R}_+$ of *requests* $r = (o_r, d_r, \ell_r)$. The point o_r is the *origin* of r, the point d_r is the *destination* of r, and ℓ_r is the *load* of r. Also, we have two functions $start : V \to X$ and $end : V \to X$. For $v \in V$, $start(v)$ is called the *starting depot* vertex of v and $end(v)$ is called the *ending depot* vertex of v.

Figure 4.10 shows an example of PDPT instance involving five points, two vehicles, and two requests.

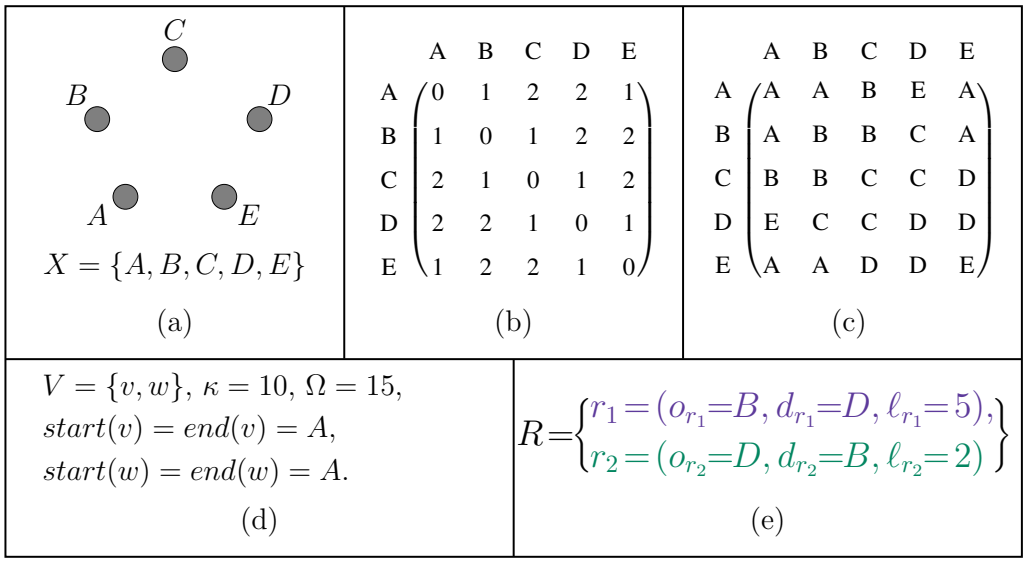


Figure 4.10: An example of PDPT instance. (a) The set $X = \{A, B, C, D, E\}$ of points. (b) A symmetric matrix whose entries define the distance function *time*. (c) A matrix defining the function *relay*. (d) The set of vehicles $V = \{v, w\}$, the capacity κ, the upper limit of the time horizon $[0, \Omega]$, and the starting and ending depot vertices of vehicles. (e) The set R of requests consisting of $r_1 = (B,\ D,\ 5)$ and $r_2 = (D,\ B,\ 2)$.

Now we will describe briefly the PDPT. Although we are not going to deal with the whole problem until Section 4.4, we give here, for completeness reasons, some intuition about this problem and clarify some of the involved details. This will allow us to set the stage for the insertion subproblem model which will be detailed later in this section.

The PDPT consists in finding a schedule for the vehicles fleet, allowing to transport the requests from their origins to their destinations. Vehicles capacity must be respected at any moment, but vehicles are allowed to transfer requests to other vehicles through a weak synchronization mechanism. This is, if a vehicle v is transferring a load ℓ_r to another vehicle w at a relay vertex z, and the receiving vehicle w arrives first to z, then w has to wait for the arriving of v with the load ℓ_r. The time duration for traversing every path in the schedule (including waiting times), must be within the given time horizon.

Depending on the context, there can be several ways to define the cost of such schedules. Here, we consider that the cost of a schedule is a nonnegative weighted linear combination of the sum of the distances traversed by the requests, the sum of the distances traversed by the vehicles, and the sum of the arrival times of the requests; and we aim to find a minimal cost schedule.

Now we proceed to describe our formal model for the insertion problem involved in the PDPT.

4.3.2 Formal Description of a PDPT Feasible Solution

Let G_X be the digraph with vertex X and arc set $\{(x, y) \in X \times X,\ x \neq y\}$. According to the above definition of a PDPT instance, we define a *solution* (Γ, Π) for the PDPT instance, which we also call a *PDPT schedule* as:

- A collection $\Gamma = \left\{\Gamma(v),\ v \in V\right\}$ of arc progressions on G_X, where the vertex sequence $\Gamma(v) = (x_0^v = start(v),\ x_1^v\ ,\ \ldots\ ,\ x_{n_v}^v = end(v))$ is the route followed by vehicle v in G_X;
- A collection $\Pi = \left\{\Pi(r),\ r \in R\right\}$ of arc progressions on G_X, where the vertex sequence $\Pi(r) = (y_0^r = o_r, y_1^r, \ldots, y_{n_r}^r = d_r)$ is the route followed by request r when moving from its origin o_r to its destination d_r. Points y_1^r, ..., $y_{n_r}^r$ belong to the set $\left\{x_i^v : v \in V, 0 \leq i \leq |\Gamma(v)| - 1\right\}$ and we may distinguish the following two types of moves for a request r.
  - If y_q^r and y_{q+1}^r are related to two consecutive vertices of path $\Gamma(v)$, then they have to be consecutive in $\Pi(r)$ and so we talk about a *vehicle move* inside vehicle v;
  - If $y_q^r = x_i^v$ and $y_{q+1}^r = x_j^w$ are related to two distinct paths $\Gamma(v)$ and $\Gamma(w)$, then they refer to the same vertex in X and so we talk about a *transfer move* from vehicle v to vehicle w.

Denoting by $X(\Gamma)$ the set $\left\{(v, i) : v \in V, 0 \leq i \leq |\Gamma(v)| - 1\right\}$, we see that such a solution induces a digraph structure $G(\Gamma, \Pi)$ on the vertex set $X(\Gamma)$, with arc set $A(\Gamma, \Pi)$ defined as follows.

- With any $v \in V$ and any $0 \leq i \leq |\Gamma(v)| - 1$, we associate a vehicle arc $a = ((v, i), (v, i + 1))$, with weight $time(a) := time(x_i^v, x_{i+1}^v)$. Following the paths $\Pi(r)$ allows the computation of the load ℓ_i^v of arc a, which is the sum of loads ℓ_r for requests r which move through this arc. Of course ℓ_i^v does not have to exceed the capacity κ. (C1: *Load Constraints*)
- With any $r \in R$ and $1 \leq q \leq |\Pi(r)| - 1$, such that moving from $y_q^r = x_i^v$ to $y_{q+1}^r = x_j^w$ corresponds to a transfer move from vehicle v to vehicle w, we associate a synchronization arc $a = ((v, i), (w, j))$, with weight $time(a) := time(y_q^r, y_{q+1}^r) = 0$ (because in this case y_q^r and y_{q+1}^r correspond to the same place). We denote by A' the set of those synchronization arcs.

Observe that proceeding in this manner, we have extended the weight function $time : A(G) \to \mathbb{R}_+$ of digraph G to a weight function $time : A(\Gamma, \Pi) \to \mathbb{R}_+$ of digraph $G(\Gamma, \Pi)$.

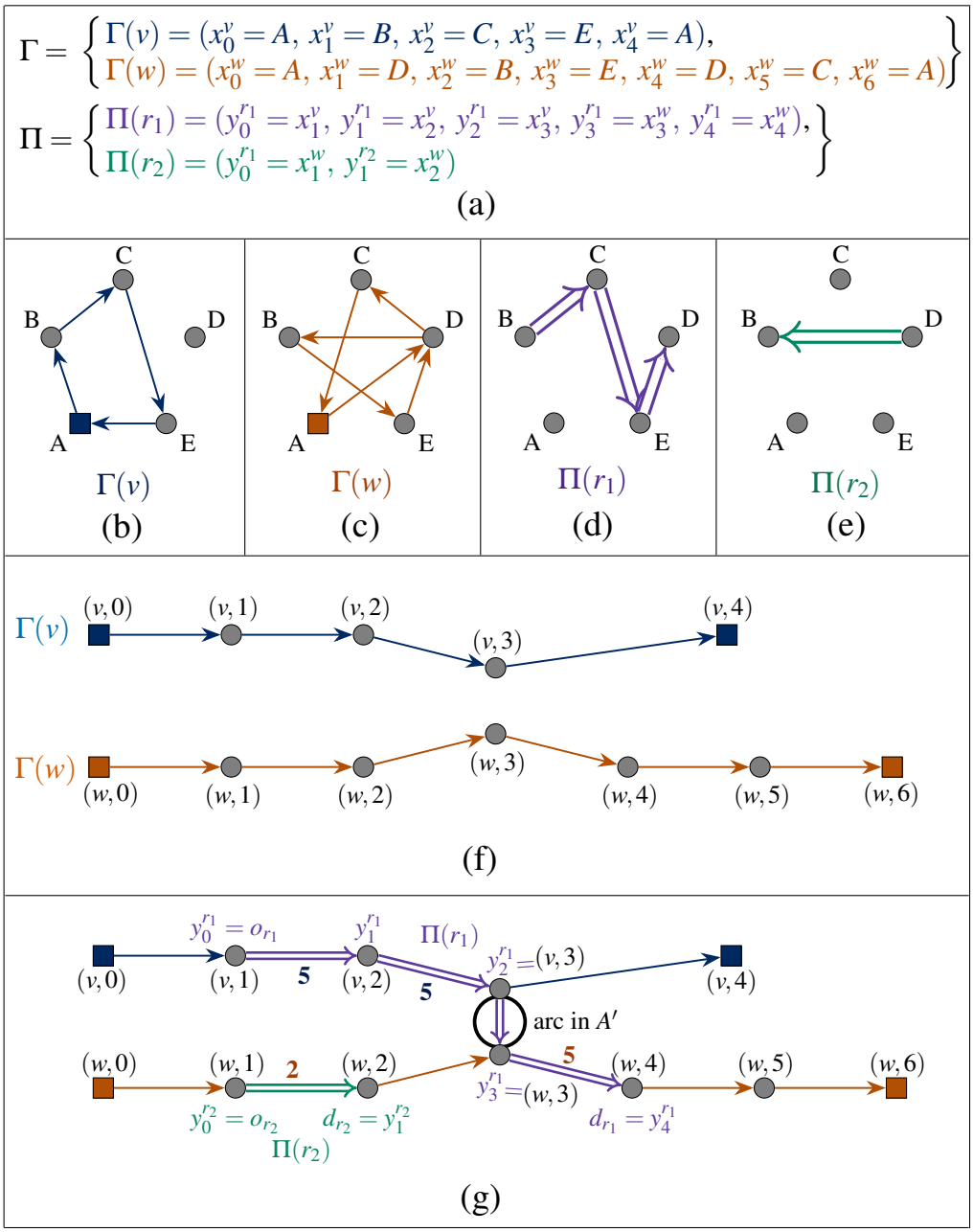


Figure 4.11: (a) An example of solution for the instance defined in Figure 4.10. (b)-(e) Subgraphs of G_X arc-induced by the paths $\Gamma(v)$, $\Gamma(w)$, $\Pi(r_1)$, and $\Pi(r_2)$, respectively. (f) The paths on the digraph $G(\Gamma, \Pi)$ corresponding to the directed paths $\Gamma(v)$ and $\Gamma(w)$ of G_X. (g) The digraph $G(\Gamma, \Pi)$. We have also depicted with double arrows the directed paths corresponding to $\Pi(r_1)$ and $\Pi(r_2)$. Numeric arc labels indicate positive loads traversing through arcs in $A(\Gamma, \Pi) \setminus A'$.

Next, we define a function $time_{G(\Gamma,\Pi)} : X(\Gamma) \times X(\Gamma) \to \mathbb{R}_+$, by taking

$$time_{G(\Gamma,\Pi)}\big((v,i),(w,j)\big) := dist_{(G(\Gamma,\Pi),time)}\big((v,i),(w,j)\big)$$

for all (v,i), $(w,j) \in X(\Gamma)$. This value $time_{G(\Gamma,\Pi)}((v,\, i),\, (w,\, j))$ can be understood as the minimal amount of time that is necessary to travel in $G(\Gamma,\, \Pi,\, r)$ from $(v,\, i)$ to $(w,\, j)$.

Figure 4.11 shows an example of solution (Γ, Π) for the instance defined in Figure 4.10, and the associated digraph $G(\Gamma, \Pi)$.

Feasibility of (Γ, Π). As we just mentioned, (Γ, Π) must meet the above load constraints (C1). Also it must satisfy time consistency constraints defined as follows.

- With any vertex (v,i) in the digraph $G(\Gamma, \Pi)$ we associate a time value τ_i^v which represents the earliest time when vehicle v may leave vertex (v,i). Then we see that:
  - for any vehicle arc $a = ((v,i),(v,i+1))$, we have that $\tau_{i+1}^v \geq \tau_i^v + time(a)$;
  - weak synchronization implies that $\tau_j^w \geq \tau_i^v$ for any transfer arc $((v,i),(w,j))$.

They can be summarized as: (C2: *Time Consistency Constraints*)

- for any arc $a = ((v,i),(w,j))$ in the digraph $G(\Gamma, \Pi)$, we have a constraint $\tau_j^w \geq \tau_i^v + time(a)$;
- the time limit given by the time horizon $[0, \Omega]$ implies that, for any $(v,i) \in X(\Gamma)$ we have that $0 \leq \tau_i^v \leq \Omega$.

They impose that: (C3: *No Cycle Constraints*)

- arc set $A(\Gamma, \Pi)$ does not contain any cycle.

If (Γ, Π) satisfies those constraints, we can associate, with every (v,i) in $G(\Gamma, \Pi)$, a time window $[Min_{(v,i)},\, Max_{(v,i)}]$ which contains the feasible values of variable τ_i^v and which is constrained by (C2).

Cost of a solution (Γ, Π). As we said in Section 4.3.1, we are going to define the *cost* of a solution as a nonnegative weighted linear combination of the sum of the distances traversed by the requests, the sum of the distances traversed by the vehicles, and the sum of the arrival times of the vehicles.

4.3.3 The 1-Request Insertion PDPT Model

Those preliminaries allow us to define in a formal way the 1-Request Insertion PDPT, about the insertion of a new request into a current feasible PDPT schedule (Γ, Π). So we start from such a feasible schedule (Γ, Π) and from an additional request $r = (o_r, d_r, \ell_r)$. Intuitively, inserting request r means building a suitable sequence of the following five types of moves.

(1) Start from o_r and enter into some path $\Gamma(v)$ at the level of some vertex $(v, i{+}1)$, and so imposing vehicle v to make a deviation between (v, i) and $(v, i+1)$. Such a move will be performed once as the initial move (see Figure 4.12 (a)).

(2) Leave some path $\Gamma(w)$ at the level of some vertex (v, i) in order to reach d_r and so impose a vehicle a deviation between (v, i) and $(v, i+1)$. Such a move will be performed once as the final move (see Figure 4.12 (b)).

(3) Keep on inside vehicle v, while moving from some (v, i) to its successor $(v, i{+}1)$. Such type of move will be possibly performed several times (see Figure 4.12 (c)).

(4) Move from vehicle v to vehicle w while using some arc $((v, i), (w, j))$ of A' and some shared point $x_i^v = x_j^w$. Such type of move will be possibly performed several times (see Figure 4.12 (d)).

(5) Move from vehicle v to vehicle w at a relay vertex z while v is running between some vertex (v, i) and its successor $(v, i{+}1)$, and while w performs a deviation from $(w, j-1)$ to the relay vertex z and then returns to (w, j). Such a move will be possibly performed several times (see Figure 4.12 (e)).

Remark 2. For simplicity, here we are going to restrict our study to these five types of moves. However, we note that it is also possible to combine:

(6) moves of types (1) and (2), in such a way that r is inserted through a simple deviation of a vehicle v between two successive vertices of $\Gamma(v)$. This is, while traversing $\Gamma(v)$, vehicle v performs a deviation from (v, i) to transport the load ℓ_r from o_r directly to d_r, and then v returns to its original path at the level of vertex $(v, i+1)$ (see Figure 4.12 (f)). We call *direct-arcs* this type of arcs.

(7) movements of types (1) and (5), in such a way that, request r starts from o_r in a vehicle v and then is transferred directly to another vehicle w at some relay vertex z, while v is moving from (v, i) to $(v, i{+}1)$, and while w is moving from $(w, j-1)$ to (w, j) (see Figure 4.12 (g)).

(8) movements of types (2) and (5), in such a way that r leaves some path $\Gamma(v)$ (at the level of a vertex (v,i)), to be transferred from v to w at some relay vertex z. Then r reaches d_r directly from z in vehicle w. The first part of this process happens while v is moving from (v,i) to $(v,i+1)$, and the second one while w is moving from $(w,j-1)$ to (w,j) (see Figure 4.12 (h)).

(9) moves of types (1), (2) and (5), in such a way that r starts from o_r in a vehicle v, then r is transferred directly to another vehicle w at some relay vertex z, and finally r reaches d_r directly from z inside vehicle w, while v is moving from (v,i) to $(v,i+1)$, and while w is moving from $(w,j-1)$ to (w,j) (see Figure 4.12 (i)).

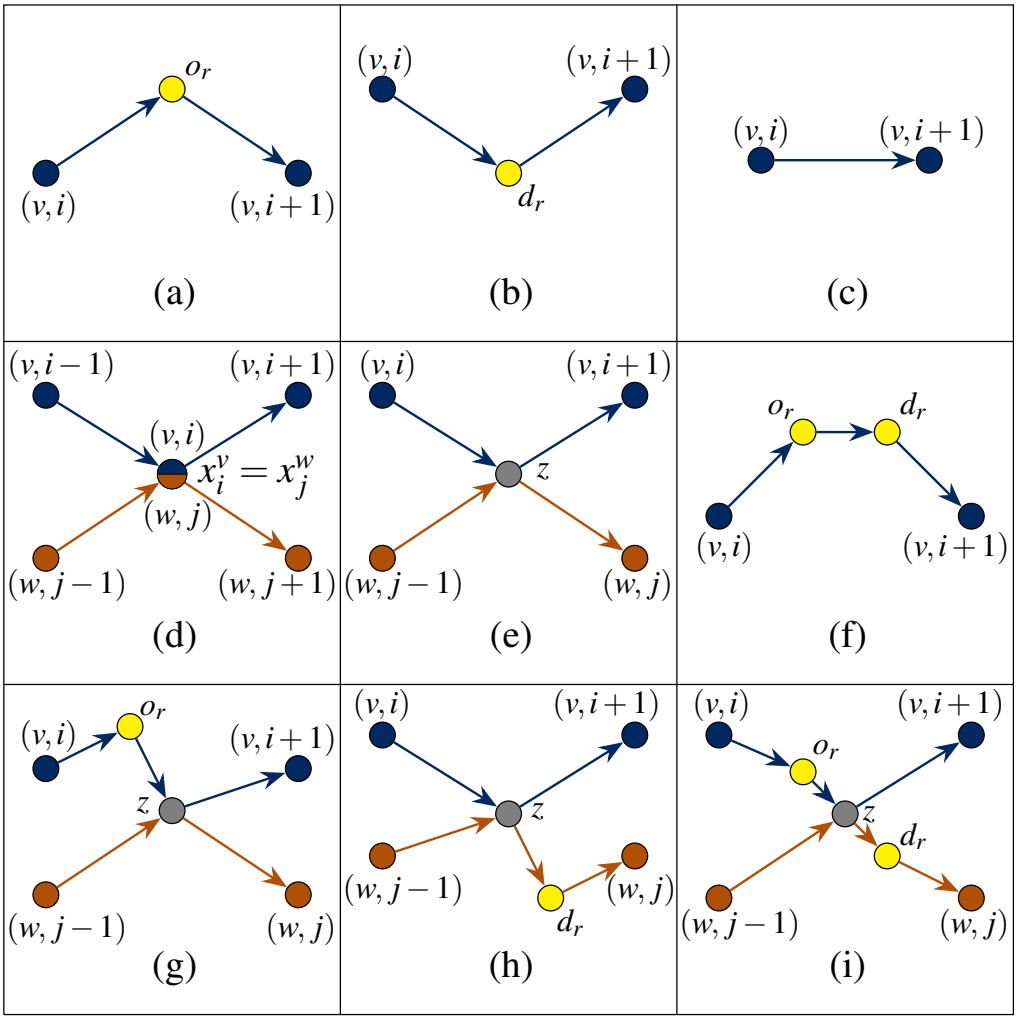


Figure 4.12: Types of moves for transporting a request $r = (o_r, d_r, \ell_r)$.

While moves of types (1), (2), (3), and (4) are easy to understand, we need to better explain the moves of type (5). Clearly, a move of type (5) involves some relay vertex z: both vehicle v and w are going to perform a deviation through z, vehicle

v will drop load ℓ_r in z, and vehicle w will take it as soon as possible. In order to identify z, we use the function *relay* and we take:

$$z = relay\big(relay(x_i^v, x_j^w),\ relay(x_{i+1}^v, x_{j-1}^w)\big). \tag{C4}$$

Please notice that (C4) restricts our freedom to choose the relay vertex, and so it has to be considered as an hypothesis of the model.

Now, given a PDPT schedule (Γ, Π) and one additional request r, we define the following digraph $H(\Gamma, \Pi, r)$.

- The vertices of $H(\Gamma, \Pi, r)$ are the vertices of $G(\Gamma, \Pi)$ plus the two vertices o_r and d_r;
- The arcs of $H(\Gamma, \Pi, r)$ are:
  - *In-arcs*: they have the form $a = (o_r, (v, i))$, with $0 \le i < |\Gamma(v)| - 1$, weight $time(a) := time(o_r, x_{i+1}^v)$ and such that $\ell_i^v + \ell_r \le \kappa$. Every in-arc $a = (o_r, (v, i))$ is associated with a value $\Delta_1^a := time\big(x_{i-1}^v, o_r\big) + time\big(o_r, x_i^v\big)$;
  - *Out-arcs*: with the form $a = ((w, j), d_r)$, with $1 \le j \le |\Gamma(w)| - 1$, weight $time(a) := time(x_j^w, d_r)$ and such that $\ell_j^w + \ell_r \le \kappa$. Every out-arc $a = ((w, j), d_r)$ is associated with a value $\Delta_1^a := time\big(x_j^w, d_r\big) + time\big(d_r, x_{j+1}^w\big)$;
  - *Vehicle-arcs*: arcs $a = ((v, i), (v, i+1))$ in $G(\Gamma, \Pi)$, with weight $time(a) := time(x_i^v, x_{i+1}^v)$ and such that $\ell_i^v + \ell_r \le \kappa$. Every vehicle-arc $a = ((v, i), (v, i+1))$ is associated with a value $\Delta_1^a := time\big(x_i^v, x_{i+1}^v\big)$;
  - *A′-arcs*: the arcs $a = ((v, i), (w, j)) \in A'$, with weight $time(a) := time(x_i^v, x_j^w) = 0$. Every A'-arc $a = ((v, i), (w, j))$ is associated with a value $\Delta_1^a := 0$;
  - *Transfer-arcs*: with the form $a = ((v, i), (w, j))$, weight $time(a) := time(x_i^v, z) + time(z, x_j^w)$ where $z = relay\big(relay(x_i^v, x_j^w), relay(x_{i+1}^v, x_{j-1}^w)\big)$. These arcs are such that $\ell_i^v + \ell_r \le \kappa$ and $\ell_{j-1}^w + \ell_r \le \kappa$. Every transfer-arc $a = ((v, i), (w, j))$ is associated with a value $\Delta_1^a := time\big(x_i^v, z\big) + time\big(z, x_{i+1}^v\big)$, a value $\Delta_2^a := time\big(x_{j-1}^w, z\big) + time\big(z, x_j^w\big)$, and a value $\Delta_3^a := time\big(x_i^v, z\big) + time\big(z, x_j^w\big)$.

We define a function $time_{H(\Gamma,\Pi)} : (X(\Gamma) \cup \{o_r, d_r\}) \times (X(\Gamma) \cup \{o_r, d_r\}) \to \mathbb{R}_+$, by taking

$$time_{H(\Gamma,\Pi,r)}\big((v, i),\ (w, j)\big) := dist_{(H(\Gamma,\Pi), time)}\big((v, i),\ (w, j)\big)$$

for all (v, i), $(w, j) \in X(\Gamma)$. This value $time_{H(\Gamma,\Pi)}((v,\ i),\ (w,\ j))$ can be understood as the minimal amount of time that is necessary to travel in $H(\Gamma,\ \Pi,\ r)$ from $(v,\ i)$ to $(w,\ j)$.

Figure 4.13 shows an example of construction of the digraph $H(\Gamma, \Pi, r)$ from a PDPT schedule (Γ, Π) and one additional request $r = (o_r, d_r, \ell_r)$.

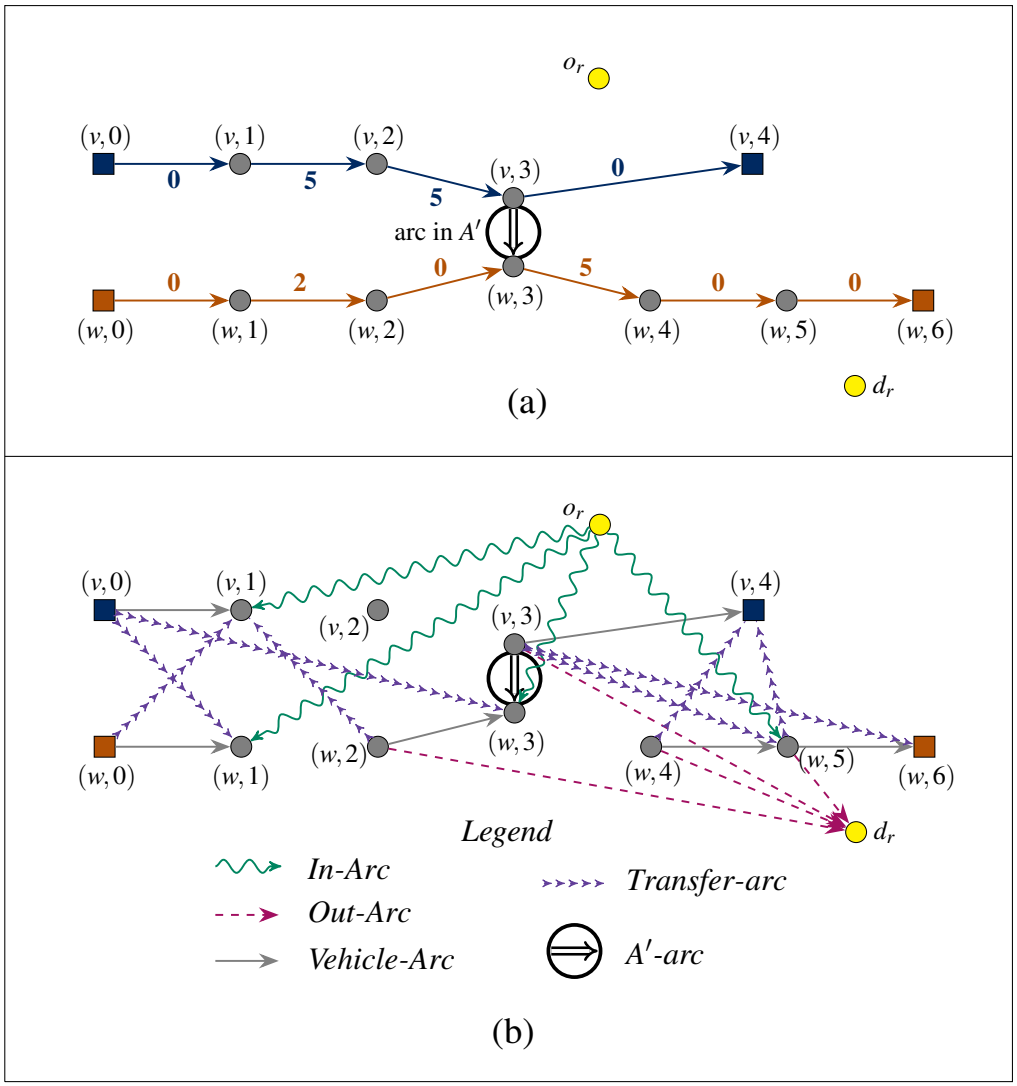


Figure 4.13: Deriving digraph $H(\Gamma, \Pi, r)$ from the PDPT schedule (Γ, Π) corresponding to Figure 4.11, and one additional request $r = (o_r = D, d_r = B, \ell_r = 9)$. (a) The digraph $G(\Gamma, \Pi)$ and the additional request $r = (o_r, d_r, \ell_r)$. (b) The digraph $H(\Gamma, \Pi, r)$. Notice that the capacity requirement forbids some vehicle-arcs of $G(\Gamma, \Pi)$.

If we now look back to previous Section 4.2 we see that, the restriction of the digraph $H(\Gamma, \Pi, r)$ to what we have called its A'-arcs and its vehicle-arcs, is a virtual graph: the route collection Γ corresponds to the collection Λ, whilst o_r and d_r correspond to $\hat{o}$ and $\hat{d}$, respectively.

Table 4.1 show how to interpret the A'-arcs and vehicle-arcs of digraph $H(\Gamma, \Pi, r)$ as arcs of a virtual graph. Also, we note that the cases depicted in Figure 4.12 (a)-(h) correspond, respectively, to the constructions of the virtual graph arcs illustrated in Figure 4.3 (a)-(h).

Table 4.1: Interpretation of the vehicle's moves in Figure 4.12 as virtual graph arcs.

Type of move♣	Interpretation as an arc of the virtual graph
(a)	$\big(o,(v,i+1),\texttt{Nil},((v,i),(v,i+1))\big)$
(b)	$\big((v,i),d,((v,i),(v,i+1)),\texttt{Nil}\big)$
(c)	real-arc
(d)	real-arc
(e)	$\big((v,i),(w,j),((v,i),(v,i+1)),((w,j-1),(w,j))\big)$
(f)	$\big(o,d,((v,i),(v,i+1)),((v,i),(v,i+1))\big)$
(g)	$\big(o,(w,j),((v,i),(v,i+1)),((w,j-1),(w,j))\big)$
(h)	$\big((v,i),d,((v,i),(v,i+1)),((w,j-1),(w,j))\big)$
(i)	$\big(o,d,((v,i),(v,i+1)),((w,j-1),(w,j))\big)$

♣According to Figure 4.12.

Let π be a path from o_r to d_r in $H(\Gamma,\Pi,r)$. Note that π may be interpreted as a sequence of moves of types (1)-(5) allowing to transport the request r from o_r to d_r. If τ_i^v denotes the time when vehicle v leaves vertex (v,i), then every in-arc, out-arc and transfer-arc of π is going to impose additional constraints, to be added to constraints (C2):

- An in-arc $\big(o_r,(v,i)\big)$ imposes one additional constraint

$$\tau_{i+1}^v \geq \tau_i^v + time(x_i^v,o_r) + time(o_r,x_{i+1}^v); \tag{C5}$$

- An out-arc $\big((w,j),d_r\big)$ imposes one additional constraint

$$\tau_{j+1}^w \geq \tau_j^w + time(x_j^w,d_r) + time(d_r,x_{j+1}^w); \tag{C6}$$

- A transfer-arc $\big((v,i),(w,j)\big)$ imposes three additional constraints (here, z is taken as in (C4)):

$$\left.\begin{array}{l} 1.\ \tau_{i+1}^v \geq \tau_i^v + time(x_i^v,z) + time(z,x_{i+1}^v);\ (\textit{Deviation for vehicle } v)\ \text{(C7.1)} \\ 2.\ \tau_j^w \geq \tau_{j-1}^w + time(x_{j-1}^w,z) + time(z,x_j^w);\ (\textit{Deviation for vehicle } w)\ \text{(C7.2)} \\ 3.\ \tau_j^w \geq \tau_i^v + time(x_i^v,z) + time(z,x_j^w).\ (\textit{Weak Synchronization})\ \text{(C7.3)} \end{array}\right\} \text{(C7)}$$

Then a path π in $H(\Gamma,\Pi,r)$ is going to be *time-consistent* if it allows the existence of time vectors $\mathbf{t}^v = (\tau_0^v,\ldots,\tau_{|\Gamma(v)|-1}^v)$, $v \in V$, which meet constraints (C1) and additional constraints (C5, C6, C7) related to π. Notice that if π is given, checking the time consistence of π can be checked by constraint propagation.

Now the 1-Request Insertion PDPT can be summarized in the following way.

***1-Request Insertion PDPT*:** Given a PDPT schedule (Γ, Π) and one request $r = (o_r, d_r, \ell_r)$, compute a time-consistent path π of minimum weight (with respect to the weight function $time : A(H(\Gamma, \Pi, r)) \to \mathbb{R}_+$) from o_r to d_r in the digraph $H(\Gamma, \Pi, r)$.

As we have previously said, this 1-Request Insertion PDPT can be seen as a particular case of the Virtual Path Problem that we have introduced in Section 4.2. So, we can use the Virtual A* algorithm for solving the 1-Request PDPT in an exact way. Still, there are also some structural properties of the problem that can be exploited to obtain an improved implementation of the Virtual A* in this particular case. We omit the implementation details. They can be found in the article of Figueroa et al. (2022) [86].

4.3.4 An Empirical Dijkstra-Like Algorithm

Since our problem is likely to arise in a dynamic context, we must try to propose approaches which are less time consuming than the previous one. In order to do it, we fix some integer parameter $m > 0$ and next for every (v, i) with $v \in V$, $i \in \{0, \ldots, |\Gamma(v)| - 1\}$, we compute minimum weight paths $\pi(v, i)$ (with respect to the weight function $time_{H(\Gamma,\Pi,r)}$) in the digraph $H(\Gamma, \Pi, r)$, from vertex (v, i) to destination d_r, while using Dijkstra's algorithm (see Figure 4.14 (a)). We denote by $\omega(v, i)$ the *time* weight of $\pi(v, i)$. Notice that such computations are involved in the preprocess of the virtual A* algorithm in order to provide us with the values $\texttt{W1}[x]$ for all x in X^*.

Then, we close those computed paths $\pi(v, i)$ (see Figure 4.14 (b)) to obtain o_r-d_r -paths $\pi^*(v, i) = (o_r, (v, i)) + \pi(v, i)$ (i.e., we create a path from o_r to d_r whose first arc is the in-arc $(o_r, (v, i))$ and the remaining arcs are the ones of path $\pi(v, i)$). For each constructed path $\pi^*(v, i)$, we compute the value $time\big(o_r, (v, i)\big) + \omega(v, i)$, and then we select the m paths $\pi^*(v_1, i_1), \ldots, \pi^*(v_m, i_m)$ with the lowest values $time\big(o_r, (v, i)\big) + \omega(v, i)$. Finally, we check for every selected path $\pi^*(v_k, i_k)$, $1 \leq k \leq m$, the consistence of the additional constraints induced by the in-arc, the out-arc and the transfer-arcs of $\pi^*(v_k, i_k)$, and we keep the path π^* which meets the time-consistence test and which is related to the smallest $time\big(o_r, (v, i)\big) + \omega(v, i)$ value. This process is summarized in Algorithm 9.

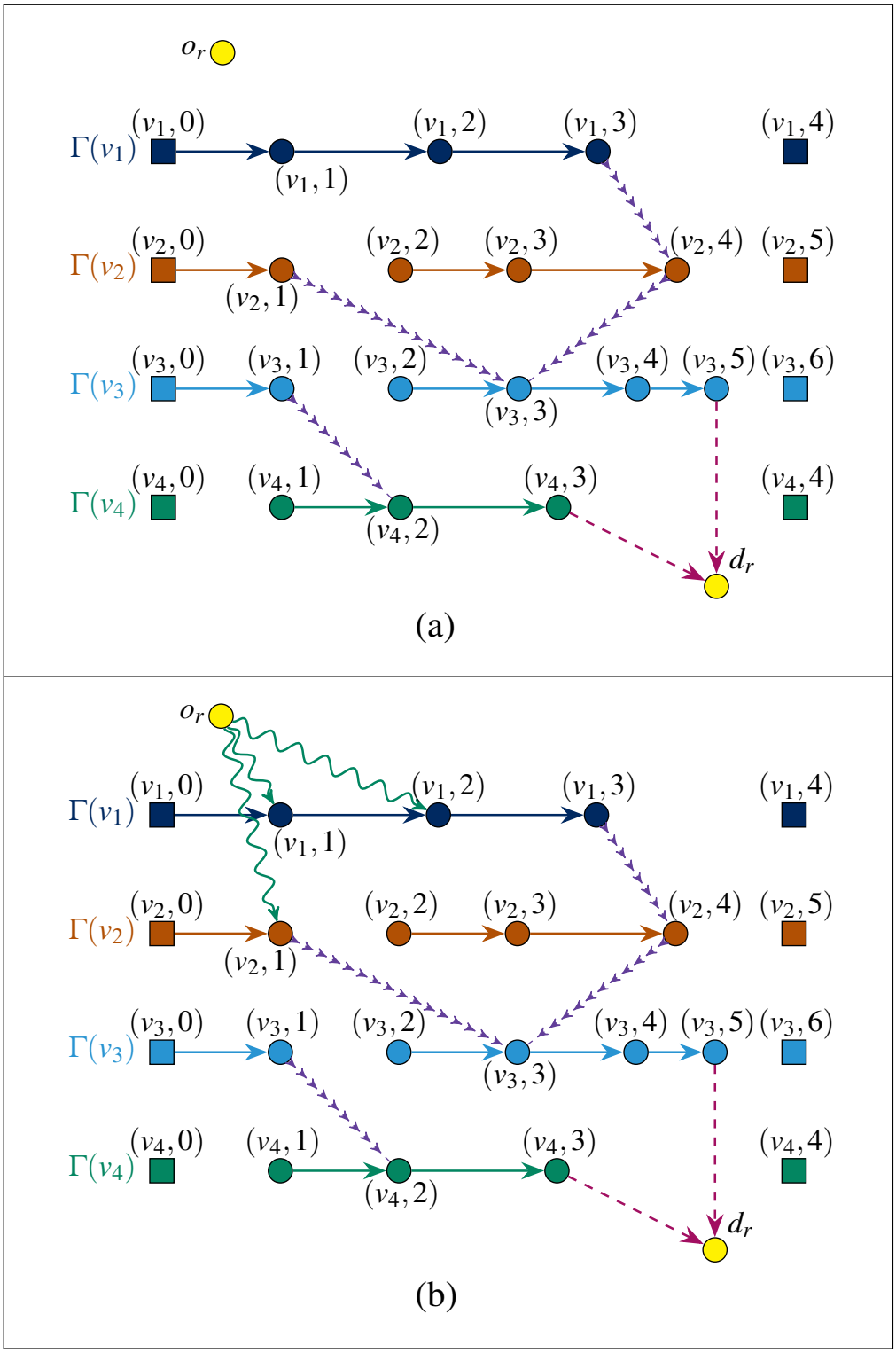


Figure 4.14: Deriving solutions from a Dijkstra's minimum weight path computation. (a) A subgraph Ψ of a digraph $H(\Gamma, \Pi, r)$ obtained by Dijkstra's algorithm while computing a minimum weight path $\pi(v, i)$ (with respect to the weight function $time_{H(\Gamma,\Pi,r)}$) from every vehicle vertex (v, i) to d_r in $H(\Gamma, \Pi, r)$. (b) Digraph Ψ is extended by adding the m in-arcs with smallest *time* weight in $H(\Gamma, \Pi, r)$ (wavy arrows). Every path from o_r to d_r in the resulting digraph gives rise to a solution for handling request r and the time-feasibility of such a solution is then checked by constraint propagation.

Algorithm 9: Dijkstra 1-PDPT algorithm

input : A digraph $H(\Gamma, \Pi, r)$.

output: A time-consistent o_r-d_r-path π^*, or a failure message.

1 $\pi^* \leftarrow$ Nil ;

2 For every (v, i) with $v \in V$ and $i \in \{0, \ldots, |\Gamma(v)| - 1\}$, compute through Dijkstra's algorithm, minimum *time* weigth paths $\pi(v, i)$ from (v, i) to destination d_r, together with their corresponding weight $\omega(v, i)$ in the digraph $H(\Gamma, \Pi, r)$;

3 Close every path $\pi(v, i)$ computed in 2 by adding the corresponding in-arc $\big(o_r, (v, i)\big)$ and select the m resulting paths $\pi^*(v_1, i_1), \ldots, \pi^*(v_m, i_m)$ with best $time\big(o_r, (v, i)\big) + \omega(v, i)$ values;

4 For every path $\pi^*(v_k, i_k), 1 \leq k \leq m$ selected in 3, test its time-consistence (through constraint propagation) and keep as π^* the time-consistent path $\pi^*(v_k, i_k)$ with best $time\big(o_r, (v_k, i_k)\big) + \omega(v_k, i_k)$ value;

5 **if** π^* = Nil **then**

6 Print failure message "No path found"

7 **else**

return : π^*

4.3.5 Controlling Transfer-Arcs Number

As we have mentioned previously, the number of transfer-arcs is an issue since most arcs of $H(\Gamma, \Pi, r)$ are transfer-arcs, while at the end, the best time-consistent paths usually contain very few of such arcs. In order to deal with the number of transfer-arcs issue, we impose to transfer-arcs some eligibility requirement, which depends on a threshold parameter $\varepsilon \geq 0$.

ε-eligibility of a transfer-arc $\big((v, i), (w, j)\big)$. We say that a transfer-arc $a = \big((v, i), (w, j)\big)$ is *ε-eligible* if $time(x_i^v, x_j^w) \leq \varepsilon$ and the intersection of time windows $[Min_{(v,i)} + time(a),\ Max_{(v,i)} + time(a)]$ and $[Min_{(w,j)},\ Max_{(w,j)}]$ is nonempty.

By limiting the maximum number of allowed transfers to a small fixed number (e.g. 3 or 5), by fixing ε, and by imposing transfer-arcs to be ε-eligible, we become able to control running times of both virtual A* and Dijkstra 1-PDPT algorithms. As Section 4.3.6 will show, for most of the considered instances, this control is not going to induce any significant loss of solutions quality.

4.3.6 Numerical Experiments

Purpose. We have been performing experiments to:

1. analyze the performance of both virtual A* and Dijkstra 1-PDPT algorithms from the point of view of running time costs and, in the case of Dijkstra 1-PDPT, of gap to optimality;
2. evaluate the impact of the ε-eligibility on those algorithms;
3. estimate about the potential interest of transfers, through both the number of transfer-arcs involved in an optimal path π, and the gain induced by those arcs.

Instances. Points of X are randomly generated on an integral grid with size $n \times n$, distance function *time* is the upper rounding of the Euclidean distance or the taxicab distance. The upper limit of the time horizon $[0, \Omega]$ is indicated by Ω and is generated as a linear function of n, also we will consider a fixed capacity $\kappa = 10$.

While designing the paths, we follow the following strategies.

- Strategy Free: For every $v, w \in V$ we that $start(v) = end(v) = start(w) = end(w)$. This is, all of the vehicles start and end their paths in a common depot vertex. Paths $\Gamma(v)$ are generated in a pseudorandom way;
- strategy Closed: For every $v, w \in V$ we have $start(v) = end(v)$ and $start(w) = end(w)$. This is, every vehicle starts and ends its path in a same depot vertex, but different vehicles may have distinct depot vertices. Paths $\Gamma(v)$ are generated by following some linear or circular orientation;
- strategy Open: Every vehicle v starts its path in a starting depot vertex $start(v)$ and ends its path in an ending depot vertex $end(v)$ (possibly different from $start(v)$); furthermore, different vehicles may have distinct starting depot vertices and different ending depot vertices. Paths $\Gamma(v)$ are generated by following some linear or circular orientation.

The designed paths have around ten arcs on average, and the arcs loads are generated in a uniform pseudorandom way by choosing elements from set $\{0, 1, \ldots, 10\}$. Another important path feature is the mean ratio $\rho = \frac{|V| \cdot \Omega}{\sum_{v \in V} \text{cost of } \Gamma(v)}$, which gives us an approximate idea about the difficulty of an instance.

The sets A' are created in the following way: for every $(v, i) \in \Gamma(v)$ and every $(w, j) \in \Gamma(w)$, such that $x_i^v = x_i^w$ and $\big((w, j), (v, i)\big) \notin A'$ we create an arc

$\big((v,i),(w,j)\big)$ and we add it to A' with a probability of 0.7 if the resulting digraph is acyclic and does not imply exceeding the time horizon.

Requests are generated also in a pseudorandom way, that is, we have selected two different points o and d from X, and a random load value ℓ from set $\{1,2,3,4\}$.

In this first set of experiments we are going to consider the distance traversed by the request as the cost to optimize.

Every instance is summarized by the side's length n of the squared integer grid, the number $|X|$ of points, the distance function *time* (each instance is tested with the Euclidean and taxicab distances), the number of vehicles $|V|$, the number $|A'|$ of arcs in A', the upper limit Ω of the time horizon, the mean ratio ρ, the path generation strategy strat, and a request $(o,d,\ell) \in X \times X \times \{1,2,3,4\}$.

Instances which are going to be used here are summarized by the rows of Table B.1 in the Appendix of this document. In general, the taxicab distance between two points in $\mathbb{R}^2$ is greater than or equal to the corresponding Euclidean distance between the two points. As a consequence, an instance is usually more time-constrained when we use the taxicab distance than when we use the upper rounded Euclidean distance. In particular, almost all of the instances in Table B.1 admit solutions without transfers when we consider the Euclidean distance. To attenuate the effect of that property, we have chosen to reduce by ten time units the time horizon of any instance with the Euclidean distance.

Outputs. For any instance we compute the following.

1. The value No Transfer of the solution (obtained through enumeration) considering direct-arcs and forbidding transfer-arcs. The value VAL A* computed by virtual A*, together with related running time CPU A* in seconds (including the time for preprocessing, the time for testing the direct-arcs, and the running-time of the Dijkstra 1-PDPT algorithm), the number Transfer A* of transfer arcs of the solution. The value VAL DI computed by Dijkstra 1-PDPT, together with related running time CPU DI in seconds (without including preprocessing), the number Transfer DI of transfer arcs of the solution. The results of these tests are shown in Table B.2 of the Appendix.

2. The same corresponding values obtained when limiting the maximum number of allowed transfers to three. These results are shown in Table B.3 of the Appendix.

3. The same corresponding values obtained while taking the eligibility threshold ε as 1.0 n, 0.50 n, and 0.25 n.

Technical context. Experiments were performed on a computer with a 2.7 GHz Intel Core i5 processor and 8 GB 1866 MHz RAM. The implementations were built in C++ 11 by using the Apple Clang compiler 13. Note that some values are not available due to the absence of found solutions, in such cases the missing values are indicated by NA (not available).

Results and Comments

Figure 4.15 shows the boxplots of running times for the tests without any eligibility restriction. We note that the associated running times of both algorithms can be considered "acceptable" for most of the examined instances. Although it is well known that Dijkstra's algorithm has a time complexity that is always polynomially bounded on the size of the input digraph, for the virtual A* this is not the case; this is suggested by the extremal values of CPU A* appearing in Figure 4.15.

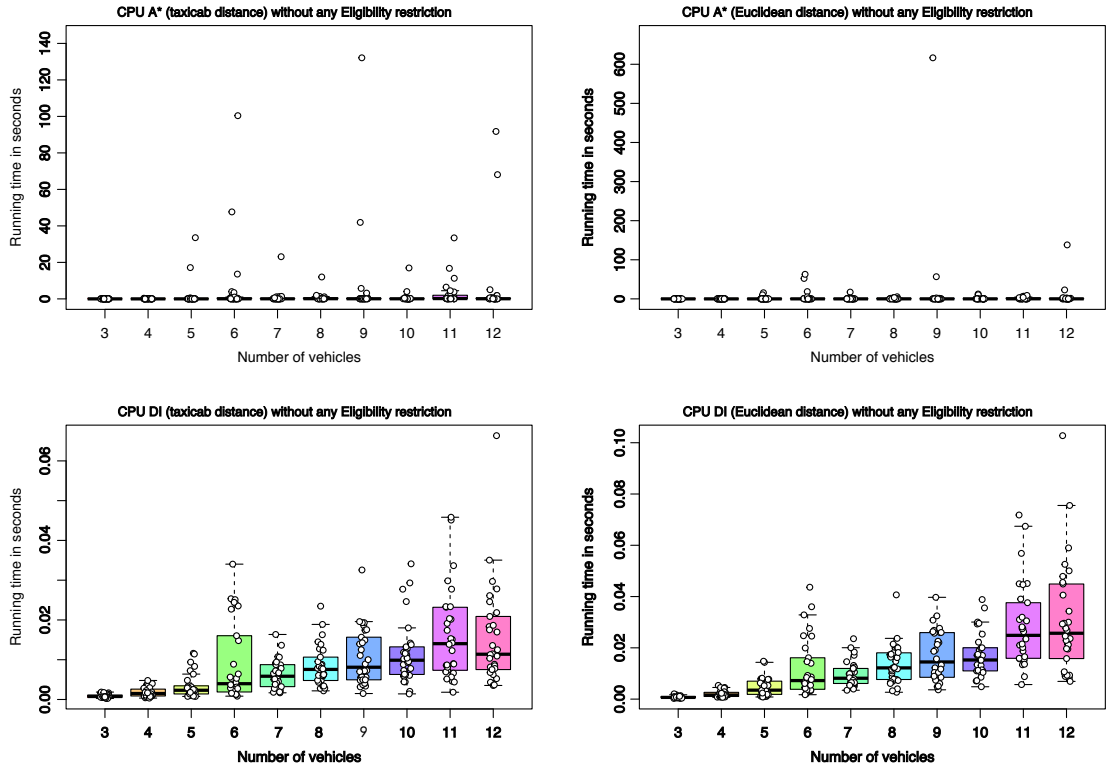


Figure 4.15: Boxplots of running times for the algorithms virtual A* and Dijkstra 1-PDPT when executed without any eligibility restriction on the 300 instances of Table B.1 of the Appendix. The small circles (i.e., jittered points) correspond to individual data values, the horizontal black line dividing each box corresponds to the median value in the category (i.e, the Q_2 quartile), the lower (respectively the upper) side of a box indicates the Q_1 (respectively the Q_3) quartile. The horizontal line below (respectively above) each box corresponds to the Q_0 (respectively the Q_4) quartile.

However, if we limit the maximum number of allowed transfers to a small constant (e.g. two or three transfers at most), the complexity decreases and we achieve running times that are polynomially bounded on the size of the input digraph. This is confirmed by the boxplots in Figure 4.16 that correspond to the results from Table B.3 of the Appendix; these results were obtained by limiting the maximum number of allowed transfers to three. Note that, in comparison with the values in Figure 4.15, the maximum extremal values for the virtual A* algorithm have decreased in at least one order of magnitude.

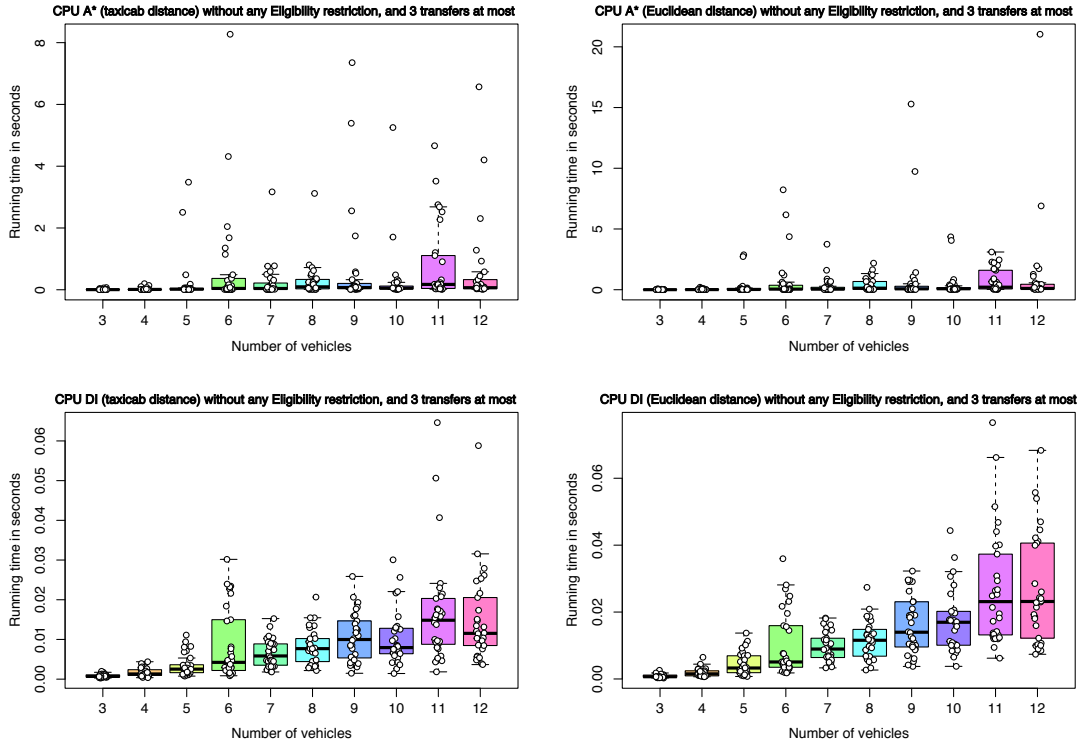
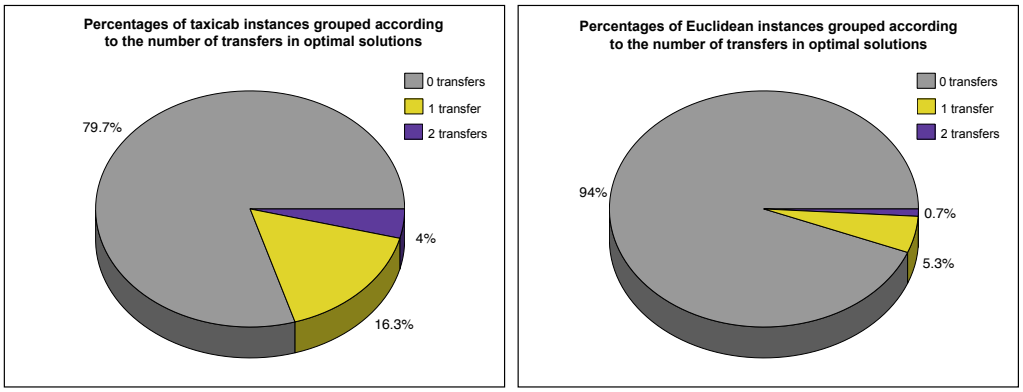


Figure 4.16: Boxplots of running times for the algorithms virtual A* and Dijkstra 1-PDPT, limiting to three the maximum number of allowed transfers, and executing those algorithms without any eligibility restriction on the 300 instances of Table B.1 of the Appendix. The small circles (i.e., jittered points) correspond to individual data values, the horizontal black line dividing each box corresponds to the median value in the category (i.e, the Q_2 quartile), the lower (respectively the upper) side of a box indicates the Q_1 (respectively the Q_3) quartile. The horizontal line below (respectively above) each box corresponds to the Q_0 (respectively the Q_4) quartile.

Furthermore, we can verify from the results in Table B.2 that, for the considered instances, the optimal solutions have required two transfers at most (see Figure 4.17), so the solutions obtained when limiting the maximum number of allowed transfers to three (or even two) are still optimal solutions.



(a) (b)

Figure 4.17: Pie charts showing the classification the instances in Table B.1 of the Appendix according to the number of transfers in the optimal solutions found by the virtual A* algorithm when executed without any eligibility restriction. (a) Classification of the instances with the taxicab distance. (b) Classification of the instances with the upper rounded Euclidean distance.

Regarding the quality of the solutions computed by the virtual A*, Figure 4.18 compares the sum of the costs of all instances admitting solutions without transfers. We can see the way in which the solution costs are increased when we reduce the parameter ε of transfer-arc eligibility.

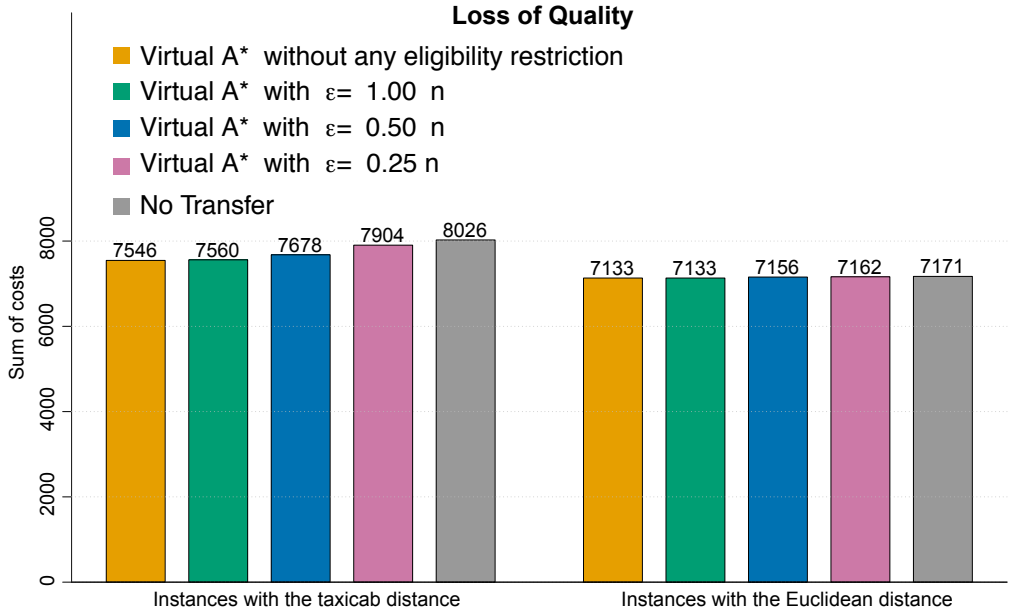


Figure 4.18: Sum of costs for the algorithm virtual A* executed with several values ε of transfer-arcs eligibility. These tests were performed on the instances of Table B.1 that admit solutions without transfers. For an easier comparison, we have added a bar corresponding to the sum of costs of the optimal solutions without transfers.

With respect to the effectiveness, Figure 4.19 shows how the percentage of instances solved by the A* algorithm decreases when we reduce the ε parameter.

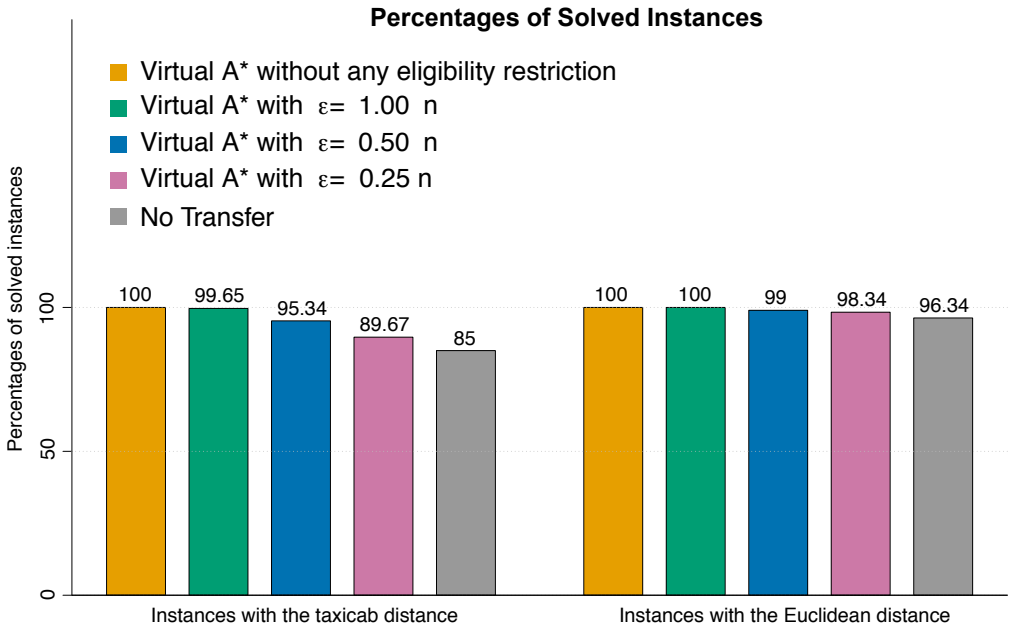


Figure 4.19: Percentages of solved instances by the algorithms virtual A* executed with several parameters ε of transfer-arcs eligibility. These tests were performed on all the instances of Table B.1 of the Appendix.

Figure 4.20 compares the sum of costs obtained by the enumeration without transfers, the Dijsktra 1-PDPT, and the virtual A* algorithms executed on all the instances in Table B.1 that admit solutions without transfers.

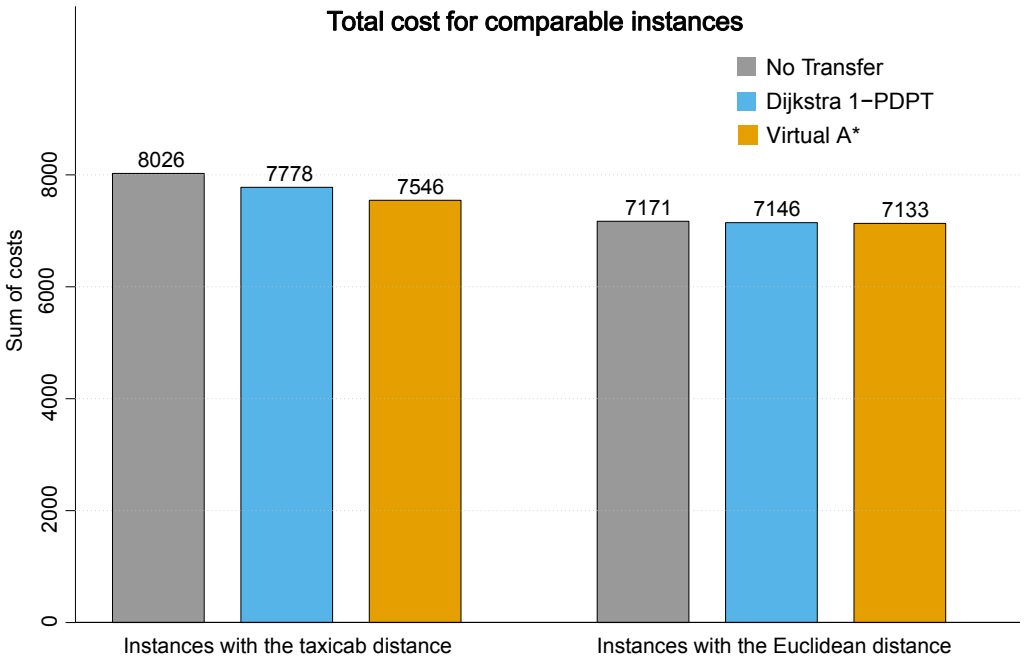


Figure 4.20: Sum of solution costs computed by enumeration without transfers, Dijkstra 1-PDT, and virtual A* algorithms executed without any eligibility restriction on the instances of Table B.1 that admit solutions without transfers.

Figure 4.21 shows the percentages of instances solved by enumeration without transfers, Dijkstra 1-PDPT, and virtual A* without any eligibility restriction. These results confirm the worthiness of the Dijkstra 1-PDT algorithm to search for good quality solutions in an acceptable amount of time.

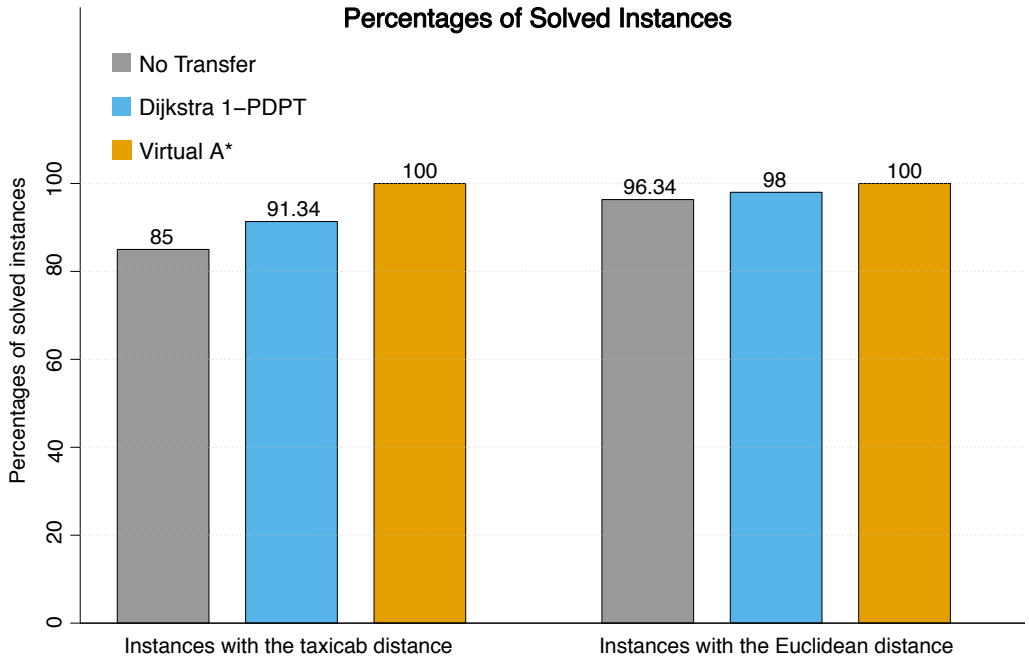


Figure 4.21: Percentages of solved instances by enumeration without transfers, Dijkstra 1-PDPT, and virtual A* algorithms executed without any eligibility restriction. These results correspond to all the instances from Table B.1 of the Appendix.

4.3.7 Possible Extensions of the Algorithms

The Algorithms 8 and 9 can be easily adapted to handle instances with time windows. For that we only need to add a procedure for checking that those time windows are respected by a (partial) solution.

Also, we can handle the dynamic version of the PDPT by "taking" a picture of the computed schedule at a current time $t \in (0, \Omega)$. That is, for every $v \in V$ and every $(v, i) \in \Gamma(v)$, we discard in digraph $G(\Gamma, \Pi, r)$ all the vertices (v, i) with $Min_{(v,i)} < t$ and maintain the current arc load values.

If at time t a vehicle v is at some location x' (here either $x' \in X$ or x' may be a new auxiliary point), we redefine $(v,0) = x'$ and:

- if (at time t) vehicle v is running through an arc $a = \big((v,i)(v,i+1)\big)$ then we replace (v,i) by x', the arc a is redefined as $a = \big((v,0) = x', (v,1) = x^v_{i+1}\big)$, and the weight $time(a)$ is equal to the time the vehicle v needs to reach x^v_{i+1} from x';
- if (at time t) vehicle v is waiting for a request at a vertex (v,i), we only need to add an auxiliary arc $a = \big((v,0) = x', (v,1) = x^v_i\big)$ with $time(a)$ defined as the time that remains v to wait at (v,i) from the current time t.

Next we proceed to reindex all the tour vertices and we take resulting digraph as the digraph $H(\Gamma, \Pi, r)$. We also update *time* and *relay* functions to consider also the (auxiliary) point x', we replace time horizon $[0,\Omega]$ by $[0,\Omega - t]$, and for every $v \in V$ and $(v,i) \in \Gamma(v)$ we replace $Min_{(v,i)}$ by $Min_{(v,i)} - t$.

4.4 Handling Multiple Requests

The main difficulty for handling multiple requests into an insertion approach for the PDPT, resides in the correct design and implementation of the data structures that allow us to keep track of the evolution of a given PDPT schedule when we perform alternately some sequences of insertions and deletions of requests. In particular, at any time of the insertion/deletion process, we should be able to retrieve:

- the requests that are currently inserted, together with the order in which they were inserted;
- the collections Γ and Π corresponding, respectively, to the routes followed by the vehicles, and the routes followed by the inserted requests;
- the vertices and A'-arcs that were part of the given input instance;
- the transfer-arcs that have been created, together with the inserted requests that are using them;
- the load of every vehicle-arc.

For the implementation of our data structures, we are going to proceed (like in the case of the insertion of a single request into a given PDPT schedule (Γ, Π)) by creating the digraph $G(\Gamma, \Pi)$ described in Section 4.3. We will take $\Gamma(v) \in \Gamma$ as a linked list that contains the sequence of vertices followed by the vehicle v. Also, every arc $a = \big((v,i),(v,i+1)\big)$ of $\Gamma(v)$ has associated a load ℓ_i^v.

Now, for every (v,i) in $\Gamma(v)$ we associate:

- an element `point`$[(v,i)]$ in X, corresponding to the physical location of vertex (v,i);
- an array of labels `info`$[(v,i)]$, which allows us to determine if (v,i) was given as a part of the initial input instance or if it is involved in the path traversed by some currently inserted request.

In our implementation `info`$[(v,i)]$ may contain the following types of labels:

- a label (`Nil`, "`fixed-vertex`"), if (v,i) was given as a part of the initial input instance;
- a label (r, "`tail in-arc`") (respectively (r, "`tail` A'`-arc`")), if (v,i) was the tail of an in-arc (respectively an A'-arc) that was used during the insertion of the request r.
- a label (r, "`head out-arc`") (respectively (r, "`head` A'`-arc`")), if (v,i) was the head of an out-arc (respectively an A'-arc) that was used during the insertion of the request r.
- a label (r, "`relay vertex in emitting tour`"), if (v,i) corresponds to the relay vertex involved in a transfer-arc $\big((v,i),(w,j)\big)$;
- a label (r, "`relay vertex in receiving tour`"), if (v,i) corresponds to the relay vertex involved in a transfer-arc $\big((w,j),(v,i)\big)$;
- a label (r, "`origin direct-arc`"), if (v,i) corresponds to the origin of a request that was inserted using a direct-arc;
- a label (r, "`destination direct-arc`"), if (v,i) corresponds to the destination of a request that was inserted using a direct-arc.

We maintain a linked list `existent_transfers` to keep track of all the A'-arcs that either were initially on the given input instance or that were created as transfer-arcs during the insertion of some currently inserted request r. An element of this list contains a pair (a, `involved_requests`$[a]$) where a is an A'-arc of the current PDPT schedule and `involved_requests`$[a]$ is a list that indicates the currently inserted requests that are using arc a as an arc of their routes, we also add a dummy label `Nil` on the list `involved_requests`$[a]$ if a was given as an A'-arc of the initial input instance.

Now, suppose that we have just inserted a request r by using an o_r-d_r-path π_r, and that the request r follows a sequence of vertices $(v_{i_1}, j_1)\cdots(v_{i_k}, j_k)$ in the resulting PDPT schedule. Then, for every (v, i) in the sequence $(v_{i_1}, j_1)\cdots(v_{i_k}, j_k)$, we store the memory address of the element (v, i) of $\Gamma(v)$ in an array `path`$[r]$ if :

- (v, i) is the first vertex of the sequence $(v_{i_1}, j_1)\cdots(v_{i_k}, j_k)$;
- (v, i) corresponds to the tail of an A'-arc in π_r;
- (v, i) corresponds to the head of an A'-arc in π_r;
- (v, i) corresponds to the relay vertex of a transfer arc in π_r;
- (v, i) is the last vertex of the sequence $(v_{i_1}, j_1)\cdots(v_{i_k}, j_k)$.

Note that `path`$[r]$ is nothing but a simplified sequence of vertices addresses that allow us to retrieve the path $\Pi(r)$ followed by the request r in the resulting PDPT schedule. Also, for every A'-arc or transfer-arc $a = \big((v, i), (w, j)\big)$ that was used in the insertion of r, we store the memory address of the element (a, `involved_requests`$[a]$) of the linked list `existent_transfers`, in an array `transfers`$[r]$.

Every time that we insert a request r, we store the triple (r, `path`$[r]$, `transfers`$[r]$) in a list `inserted_requests`; in contrast, we use a list `remaining_requests` to maintain the requests that are not currently inserted in the PDPT schedule.

4.4.1 Deletion of an Inserted Request

Let us suppose that we are given a PDPT schedule codified by the data structures described in the previous section. Every inserted request $r = (o_r, d_r, \ell_r)$ corresponds to an element (r, `path`$[r]$, `transfers`$[r]$) of the list `inserted_requests`. For deleting r from (Γ, Π), we proceed in the following way.

1. For every arc $a = \big((v, i), (w, j)\big)$ in $\Pi(r)$, we set $\ell_i^v := \ell_i^v - \ell_r$.

2. For every vertex (v, i) whose address is stored in the array `path`$[r]$, we remove from `info`$[(v, i)]$ any label involving request r. If `info`$[(v, i)]$ becomes empty, then we remove vertex (v, i) from $\Gamma(v)$.

3. For every entry in the list `transfers`$[r]$ we have the address of an element $(a,$ `involved_requests`$[a])$ of the list `existent_transfers`. We remove the label r from `involved_requests`$[a]$. If `involved_requests`$[a]$ becomes empty, then we remove the element $(a,$ `involved_requests`$[a])$ from `existent_transfers`.

4. We remove the element (r,`path`$[r]$, `transfers`$[r]$) from `inserted_requests` and we insert $r = (o_r, d_r, \ell_r)$ into `remaining_requests`.

4.4.2 Search Algorithms

In this section we describe some well-known metaheuristics that we have implemented in combination with the virtual A* to search for a good sequence for inserting the requests into a PDPT schedule.

Let $\mathcal{I} = (X, time, relay, V, start, end, \kappa, \Omega, R)$ be a PDPT instance and let P be a permutation of the elements of R. Suppose that we have proceeded to insert the requests in the PDPT instance by following the ordering given by P while ignoring infeasible insertions. Let $(\Gamma, \Pi)^{\mathcal{I}}_P$ be the resulting PDPT schedule.

In the rest of this chapter, we are going to denote by $insertions((\Gamma, \Pi)^{\mathcal{I}}_P)$ the number of requests satisfied by $(\Gamma, \Pi)^{\mathcal{I}}_P$ and by $cost((\Gamma, \Pi)^{\mathcal{I}}_P)$ the related cost of the PDPT schedule $(\Gamma, \Pi)^{\mathcal{I}}_P$. Also, we denote by $swap(P)$ the list of all permutations of R that differ from P by the swapping of exactly two elements.

We start with the description of one of the simplest search mechanisms.

Naive GRASP Search

GRASP stands for greedy randomized adaptive search procedure. It is a process that constructs a greedy randomized solution at each iteration of the main loop. A randomized solution is usually generated by adding elements to the problem's solution set from a list of "promising" elements which are often placed in a restricted candidate list and chosen at random when building up the solution.

Here we propose a naive implementation of the GRASP search where, we simply generate a random permutation P of the requests and we use the virtual A* algorithm for trying to insert the requests in the ordering given by P and while ignoring infeasible insertions. Then we evaluate the resulting solution and we remove all

of the inserted requests. By repeating several times the above process, we obtain Algorithm 10.

Algorithm 10: Naive GRASP search algorithm for the PDPT

Input : A PDPT instance $\mathcal{I} = (X, time, relay, V, start, end, \kappa, \Omega, R)$, and an integer $n > 0$ indicating the number of iterations desired.

Output: Either a PDPT solution (Γ, Π) or a failure message.

```
1  set iteration ← 0, max_insertions ← 0, and best_cost ← +∞
2  set best_solution ← Nil
3  while iteration < n do                                              [Main loop]
4  |  Generate a random permutation P of the requests in R
5  |  Apply the virtual A* algorithm to I for inserting the requests, according to the
   |   ordering given by P and ignoring infeasible insertions
6  |  if (max_insertions < insertions((Γ,Π)^I_P')) or ((max_insertions =
   |   insertions((Γ,Π)^I_P')) and (best_cost > cost((Γ,Π)^I_P'))) then
7  |  |  max_insertions ← insertions((Γ,Π)^I_P)
8  |  |  best_cost ← cost((Γ,Π)^I_P)
9  |  |  best_solution ← (Γ,Π)^I_P
10 |  Remove from (Γ,Π)^I_P all the inserted requests                   [Restore I]
11 |  set iteration ← iteration + 1
12 if best_cost < +∞ then                                              [If any solution]
13 |  return best_solution
14 else                                                                [If no solutions]
15 |  Print "No solution found"                                        [Print failure message]
```

Random Walk Search

This metaheuristic is based on a random local search. We start with a permutation P of the elements of R. Then, we use the virtual A* algorithm for trying to insert the requests in the ordering given by P, while discarding infeasible insertions. Next, we remove all the inserted requests, we replace P by a random element of $swap(P)$, and we repeat the whole process with that new permutation.

Descent Search

This is a local search metaheuristic with a greedy component. We start with a permutation P of the elements in R. Then, we use the virtual A* algorithm for trying to insert the requests in the ordering given by P, while discarding infeasible insertions. Next, we remove all the inserted requests and we generate, one bye one, the elements of $swap(P)$. If we find a permutation $P' \in swap(P)$ such that insertions$((\Gamma, \Pi)^{\mathcal{I}}_{P'})$ > insertions$((\Gamma, \Pi)^{\mathcal{I}}_{P})$ or such that insertions$((\Gamma, \Pi)^{\mathcal{I}}_{P'})$ =

insertions$((\Gamma,\Pi)_P^{\mathcal{I}})$ and $cost((\Gamma,\Pi)_{P'}^{\mathcal{I}}) < cost((\Gamma,\Pi)_P^{\mathcal{I}})$ then we replace P by P', we remove all the inserted requests and we repeat the whole process, otherwise we stop the search.

Simulated Annealing

The simulated annealing is a local search metaheuristic of general purpose that was proposed by Kirpatrick, Gelett and Vecchi [139] and Cerny [224] for approximating the global optimum of a cost function with possibly multiple local optima. The name of the algorithm comes from annealing in metallurgy, which is a technique involving heating and controlled cooling of a material to alter its physical properties.

There exists many variants of the simulated annealing algorithm and in most applications the success of the algorithm is very sensitive against the choice of the input parameters. Here, we have implemented a variant of simulated annealing that is known by the name of threshold accepting. The reason of our choice is based on the empirical results obtained by Dueck and Scheuer (1990) [74] for some optimization problems like the Traveling Salesperson Problem.

Another interesting property of the threshold accepting algorithm is that the search is driven by tunning a set of meaningful parameters. We describe those parameters next.

Initial temperature. This parameter is used as an initial quality threshold for discarding bad cost solutions. At the beginning of the threshold accepting algorithm, we declare a variable `temperature` which is initialized to the value of this parameter. Then during the main loop of the algorithm the `temperature` value decreases gradually.

Freezing point. This is a termination parameter. The algorithm finishes when the value of the variable `temperature` is less than or equal to the value of this input parameter.

Batch size. This input parameter is related to the time during which the threshold accepting maintains fixed the current quality threshold. Each time the algorithm completes a batch of solutions with acceptable quality, compares the average cost of current batch with the average cost of previous batch. If the average cost is not improving, the algorithm reduces the value of variable `temperature`.

Cooling factor. This input parameter is usually a rational number in the interval $(0, 1)$, and is used to reduce gradually the value of variable `temperature`. Such a reduction is performed by replacing the current value of variable `temperature` by its product with the cooling factor. Therefore, the cooling factor parameter is related directly to the intensity of the search (i.e., the more close the cooling factor is to 1, the more intensive the search is) and indirectly to the total running time of the threshold accepting algorithm.

During an iteration of the main loop we have a current permutation P of the requests and we explore the elements P' of $swap(P)$ in a random order. Each time that we find an "acceptable" permutation P', we sum its cost to the cost of the current batch. Then we take P' as the new current permutation and we start a new iteration.

When a batch of accepted solutions is completed, we compare its cost against the cost of the previous one. If the cost of the current batch is not "better", then we multiply the `temperature` by the cooling factor; otherwise we maintain the current temperature. Next we start the collection of a new batch of accepted solutions.

Given the cost function that we are considering, we say that a permutation P' is acceptable in comparison with a permutation P in the following cases.

- If $insertions((\Gamma, \Pi)^I_{P'}) > insertions((\Gamma, \Pi)^I_P)$.
- If $insertions((\Gamma, \Pi)^I_{P'}) = insertions((\Gamma, \Pi)^I_P)$ and $cost((\Gamma, \Pi)^I_{P'}) < cost((\Gamma, \Pi)^I_P)$+`temperature`.
- If $insertions((\Gamma, \Pi)^I_{P'}) = insertions((\Gamma, \Pi)^I_P) - 1$ and $cost((\Gamma, \Pi)^I_{P'}) < cost((\Gamma, \Pi)^I_P)$−`temperature`.

Analogously, we say that the cost of batch of accepted solutions A is better than the cost of a batch of accepted solutions B in the following cases.

- If $\sum_{P \in A} insertions((\Gamma, \Pi)^I_P) > \sum_{P \in B} insertions((\Gamma, \Pi)^I_P)$.
- If $\sum_{P \in A} insertions((\Gamma, \Pi)^I_P) = \sum_{P \in B} insertions((\Gamma, \Pi)^I_P)$ and $\sum_{P \in A} cost((\Gamma, \Pi)^I_P) < \sum_{P \in B} cost((\Gamma, \Pi)^I_P)$.

Algorithm 11 shows a pseudocode of the resulting threshold accepting algorithm.

Algorithm 11: Threshold accepting algorithm for the PDPT

Input : A PDPT instance $\mathcal{I}$ with a set of requests R, and numerical parameters: `batch_size`, `initial_temperature`, `cooling_factor`, and `freezing_point`.

Output: Either a PDPT solution (Γ, Π) or a failure message.

1 **set** `previous_batch_cost` $\leftarrow 0$, `previous_batch_insertions` $\leftarrow 0$, `current_batch_cost` $\leftarrow 0$, `current_batch_insertions` $\leftarrow 0$, `accepted_solutions` $\leftarrow 0$, `max_insertions` $\leftarrow 0$, and `explored_neighbors` $\leftarrow 0$, `best_solution` $\leftarrow$ `Nil`, and `best_cost` $\leftarrow +\infty$

2 **set** `temperature` $\leftarrow$ `initial_temperature`

3 **set** `dynamic_equilibrium` $\leftarrow$ `false`

4 Generate a random permutation P of the requests in R

5 Apply the virtual A* algorithm to $\mathcal{I}$ for inserting the requests, according to the ordering given by P, and while ignoring infeasible insertions

6 **while** `temperature` > `freezing_point` **do** `[Main loop]`

7 `dynamic_equilibrium` $\leftarrow$ `false`

8 **while** not `dynamic_equilibrium` **do**

9 `explored_neighbors` $\leftarrow 0$

10 **if** $\Big($(`max_insertions` < $insertions((\Gamma,\Pi)^{\mathcal{I}}_P)$) or ((`max_insertions` = insertions$((\Gamma,\Pi)^{\mathcal{I}}_P)$) and (`best_cost` > $cost((\Gamma,\Pi)^{\mathcal{I}}_P)$))$\Big)$ **then** `[Best solution?]`

11 `max_insertions` $\leftarrow insertions((\Gamma,\Pi)^{\mathcal{I}}_P)$, `best_solution` $\leftarrow (\Gamma,\Pi)^{\mathcal{I}}_P$, and `best_cost` $\leftarrow cost((\Gamma,\Pi)^{\mathcal{I}}_P)$,

12 Remove from $(\Gamma,\Pi)^{\mathcal{I}}_P$ all the inserted requests

13 Shuffle $swap(P)$

14 **for** $P' \in swap(P)$ **do** `[Explore neighbor permutations of` P`]`

15 Apply the virtual A* algorithm to $\mathcal{I}$ for inserting the requests, according to the ordering given by P', and while ignoring infeasible insertions

16 **if** $\Big(\big(insertions((\Gamma,\Pi)^{\mathcal{I}}_{P'}) > insertions((\Gamma,\Pi)^{\mathcal{I}}_P)\big)$ or $\big((insertions((\Gamma,\Pi)^{\mathcal{I}}_{P'}) = insertions((\Gamma,\Pi)^{\mathcal{I}}_P)$ and $(cost((\Gamma,\Pi)^{\mathcal{I}}_{P'}) < cost((\Gamma,\Pi)^{\mathcal{I}}_P)$+`temperature`$)\big)$ or $\big((insertions((\Gamma,\Pi)^{\mathcal{I}}_{P'}) = insertions((\Gamma,\Pi)^{\mathcal{I}}_P) - 1)$ and $(cost((\Gamma,\Pi)^{\mathcal{I}}_{P'}) < cost((\Gamma,\Pi)^{\mathcal{I}}_P)$ – `temperature`$)\big)\Big)$ **then** `[If` P' `is acceptable]`

17 `accepted_solutions` $\leftarrow$ `accepted_solutions` $+1$

18 `current_batch_cost` $\leftarrow$ `current_batch_cost` $+ cost((\Gamma,\Pi)^{\mathcal{I}}_{P'})$

19 `current_batch_insertions` $\leftarrow$ `current_batch_insertions` $+ insertions((\Gamma,\Pi)^{\mathcal{I}}_{P'})$

20 $P \leftarrow P'$

21 **break**

22 `explored_neighbors` $\leftarrow$ `explored_neighbors` $+1$

23 Remove from $(\Gamma,\Pi)^{\mathcal{I}}_{P'}$ all the inserted requests

24 **if** `accepted_solutions` = `batch_size` **then** `[If a new batch is completed]`

25 **if** $\Big($(`current_batch_insertions` < `previous_batch_insertions`) or (`current_batch_insertions` = `previous_batch_insertions`) and (`current_batch_cost` ≥ `previous_batch_cost`)$\Big)$ **then**

26 `temperature` $\leftarrow$ `cooling_factor` $*$ `temperature`

27 `dynamic_equilibrium` $\leftarrow$ `true`

28 `previous_batch_cost` $\leftarrow$ `current_batch_cost`

29 `previous_batch_insertions` $\leftarrow$ `current_batch_insertions`

30 `current_batch_cost` $\leftarrow 0$, `current_batch_insertions` $\leftarrow 0$, and `accepted_solutions` $\leftarrow 0$,

31 **if** `explored_neighbors` = $|R|(|R|-1)/2$ **then** `[If all neighbors were explored]`

32 `temperature` $\leftarrow$ `freezing_point`

33 **break**

34 **if** `best_cost` < $+\infty$ **then** `[If any feasible solution]`

35 **return** `best_solution`

36 **else** `[If no solutions]`

37 Print "No solution found" `[Failure message]`

4.4.3 Numerical Experiments

For the experiments presented in this section, we are going to consider the virtual A* algorithm without any ε-eligibility restriction, and allowing three transfers at most. Also, we are going to consider the cost as a nonnegative weighted sum of the distances traversed by the requests, the distances traversed by the vehicles, and the times at which the vehicles complete their routes.

We have been performing numerical experiments with the following objectives.

- Testing the performance of search algorithms of Section 4.4.2 to evaluate the pertinence of using them in a practical context.
- Compare the behavior of those algorithms, with different assignations of values of the cost parameters α, β, and γ, respectively, for scaling the sum of the distances traversed by the requests, the sum of the distances traversed by the vehicles, and the sum of the times at which the vehicles complete their routes.
- Compare the behavior of algorithms from the point of view of the number of inserted requests, the cost of the computed solutions, and the running times.

We examine the following values of the cost parameters: $\alpha = 1$, $\beta \in \{0, 1\}$, and $\gamma \in \{0, 1\}$. For the naive GRASP search and the random walk search we examine the results for 10, 100, and 1000 iterations. In the case of the descent search algorithm we examine the results when allowing a maximum of 10, 100, and 1000 iterations. Finally, we test Algorithm 11 with parameters: `batch_size` = 15, `temperature` = $4 \cdot n$, `cooling _factor` = 0.85, and `freezing_point` = n; where n is the size of the integral grid that was considered for constructing the instance.

If P and P' are permutations that only differ in the last k elements, and the requests are currently inserted in the order given by P. We use the following shortcut to insert the requests in the order given by P': we remove the requests corresponding to the last k elements of P and then we reinsert them in the order given by P'.

Instances. We have constructed a set of 31 instances which vary from two to twelve vehicles, and from two to fifty requests. Instances 1-26 and 30 are similar to those ones described in Section 4.3.6. Instances 6-10, 21-16, and 30 start from collections of tours consisting of one arc at most. The remaining instances start from collections of prescheduled tours with more than one arc. With exception of instances 27-29, and instance 31 (that were constructed manually), all instances were constructed in a random way. Instances that are considered in this part are summarized in Table B.4 of the Appendix.

Quality criteria. We are going to consider that a solution $(\Gamma, \Pi)_P^{\mathcal{I}}$ is better than a solution $(\Gamma, \Pi)_{P'}^{\mathcal{I}}$ either if:

- $insertions\big((\Gamma, \Pi)_P^{\mathcal{I}}\big) > insertions\big((\Gamma, \Pi)_{P'}^{\mathcal{I}}\big)$, or
- $insertions\big(\Gamma, \Pi)_P^{\mathcal{I}}\big) = insertions\big((\Gamma, \Pi)_{P'}^{\mathcal{I}}\big)$ and $cost\big((\Gamma, \Pi)_P^{\mathcal{I}}\big) < cost\big((\Gamma, \Pi)_{P'}^{\mathcal{I}}\big)$.

Technical context. Experiments were performed on a computer with a 2.7 GHz Intel Core i5 processor and 8 GB 1866 MHz RAM. The implementations were built in C++ 11 by using the Apple Clang compiler 13.

Results and Comments

Results are summarized in Tables B.5, B.6, B.7, and B.8 of the Appendix. Of course, there are several ways of using those results for comparing the examined algorithms. Here, we start by comparing them via the results obtained by executing the Algorithm 10 (naive GRASP search) with 1000 iterations. That is a reasonable choosing for an empirical comparison because we have considered the same fixed random seed and therefore, we would be comparing the results for the same set of random permutations.

Effectiveness. Figure 4.22 shows the percentages of inserted requests, obtained by performing the Algorithm 10 with 1000 iterations over each instance in Table B.4.

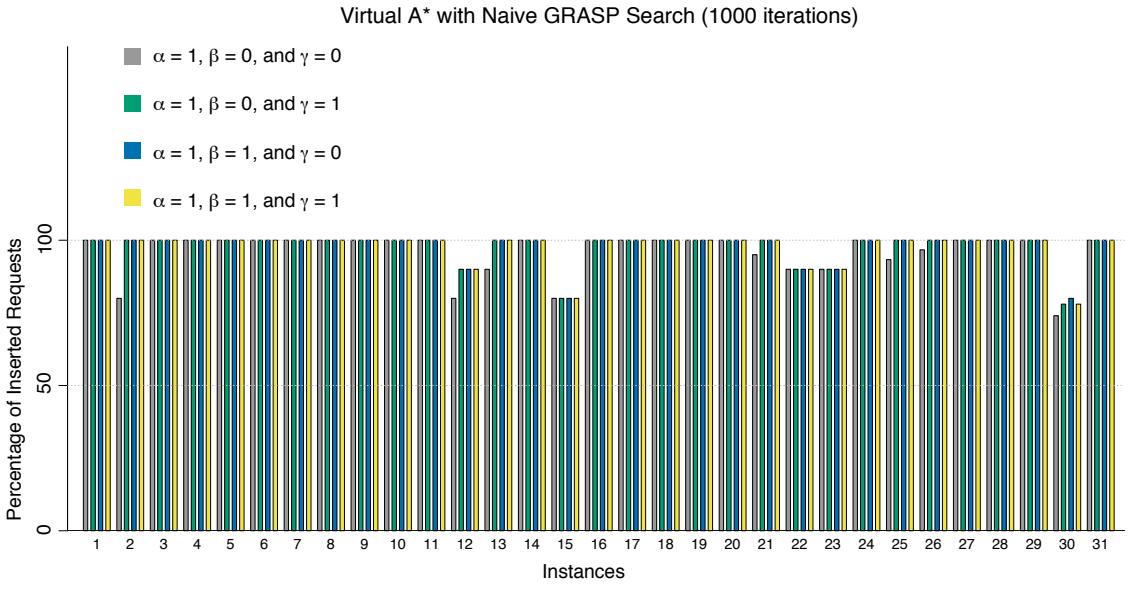


Figure 4.22: Percentages of inserted requests obtained by performing 1000 iterations of Algorithm 10 over each instance in Table B.4.

First, we observe that for seven of the 31 instances considered, the assignation of cost parameters $\alpha = 1$, $\beta = 0$, $\gamma = 0$, achieved a lower number of insertions than the other assignations of cost parameters that were examined. This observation agrees with our intuition that by focusing only on the current request, we may perform insertions with a large impact on the time/distance of the tours, and this in turn may reduce our chances of inserting more requests. In contrast, it does not seem to be a significant difference between the assignations $\alpha = 1$, $\beta = 0$, $\gamma = 1$; $\alpha = 1$, $\beta = 1$, $\gamma = 0$; and $\alpha = 1$, $\beta = 1$, $\gamma = 1$.

Of course, it can be argued that 1000 iterations could be too much computing effort for the considered instances. However, a similar behavior is observed when we consider only 100 iterations of Algorithm 10 (we omit the corresponding plots).

Average running time for a single request insertion. Figure 4.23 shows the average running times in milliseconds for performing the insertion of a single request.

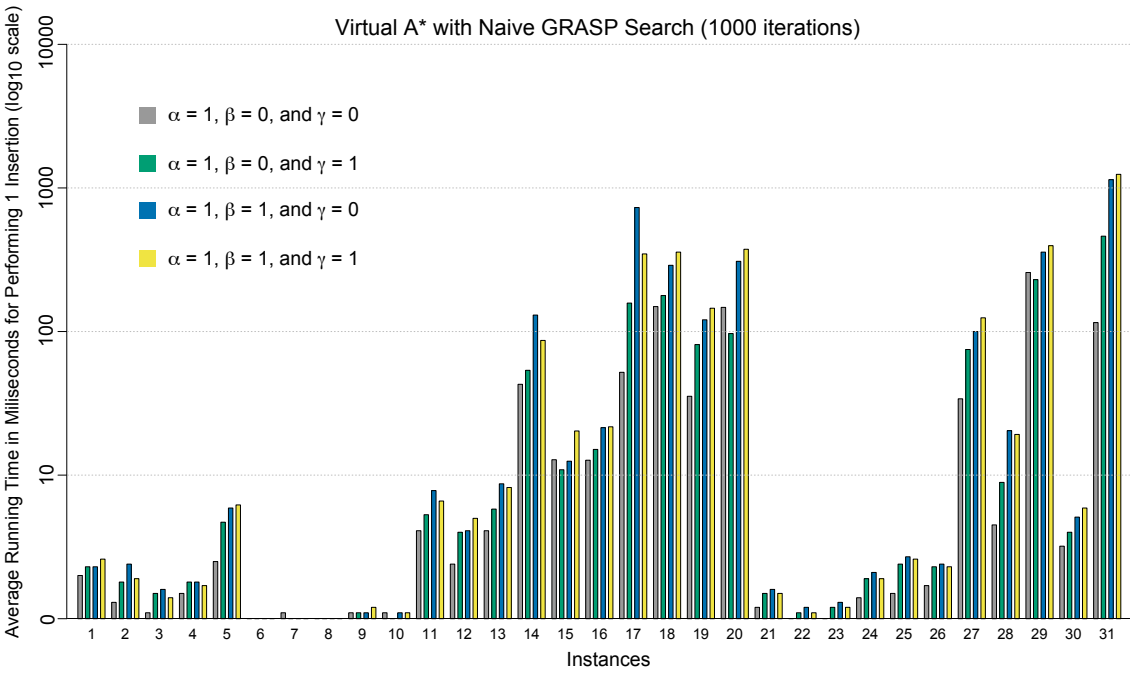


Figure 4.23: The average running time in milliseconds for performing the insertion of one request while considering the different combinations of cost parameters with $\alpha = 1$, $\beta \in \{0, 1\}$, and $\gamma \in \{0, 1\}$. These values were calculated by averaging the running times per iteration of executing Algorithm 10 with 1000 iterations over each instance in Table B.4. Note the logarithmic scale ($\log_{10}$).

For computing those values, we have proceeded in the following way.

1. First, we have computed the average running time of one iteration of the naive GRASP search algorithm; that is, we simply divide by 1000 the total running time for performing 1000 iterations of the random algorithm.

2. Next, we have divided the average running time of one iteration of the naive GRASP search algorithm by the number of requests.

Figure 4.23 suggests that inserting requests is more difficult on average on those instances which start from collections of prescheduled tours with more than one arc.

We observe also a tendency indicating that insertions are easier on average when we consider the cost parameters $\alpha = 1$, $\beta = 0$, $\gamma = 0$. In the opposite way, the insertions involving the assignations of cost parameters $\alpha = 1$, $\beta = 1$, $\gamma = 0$; and $\alpha = 1$, $\beta = 1$, $\gamma = 1$ seem to be more difficult on average.

Based on previous results, we proceed to compare the algorithms only for the assignation of cost parameters $\alpha = 1$, $\beta = 1$, $\gamma = 1$.

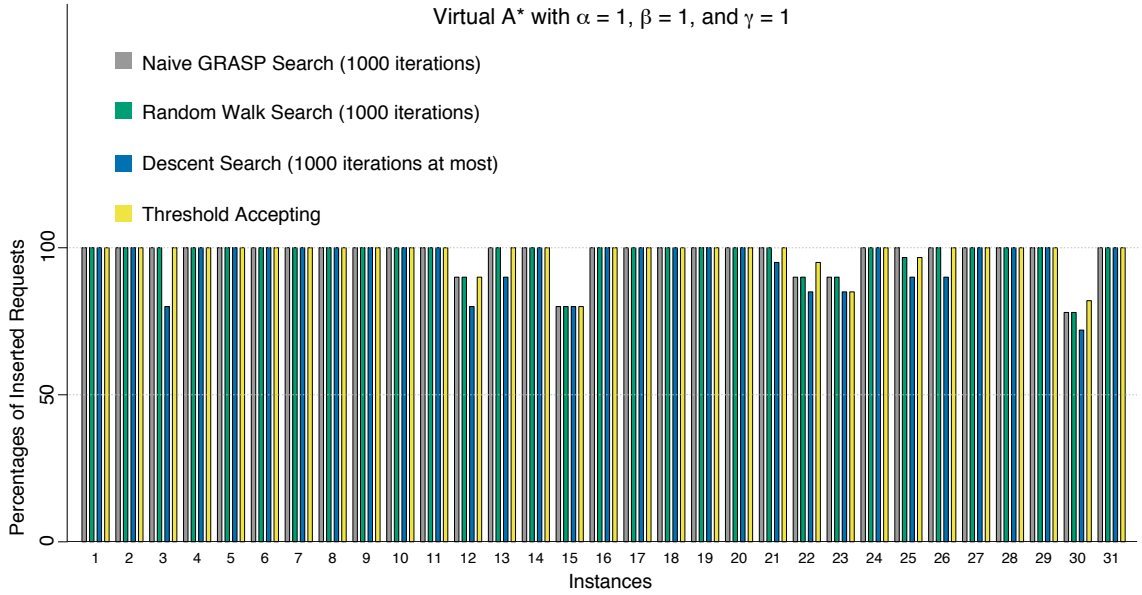


Figure 4.24: Maximum percentages of inserted requests obtained from the execution of the naive GRASP search with 1000 iterations, the random walk search with 1000 iterations, the descent search with at most 1000 iterations, and the threshold accepting algorithm with parameters `batch_size` = 15, `temperature` = $4 \cdot n$, `cooling_factor` = 0.85, and `freezing_point` = n (where n is the size of the integral grid that was considered for constructing the instance). Those algorithms were executed over each instance in Table B.4 while considering the cost parameters $\alpha = 1$, $\beta = 1$, and $\gamma = 1$.

Figures 4.24-4.25 indicate that the descent search algorithm achieves solutions with a smaller number of insertions and with a worse cost on average. However, if we examine Figure 4.26, we can verify that the descent search algorithm has required lower running times than the other examined algorithms.

A careful scrutiny of results in Table B.8 shows us that there were no difference between the solutions found by the descent search algorithm with 10, 100, and 1000 iterations. It results that the descent search algorithm has found a local optima during the first ten iterations and therefore, there was no difference when we allowed more iterations.

In contrast, for most of the instances, there was no significant difference between the percentages of inserted requests in the solutions found by the naive GRASP search, the random walk search, and the threshold accepting algorithms; they have respectively achieved 375, 374, and 376 insertions in total; and an average 1-insertion cost of 130.42, 130.66, and 129.35, respectively.

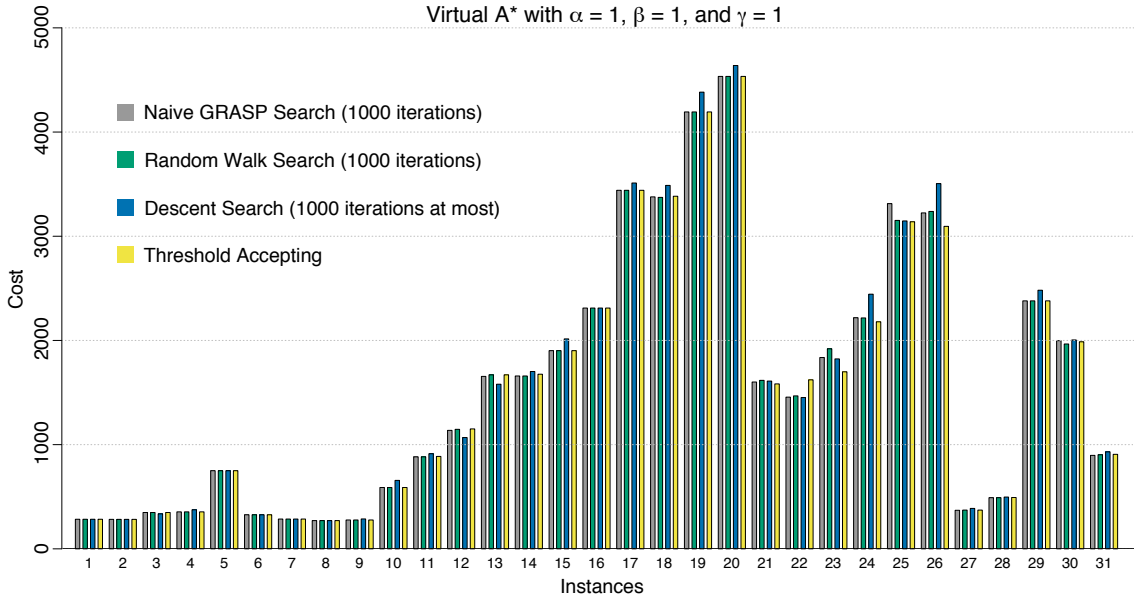


Figure 4.25: Costs of the best feasible solutions found by executing the naive GRASP search algorithm with 1000 iterations, the random walk search with 1000 iterations, the descent search with at most 1000 iterations, and the threshold accepting algorithm with parameters `batch_size` = 15, `temperature` = $4 \cdot n$, `cooling_factor` = 0.85, and `freezing_point` = n (where n is the size of the integral grid that was considered for constructing the instance). Those algorithms were executed over each instance in Table B.4 while considering the cost parameters $\alpha = 1$, $\beta = 1$, and $\gamma = 1$.

Those data become more interesting when we observe in Figure 4.26 that for most of the instances, the threshold accepting algorithm has required significant lower running times (notice the logarithmic scale) than the naive GRASP search and the random walk search.

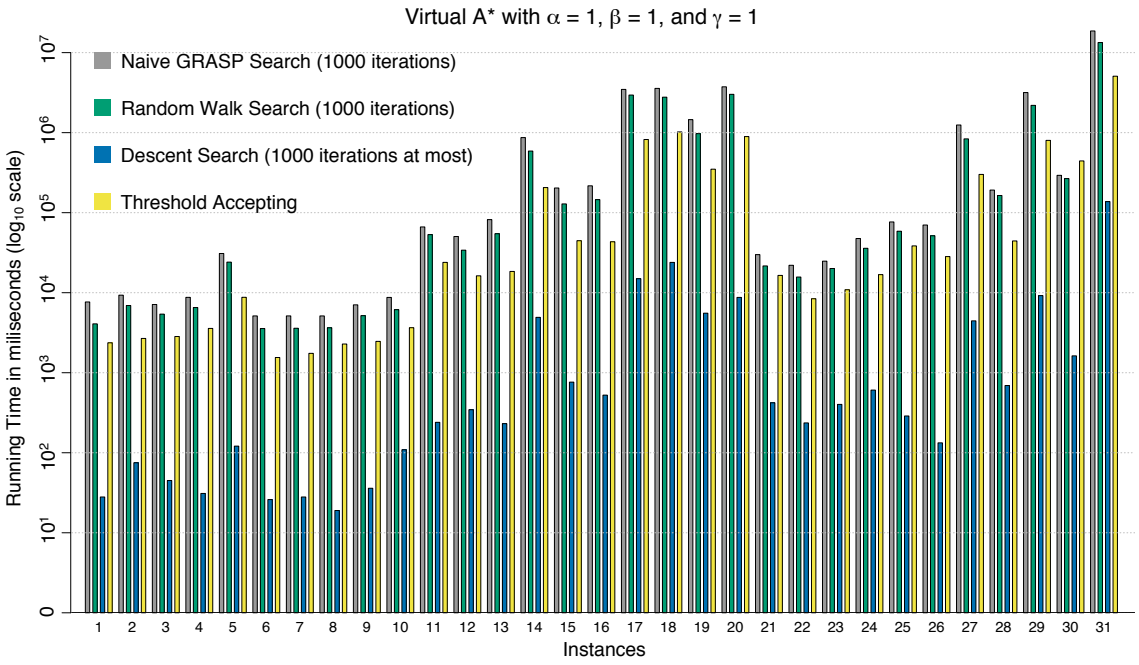


Figure 4.26: Running times in milliseconds used for the execution of the naive GRASP search algorithm with 1000 iterations, the random walk search with 1000 iterations, the descent search with at most 1000 iterations, and the threshold accepting algorithm with parameters `batch_size` = 15, `temperature` = $4 \cdot n$, `cooling_factor` = 0.85, and `freezing_point` = n (where n is the size of the integral grid that was considered for constructing the instance). Those algorithms were executed over each instance in Table B.4 while considering the cost parameters $\alpha = 1$, $\beta = 1$, and $\gamma = 1$. Note the logarithmic scale ($\log_{10}$).

Number of transfers in best found solutions. For instances 1-15 and 27-29 we have performed an exhaustive search for finding a permutation that yields a best cost solution. For the remaining instances, we have chosen the best solution found by the four search methods previously described (i.e., naive GRASP search, random walk search, descent search, and threshold accepting). Figure 4.27 shows the number of transfers in the best found solutions.

We note that it is difficult to figure out a priori the number of new transfers in the solutions of a given instance. However, we may note the following global results:

- the combination of cost parameters $\alpha = 1$, $\beta = 1$, $\gamma = 0$ has created a total of 60 new transfers,
- the combination of cost parameters $\alpha = 1$, $\beta = 1$, $\gamma = 1$ has created a total of 53 new transfers,
- the combination of cost parameters $\alpha = 1$, $\beta = 0$, $\gamma = 1$ has created a total of 45 new transfers,

- the combination of cost parameters $\alpha = 1$, $\beta = 0$, $\gamma = 0$ has created a total of 19 new transfers.

Those results suggest that the number of transfers is slightly bigger when we consider the combination of cost parameters $\alpha = 1$, $\beta = 1$, $\gamma = 0$. In contrast, the combination $\alpha = 1$, $\beta = 0$, $\gamma = 0$ usually yields solutions with a lower number of transfers.

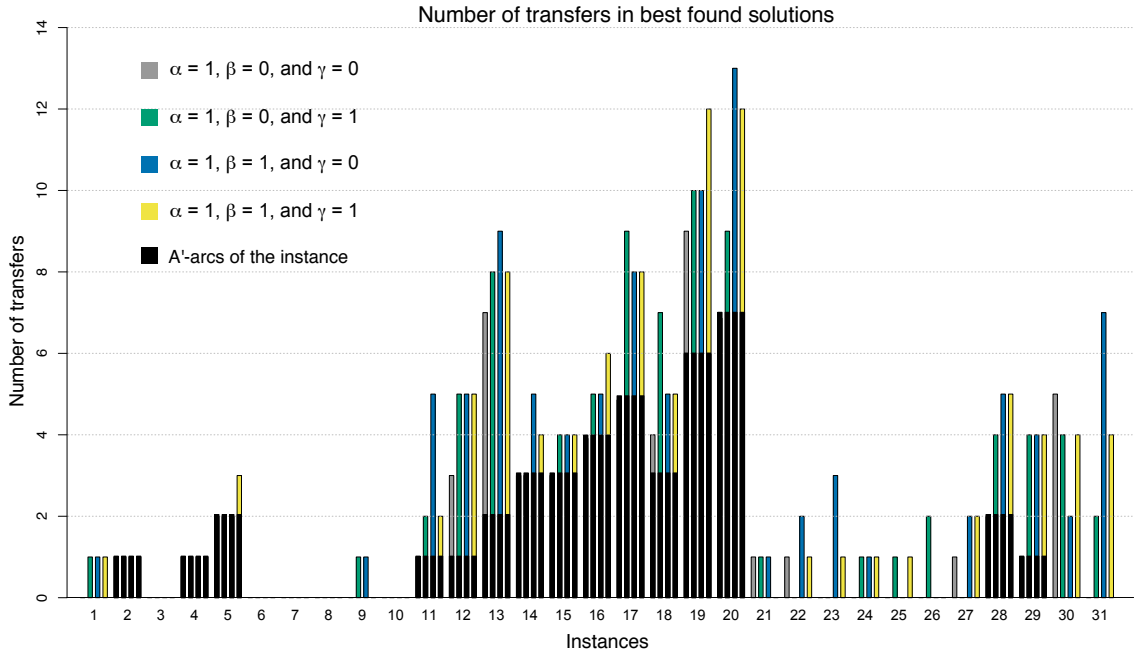


Figure 4.27: Number of transfers in best found solutions. For instances 1-15 and 27-29 we have performed an exhaustive search. For the remaining instances, we have chosen the best solution found by the four search methods described in Section 4.4.2 (i.e., naive GRASP search, random walk search, descent search, and threshold accepting).

4.5 Conclusions

We have introduced the Virtual Path Problem and we described the Virtual A* algorithm for solving it in an exact way. We also introduced the 1-Request Insertion PDPT problem and we showed that it can be seen as a particular case of the Virtual Path Problem. Then we proposed the Virtual A* algorithm for solving the 1-Request Insertion PDPT in an exact way, and we also provided a heuristic algorithm based on Dijsktra's algorithm.

We could check that, while allowing transfers is often useful, the number of transfers is usually small (see the pie charts in Figure 4.17), and that it is important to a priori filter transfer moves which are allowed.

We have also examined the behavior of the virtual A* algorithm (without any ε-eligibility restriction, and with three transfers at most) when performed as a subroutine for inserting requests in a sequential way, and in combination with some classical metaheuristics. The resulting algorithms were executed for different assignations of cost parameter values, and we have obtained the following observations.

- The virtual A* can be used in practice for small/middle size instances.
- The descent search algorithm may converge to a local optimum within a low number of iterations when we consider the neighborhood $swap(P)$ of a permutation P.
- The assignation of cost parameters $\alpha = 1$, $\beta = 0$, and $\gamma = 0$ usually limits the number of requests that we can insert in a PDPT instance. In contrast, we do not observe much difference between the assignations $\alpha = 1$, $\beta = 0$, and $\gamma = 1$; $\alpha = 1$, $\beta = 1$, and $\gamma = 0$; and $\alpha = 1$, $\beta = 1$, and $\gamma = 1$.
- By tunning correctly the input parameters of Algorithm 11 (i.e., threshold accepting) we can obtain good results with a moderate computational effort.

We close this section by noting that the computational effort necessary for performing one insertion usually exhibits a high variance. In fact, during the experiments it was possible to observe the appearance of some difficult insertions that slowed considerably the speed of the search algorithms. This impact may potentially affect the performance of local search algorithms when a set of difficult/infeasible insertions is tried several times during the examination of a sequence of neighbor solutions.

For this reason, it could be interesting to analyze the behavior of hybrid algorithms with a tabu search (respectively, a squeaky wheel [134]) component, that considers the difficulty of the insertion of a particular request to include it in some tabu list (respectively, priority list), and attenuate the impact in running time of performing difficult/infeasible insertions.

Part IV

Conclusions

We have analyzed two static Pickup-and-Delivery Problems with transfers and time horizon and proposed new methods and algorithms for handling them.

First we have studied the Item Relocation Problem and we have proposed a 2-commodity flow model over a Time-Expanded network. The model is meaningful but results very difficult to solve in the practice. For that reason, we have proposed a Project-and-Lift approach for handling it in a flexible way.

We have projected the 2-commodity flow model over the original transit network to obtain a simpler 2-commodity flow model that we have called the Projected Item Relocation Model (PIRP). To recover a part of the temporal constraints, we have introduced the Extended-Subtour constraints which link the time horizon and the number of vehicles circulating through any subset of non-depot vertices. We have shown how those constraints can be separated in polynomial time, and we have observed the efficiency of the resulting branch-and-cut algorithm and the quality of the computed solutions. We have also introduced the Feasible-Path constraints, and we have shown how to handle with them through column generation.

We have introduced two Lift problems: the Strong Lift Problem and the Partial Lift Problem, which can be distinguished by its degree of compatibility with a PIRP solution. We have proposed a MILP formulation that solves the Strong Lift Problem in an exact way. However, we have observed that most of the time, the model constructed from optimal PIRP solutions is infeasible. On the other hand, we have proposed the Weak/Cover for handling the Partial Lift Problem in a flexible way. We introduced the concept of weak-lift-consistency and we have conjectured that the Weak/Cover models can yield optimal solutions when we start from PIRP solutions which are weak-lift-consistent.

As future work it remains to identify other problems where the Extended-Subtour constraints or the lower bound provided by the cost of an optimal PIRP solution can be applied. Also, it would be interesting to find new kinds of constraints to strengthen the PIRP model and increase the probability of obtaining solutions yielding feasible Strong Lift Problems. Furthermore, the proposed Project-and-Lift approach that we have proposed for handling the Lift problems may be used for other problems involving multicommodity flows over Time-Expanded networks.

We also have introduced the Virtual Path Problem which consists in searching for an optimal path within a collection of paths that satisfy and impact a given constraint system. We have proposed the Virtual A* algorithm, which is an A*-like algorithm for solving the Virtual Path Problem in an exact way. Then we studied the 1-Request Insertion PDPT which arises as a subproblem of the Pickup-and-Delivery with paired pickups and deliveries, transfers, and time horizon (PDPT).

We have shown this problem can be seen as a particular case of the Virtual Path Problem and we proposed the virtual A* for solving the 1-Request Insertion PDPT in an exact way. Because the complexity of the virtual A* algorithm is exponential, we also have proposed some heuristics with a polynomial time complexity.

We have tested the virtual A* algorithm in an intensive way over a set of 300 random instances, and we have analyzed its behavior. We have confirmed that most of the time the number of transfers in an optimal solution is small, and that the running times exhibit a high variance. We have also confirmed how the running times are reduced when we limit the number of transfers to a small constant. Also we have observed the impact in solution quality when we filter the transfer-arcs according to a weight threshold parameter.

We showed how the virtual A* algorithm can be combined with classical search metaheuristics for inserting multiple requests into a Pickup-and-Delivery Problem with transfers schedule. We have implemented those algorithms and analyzed their behavior. As a result, we dispose of a solver which is able to handle small/middle size PDPT instances.

As a future lines of research it remains to identify other problems where the Virtual A* algorithm may be applied. It also remains to analyze the case of the strong synchronization constraints that occur when vehicles must meet within a given time window in order to perform a transfer. Also, it would be interesting to analyze the behavior of the proposed search algorithms when they are provided with a component (e.g., tabu list) to penalize the reinsertion of difficult/infeasible requests.

Finally, in this work we have restricted ourselves to generic problems. However, the models and algorithms that we have described in this work may be applied in practical contexts. Therefore, it also remains to identify the type of applications where the proposed models and algorithms can be applied.

Appendix

Appendix A

Basic Theory and Notation

The aim of this work is to introduce some transportation problems arising from practical situations and to describe algorithmic schemes for dealing with them. In order to evaluate problem's difficulty and algorithms' performance, it is necessary to formulate them by using mathematical language.

Although mathematical language is widely used in academic and scientific contexts, there is not a completely standard way of writing mathematics. So, to avoid possible ambiguities, it is necessary to give some insight about the chosen terminology and notation. This chapter introduces some definitions, establishes the notation, and surveys briefly the necessary background knowledge for reading the rest of this document.

A.1 Sets

We use the conventional notation and definitions for sets. This is, we can indicate a set by using a letter, say A, or by indicating a list of its elements between curly braces, for example $\{a, b, c\}$. Given a **set** A, we write $a \in A$ to indicate that an **element** a belongs A, and $a \notin A$ to denote that a is not an element of the set A. We say that A is a **subset** of a set B if all the elements of a A are contained in B, and we denote this by $A \subseteq B$. We say that A is a **proper subset** of a set B (denoted by $A \subset B$) if A is different from B but $A \subseteq B$. As usual, the the set without elements is called the **empty set** and is denoted by $\varnothing$. A set $\{e\}$ containing a single element e is called a **singleton**. We will usually abbreviate such a set by e instead of $\{e\}$.

The **intersection of sets** A and B is the set $A \cap B \coloneqq \{a : a \in A \text{ and } a \in B\}$, and the **union of sets** A and B is the set $A \cup B \coloneqq \{a : a \in A, \text{ or } a \in B, \text{ or both}\}$. If $A \cap B = \varnothing$, sometimes we could write $A \sqcup B$ instead of $A \cup B$. If A and B are

sets, then their **difference** $A \setminus B$ is the set $\{a \in A : a \notin B\}$ and their **symmetric difference**, denoted $A \triangle B$, is the set $(A \setminus B) \sqcup (B \setminus A)$. Most of the sets in this work are finite. Given a finite set A, the **cardinality** of A is simply the number of elements of A and it will be denoted by $|A|$.

The collection of subsets of A is called the **power set** of A and it will be denoted by 2^A. In general, a set of sets will be called a **collection**, also we use the term **member** to indicate an element of a collection of sets. Collections of sets are usually denoted by calligraphic letters. A **cover** (or **covering**) of a set A is a family of subsets of A whose union is all of A. Given a set A, a **finite partition** of A consists of a finite collection $\mathcal{A} = \{A_1, \ldots, A_k\}$ of subsets of A, such that, for every $i, j \in \{1, \ldots, k\}$, with $i \neq j$, we have that $A_i \cap A_j = \varnothing$ and $A_1 \sqcup \cdots \sqcup A_k = A$. The members of a partition are called **parts**. A partition $\mathcal{A}'$ of a set A is a **refinement** of a partition $\mathcal{A}$ of A (and we say that $\mathcal{A}'$ is **finer** than $\mathcal{A}$ and that $\mathcal{A}$ is **coarser** than $\mathcal{A}'$) if every member of $\mathcal{A}'$ is a subset of some member of $\mathcal{A}$.

Given a collection $\mathcal{A}$ of subsets of a set A, frequently we shall be interested in the **maximal** members of $\mathcal{A}$, these are the members of $\mathcal{A}$ who are not properly contained in any other member of $\mathcal{A}$; similarly, the **minimal** members of a collection $\mathcal{A}$ will be the members of $\mathcal{A}$ that are not properly containing any other member of $\mathcal{A}$.

The numerical sets of **integers**, **rational** and **real numbers** will be denoted by $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$, respectively. We add the subscript $_+$ to the right of these symbols to indicate the nonnegative elements of the corresponding set, e.g., $\mathbb{Z}_+$ is the set of **non-negative integers**.

Sometimes, we could be interested in considering a list with repetitions of elements taken from some sets and following some ordering. For this, we use the **n-tuple** denoted by $(x_1, x_2, \ldots, x_n)$ and consisting of the elements $x_1, x_2, \ldots, x_n$ in this particular order. If S is a set, a **multiset** chosen from S is a function m from S to $\mathbb{Z}_+$. For example, if $S = \{a, b, c, d, e\}$ and $\big(m(a), m(b), m(c), m(d), m(e)\big) = (2, 3, 0, 1, 0)$, then we denote this multiset by $\{a, a, b, b, b, d\}$.

Given two sets X and Y, we denote by $X \times Y$ to the **cartesian product** of X and Y which is the set $\{(x, y) : x \in X, y \in Y\}$. The cartesian product can be defined inductively to consider the product of n sets $X_1, X_2, \ldots, X_n$; and in case that $X_1 = X_2 = \ldots = X_n = X$, we abbreviate $X_1 \times X_2 \times \cdots \times X_n$ as X^n.

A **relation** R on $X \times Y$ is simply a subset of $X \times Y$. In case that $X = Y$, we say that R is a relation on X. If $(x, y) \in R$ we say that x and y are related by R, and we denote this by xRy.

A **function** f from a set A to a set B (denoted by $f : A \to B$), is a relation on $A \times B$ such that for any $a \in A$, f contains exactly one pair with first entry a; such a

pair (a, b) is usually denoted by $f(a) = b$ and we call b the **image** of a under f, we also say that f maps a to b.

A function $f : A \to B$ is called an **injective** function if for every $a, b \in A$, we have that $a \neq b$ implies $f(a) \neq f(b)$. Function f is called a **surjective** if for every $b \in B$ there is an element $a \in A$ such that $f(a) = b$. Finally, we say that f is a **bijective function** if f is injective and surjective.

Given a subset A of a set X with n elements, the **characteristic function** of the set A is the function $\chi_A : X \to \{0, 1\}$ defined by

$$\chi_A(x) := \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

The **floor function** is the function that takes as input a real number x, and gives as output the greatest integer less than or equal to x, denoted $\lfloor x \rfloor$. Similarly, the **ceiling function** maps x to the least integer greater than or equal to x, denoted $\lceil x \rceil$.

A.2 Matrices and Vector Spaces

In this section we give the notation and definitions related to matrices and vectors spaces. For an introductory treatment of vector spaces and other algebraic structures we refer the reader to the books of Grossman [111] or Insel et al. [128].

In this work, a $n \times m$ **matrix** will be simply a rectangular table of numbers with n rows and m columns. Most of the time, we will be dealing with real matrices, so in the rest of this subsection we consider matrices whose entries are real numbers. If $\mathbf{A}$ is a matrix, we denote the entry in its ith row and jth column by a_{ij}, e.g., the following picture depicts a matrix $\mathbf{A}$ with three rows and four columns.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

Alternatively, sometimes we can write (a_{ij}) to indicate the matrix whose entries are the a_{ij}. If $\mathbf{A}$ is a $m \times n$ matrix, we denote by $\mathbf{A}^T$ to the **transposed matrix** $\mathbf{A}$, which is the $n \times m$ matrix having the element a_{ji} in the ith row and jth column.

We will denote by $\mathbf{I}_{r\times s}$ the **identity matrix** of $r\times s$, this is the $r\times s$ matrix (a_{ij}) defined by

$$a_{ij} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j, \end{cases}$$

We denote by $\mathbf{J}_{r\times s}$ the $r\times s$ **matrix of all ones** (this is, the matrix with all its entries equal to one), and by $\mathbf{0}_{r\times s}$ the **zero matrix** of $r\times s$ (this is, the matrix with all its entries equal to zero). In case $r = s$, we can denote these matrices by $\mathbf{I}_r$, $\mathbf{J}_r$, and $\mathbf{0}_r$, respectively.

We recall that it is possible to perform some operations with matrices, for example, if $\mathbf{A}$ is a matrix and $\lambda \in \mathbb{R}$, the multiplication of λ and $\mathbf{A}$ is the matrix $\lambda\mathbf{A}$ obtained from $\mathbf{A}$ by multiplying each entry of $\mathbf{A}$ by λ. In particular, $-1\mathbf{A}$ is the matrix obtained by changing the sign of every entry in $\mathbf{A}$, we will denote this particular matrix by $-\mathbf{A}$. Let $\mathbf{A}$, $\mathbf{B}$ be two $m\times n$ matrices. The sum of $\mathbf{A}$ and $\mathbf{B}$, denoted by $\mathbf{A}+\mathbf{B}$, is a matrix $\mathbf{C} = (c_{ij})$, such that $c_{ij} := a_{ij} + b_{ij}$. On the other hand, if $\mathbf{A}$ is a $m\times n$ matrix and $\mathbf{B}$ is a $n\times r$ matrix, the product of $\mathbf{A}$ and $\mathbf{B}$ is denoted by $\mathbf{AB}$ and corresponds to the matrix $\mathbf{C} = (c_{ij})$ defined by $c_{ij} := a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$.

Given a set A, a **binary operation** $*$ on A is simply a function $*$ from $A\times A$ to A. Given $a, b \in A$ and a binary operation $*$ on A, we denote by $a * b$ to the image of (a, b) under $*$. A binary operation $*$ on A is **associative** if for every $a, b, c \in A$, we have the equation $(a * b) * c = a * (b * c)$; similarly, we say that operation $*$ is **commutative** if for verey $a, b \in A$ we have the equation $a * b = b * a$.

A **group** $(G, *)$ consists of a set G together with an associative binary operation $*$ such that:

- there exists an element $e \in G$, called the **unit element** with respect to $*$, such that $e * x = x * e = x$ for every $x \in G$;
- for every $a \in G$, there exists an element $b \in G$, called the **inverse of** a, such that, $a * b = b * a = e$, where e is the unit element.

A group $(G, *)$ is said **abelian** if $a * b = b * a$, for every $a, b \in G$, i.e., if $*$ is a commutative operation.

A **field** is a set $\mathbb{F}$ with two binary operations + and $\cdot$, such that, the set $\mathbb{F}$ with the operation + is an abelian group with unit element $0_{\mathbb{F}}$, and the set $\mathbb{F}\setminus\{0_{\mathbb{F}}\}$ with the operation $\cdot$ is also an abelian group with unit element denoted by $1_{\mathbb{F}}$.

A **vector space** over a field $\mathbb{F}$ is a commutative group V with a binary operation + with unit element $\mathbf{0}_V$, and an operation assigning to every pair $(\alpha,\ \mathbf{v}) \in \mathbb{F}\times V$ an element of V denoted simply as $\alpha\mathbf{v}$. Furthermore, the following properties hold for

any $\mathbf{x}$, $\mathbf{y} \in V$, and any $\alpha, \beta \in \mathbb{F}$: $\alpha(\mathbf{x}+\mathbf{y}) = \alpha\mathbf{x} + \alpha\mathbf{y}$, $(\alpha+\beta)\mathbf{x} = \alpha\mathbf{y} + \beta\mathbf{y}$, $\alpha(\beta\mathbf{x}) = (\alpha\cdot\beta)\mathbf{x}$, and $1_{\mathbb{F}}\mathbf{x} = \mathbf{x}$. The elements of a vector space are called **vectors** and we will use bold letters (such as $\mathbf{x}$, $\boldsymbol{\ell}$, $\boldsymbol{\lambda}$, or $\mathbf{Z}$) to denote vectors. A subset A of a vector space V over a field $\mathbb{F}$ is called **linearly independent** if for every $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \subseteq A$, and for every $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{F}$, we have $\alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2 + \cdots + \alpha_n\mathbf{x}_n = \mathbf{0}_V$ if and only if $\alpha_1 = \alpha_2 = \cdots = \alpha_n = 0_{\mathbb{F}}$. The maximum possible cardinality of an independet set in a vector space V is called the **dimension** of V.

If $\mathbb{F}$ is a field and r is an non negative integer, then $V(r,\mathbb{F})$ will denote the **vector space** of dimension r over $\mathbb{F}$. Note that this vector space can be thought as the cartesian product $\mathbb{F}^r$, so a vector $\mathbf{x} = (x_1, x_2, \ldots, x_r) \in V(r,\mathbb{F})$ has the form of a r-tuple, where every entry is an element of $\mathbb{F}$. However, vectors in a space of dimension r are usually regarded as a $r \times 1$ matrices (e.g., as columns of a $r \times r$ matrix), and in this way, we are able to multiply vectors with vectors or matrices, as long as they have compatible sizes to perform these operations as the corresponding matrix operations. In consonance with these comments, the transpose of a vector $\mathbf{v} \in V(r,\mathbb{F})$ will be denoted by $\mathbf{v}^T$, and can be regarded as a $1 \times r$ matrix (i.e., as a row of a $r \times r$ matrix).

We denote the ith coordinate of a vector $\mathbf{x}$ by x_i. The **standard basis** of a vector space $V(r,\mathbb{F})$ consists of the r vectors $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_r$ where $\mathbf{e}_i$ is the r-tuple having $1_{\mathbb{F}}$ in the ith coordinate and $0_{\mathbb{F}}$ elsewhere. A real vector indexed over a set A, with all its entries equal to 1 will be denoted by $\mathbf{1}_A$; similarly, we denote by $\mathbf{0}_A$ to the real vector with all its entries equal to 0.

Let us now turn our attention to metric spaces. The following definitions were taken from the book of Bartle and Sherbert [22].

A **metric** (also called **distance**) on a set A is a function $d : A \times A \to \mathbb{R}$ satisfying the following properties:

(i) $d(x,y) \geq 0$ for all $x, y \in A$ (**positivity**);

(ii) $d(x,y) = 0$ if and only if $x = y$ for all $x, y \in A$ (**definiteness**);

(iii) $d(x,y) = d(y,x)$ for all $x, y \in A$ (**symmetry**);

(iv) $d(x,y) \leq d(x,z) + d(z,y)$ for all $x, y, z \in A$ (**triangle inequality**).

A **metric space** (A,d) is a set together with a metric d on A.

Example A.1 - The Euclidean Real Vector Space
The n-dimensional real vector space $V(n,\mathbb{R})$ together with the Euclidean distance d defined for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ as:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

is an example of metric space. □

Example A.2 - The Taxicab Geometry
Another example of metric space who arises from the notion of distance in urban areas is the **taxicab geometry** (cf. Krause [143]). This metric space also consists of the n-dimensional real vector space and a distance d_{taxi}. The *taxicab* distance d_{taxi} between two vectors $\mathbf{p}$, $\mathbf{q}$ of the n-dimensional real vector space with fixed Cartesian coordinate system is defined as

$$d_{taxi}(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{n} |p_i - q_i|.$$

□

The metric d_{taxi} has the interesting property of being closed over the integers and the rational numbers, this is, if the vectors $\mathbf{p}$, $\mathbf{q}$ have integers (respectively rational) coordinates, $d_{taxi}(\mathbf{p}, \mathbf{q})$ is also an integer (respectively, rational) number. Another metric which is also closed over the integers is the **upper rounded Euclidean distance** defined for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ as $\left\lceil \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \right\rceil$.

A.3 Graph Theory

In this Section we give the definitions and results from graph theory that will be used in the rest of this document. For a more detailed treatment of graph theory we refer the reader to the books of Diestel [69] and Bondy and Murty [34].

A **graph** G is an ordered pair $(V(G), E(G))$ consisting of a set $V(G)$ of **vertices**, and a set $E(G)$, disjoint from $V(G)$, of **edges**, together with an **incidence function** ψ_G that associates with each edge of G an unordered pair of (not necessarily different) vertices of G. Usually, for notational simplicity, we write uv to denote the unordered pair $\{u, v\}$. Sometimes, a graph G may have one or more **distinguished vertices**, they are vertices of G with a special notation to distinguish them from other vertices. In this work, we may consider the following ones.

- A **depot** vertex denoted by d_G;
- a **source** vertex denoted by $\hat{s}_G$; and
- a **sink** vertex denoted by $\hat{p}_G$.

If e is an edge and x, y are vertices such that $\psi_G(e) = xy$, then we say e **joins** x **and** y, and the vertices x, y are called the **endpoints** of e. We say the endpoints of an edge are incident with the edge and *vice versa*. Two vertices incident with a common edge are **adjacent** as are two edges incident with a common vertex. Two distinct adjacent vertices are called **neighbors**. The set of neighbors of a vertex x in a graph $G = (V(G), E(G))$ is denoted by $N_G(x)$ and the set of edges in $E(G)$ that are incident with x is denoted by $\partial_G(x)$. The **degree** of a vertex x is the number of edges incident with x, counting each loop as two edges. An **isolated vertex** is a vertex of degree zero. The minimum degree of a vertex in a graph G is denoted by $\delta(G)$, and analogously, the maximum degree is denoted by $\Delta(G)$.

An edge with identical endpoints is called a **loop** and if the endpoints are different the edge is called a **link**. Two or more edges that are not loops and have the same pair of endpoints are said to be **parallel edges**. We say a graph is **simple** if it has no loops and no parallel edges.

When no confusion arises, we usually omit the explicit mention of the graph G in the notation and so, for example, we can write V, E, ψ, $N(x)$ instead of $V(G)$, $E(G)$, ψ_G, $N_G(x)$, respectively.

Example A.3 - An Example of Graph
Figure A.1 is a pictorial representation of a graph G. The set of vertices of this graph is $V(G) = \{x_1, x_2, ..., x_7\}$ and the set of edges is $E(G) = \{e_1, e_2, ..., e_{12}\}$. Edge e_5 joins vertex x_1 with itself, so it is a loop. Edges like e_1, e_2, and e_3 are parallel edges because they join the same pair of vertices x_1, x_5. Vertex x_7 is an isolated vertex because it is not incident with any of the edges. The endpoints of e_9 are x_4 and x_5. □

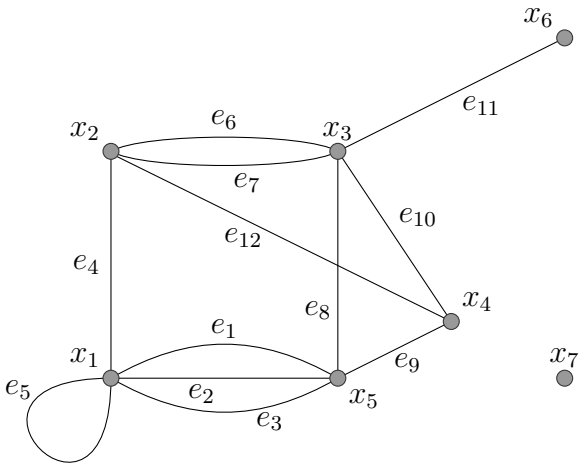


Figure A.1: An example of graph

Remark: Recall that a particular drawing of a graph is not the graph itself, and a graph can be depicted in several ways, as long as we obey the joins dictated by its incidence function. However, to avoid confusion in the drawing of a graph, every edge only have to touch those vertices which are its endpoints.

Sometimes it is useful to specify a graph by its **mod-2 vertex-edge incidence matrix**. For the graph in Figure A.1, this matrix is the following:

$$
\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{array}
\begin{array}{c}
\begin{array}{cccccccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 & e_{10} & e_{11} & e_{12} \end{array} \\
\left(\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \right)
\end{array}
$$

The rows of such a matrix $A = (a_{ij})$ are indexed by the vertices of the graph, and the columns are indexed by the edges of the graph. The entry a_{ij} is the number of times (mod 2) that the jth edge, e_j, is incident with the ith vertex, x_i. Hence a_{ij} is 0 or 1 unless e_j is a loop incident with x_i, in which case, we have $a_{ij} = 0$.

A graph H is a **subgraph** (respectively, **supergraph**) of a graph G if $V(H)$ and $E(H)$ are subsets (respectively, supersets) of $V(G)$ and $E(G)$, respectively. If V' is a subset of $V(G)$, then $G[V']$ will denote the subgraph of G whose vertex set is V' and whose edge set consists of the edges of G having both ends in V'. We say $G[V']$ is the subgraph of G **vertex-induced** by V'. Likewise, if E' is a subset of $E(G)$, then $G[E']$ is the subgraph of G **edge-induced** by E', and it has E' as its set of edges and its set of vertices consists of the ends of the edges of G in E'. Adding a set S of edges to a graph G yields a spanning supergraph of G that is denoted by $G + S$. Similarly, the addition of a set X of vertices to a graph G results in a supergraph of G denoted by $G + X$.

If G_1 and G_2 are graphs, their **union** $G_1 \cup G_2$ is the graph whose set of vertices is $V(G_1) \cup V(G_2)$ and whose set of edges is $E(G_1) \cup E(G_2)$. If $V(G_1)$ and $V(G_2)$ are disjoint then G_1 and G_2 are called **disjoint** and if $E(G_1) \cap E(G_2) = \varnothing$, then G_1 and G_2 are **edge-disjoint**.

Two graphs G and H are **isomorphic**, written $G \cong H$, if there are bijective functions $\theta : V(G) \to V(H)$, and $\phi : E(G) \to E(H)$ such that, a vertex v of G is incident with and edge e of G if and only if $\theta(v)$ is incident with $\phi(e)$ in the graph H.

The graphs depicted in Figure A.2 belong to two important types of graphs: the complete graphs and the bipartite graphs. In general, if n a positive integer, there is, up to isomorphism, a unique graph on n vertices in which each pair of distinct vertices is joined by a single edge. This graph K_n, is called the **complete graph** on n vertices. A **bipartite graph** $G[X, Y]$ is a graph whose set of vertices can be partitioned in two subsets X and Y (called **parts**), such that, every edge of G has one endpoint in X and the other one in Y. If the bipartite graph is simple and every vertex in X is joined to every vertex of Y, the graph is called a **complete bipartite graph** and is denoted by $K_{|X|\,|Y|}$.

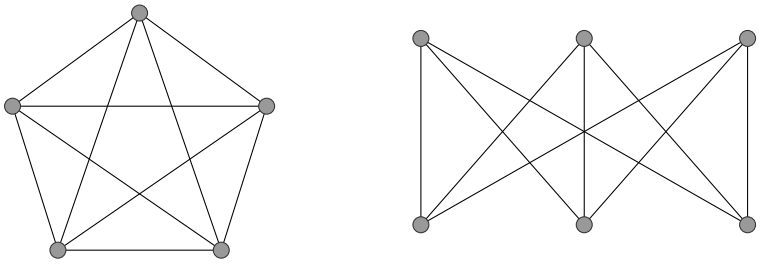


Figure A.2: The graphs K_5 and $K_{3,3}$

An **edge progression** W (from x_1 to x_{k+1}) in a graph G is a finite sequence (x_1, e_1, x_2, …, x_k, e_k, x_{k+1}) such that $k \geq 0$, and $e_i = (x_i, x_{i+1}) \in E(G)$ for $i = 1, \ldots, k$. If in addition $e_i \neq e_j$ for all $1 \leq i < j \leq k$, W is called a **walk** in G.

Let $W = (x_1 = x, e_1, x_2, \ldots, x_k, e_k, x_{k+1} = y)$ be a walk, then we say that W connects x to y, and that W is an **x-y-walk**. Vertices $x_2, \ldots, x_k$ are called the **internal vertices** of W. The **endpoints** of W are $x_1 = x$ and $x_{k+1} = y$: vertex x is the **initial vertex** of W, and vertex y is the **terminal vertex** of W. A walk with initial vertex x is an **x-walk**. If x_i and x_j are two vertices in the sequence of W, with $i < j$, the subsequence of W from x_i to x_j is denoted by $W_{[x_i,x_j]}$ and is called the **segment** of W from x_i to x_j. The **length of** W is the number of edges that contains. If all the vertices of a walk are distinct, then the edges are also distinct and the walk is called a **path**.

The terminology and notation for paths is inherited from the corresponding terminology and notation for walks, so for example, a path P from x_1 to x_{k+1} is called an x_1-x_{k+1}-path. The vertices x_1 and x_{k+1} are the **endpoints** of P. Vertices x_2, …, x_k are its **internal vertices**. By $P_{[x,y]}$ with $x, y \in V(P)$ we mean the (unique) subgraph of P which is an x-y-path. Also, it is possible to extend the notion of x-y-path to paths connecting subsets X and Y of vertices. An **X-Y-path** is a path with initial vertex in X and terminal vertex in Y, and whose internal vertices do not belong neither X nor Y. Finally, note that when G is a simple

graph, we can abbreviate any arc progression $(x_1, a_1, x_2, \ldots, x_k, a_k, x_{k+1})$ by simply listing its vertex sequence $(x_1, x_2, \ldots, x_k, x_{k+1})$.

A graph is **connected** if every pair of distinct vertices is connected by a path. A graph that is not connected is called **disconnected**. In a graph G, the maximal connected subgraphs (by contention) are called **connected components**. The sets of vertices (respectively edges) of the connected components of G form a partition of $V(G)$ (respectively $E(G)$).

If P is an x-y-path in a graph G and e is an edge joining x with y that is not in P, then the subgraph of G with set of vertices $V(P)$ and set of edges $E(P) \cup e$ is called a **cycle**. The **length** of the cycle is the number of eges that it contains. Cycles with three, four, five, and six edges are called, **triangles**, **quadrilaterals**, **pentagons**, and **hexagons**, respectively,

A graph without cycles is a **forest**, and a connected forest is a **tree**. It is not difficult to prove that a graph is a forest if and only if all of its connected components are trees. In a tree, a vertex of degree exactly one is called a **leaf** and it is a known fact that every tree with two or more vertices must contain a leaf.

A **spanning tree** of a connected graph G is a subgraph T of G, such that, T is a tree and $V(T) = V(G)$. Trees have many interesting properties and characterizations, for example, for every tree T we have that $|E(T)| = |V(T)| - 1$. As a consequence, if T is a spanning tree of a graph G, we have the equality $|E(T)| = |V(G)| - 1$. Another interesting property of trees is that any two vertices are connected by exactly one path. We denote the unique path connecting two vertices x and y of a tree T by $T_{[x,y]}$.

In some applications, we need to associate numbers with vertices or edges in a graph to take into account some additional characteristics from the real world (such as costs or capacities). A **weight function** associated with a graph G is any function $c : E(G) \to \mathbb{R}$. For a nonempty $F \subseteq E(G)$ we write $c(F) := \sum_{e \in F} c(e)$, and we define $c(\varnothing) := 0$. Moreover, $\mathrm{dist}_{(G,c)}(x, y)$ denotes the minimum $c(E(P))$ over all x-y-paths P in G. Given $x \in V(G)$ and $U \subseteq V(G)$, we define $\mathrm{dist}_{(G,c)}(x, U) := \inf_{y \in U} \mathrm{dist}_{(G,c)}(x, y)$, and symmetrically $\mathrm{dist}_{(G,c)}(U, x) := \inf_{y \in U} \mathrm{dist}_{(G,c)}(y, x)$.

Let c be a weight function associated with a graph G. Given $e \in E(G)$, the value $c(e)$ is called the **weight** of e. A graph G together with a weight function c defined on its edges is called a **weighted graph**, and is denoted by (G, c).

A **directed graph**[♣] or **digraph** D is an ordered pair $(V(D), A(D))$ consisting of a set $V = V(D)$ of vertices and a set $A = A(D)$ of arcs, together with an incidence

[♣] In some contexts, like operations research, a directed graph is often called a network.

function ψ_D that associates with each arc of D an ordered pair of (not necessarily distinct) vertices fo D. If a is an arc with $\psi_D(a) = (x, y)$, we can write $a = (x, y)$, we say that a **joins** x **to** y, and we call x, y the **endpoints** of a. Vertex x is the **tail** of a, vertex y is the **head** of a, and we say that $a = (x, y)$ **leaves** x and **enters** y. An example of a digraph is depicted in Figure A.3. Note that we have represented the arcs by arrows. For example, $a_1 = (x_1, x_2)$ has tail x_1 and head x_2.

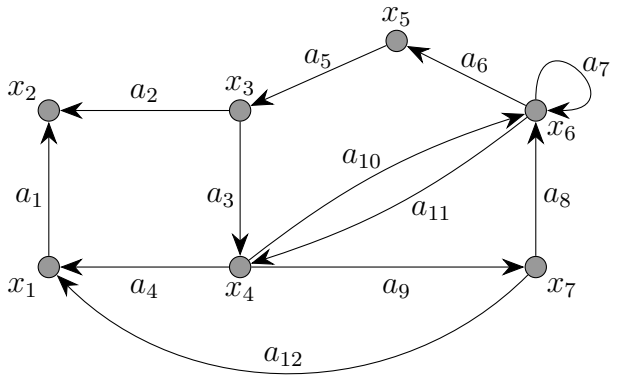


Figure A.3: A example of digraph

Given a vertex $x \in V(D)$, the tails of arcs in $A(D)$ with head x are the **in-neighbours** of x; analogously, the heads of arcs in $A(D)$ with tail x are called the **outneighbours** of x. These sets of vertices are denoted by $N_D^-(x)$ and $N_D^+(x)$, respectively, and their cardinalities are called **indegree** of x and **outdegree** of x, respectively. The maximum indegree and outdegree of a digraph D are denoted by $\Delta^-(D)$ and $\Delta^+(D)$, respectively.

Clearly we can obtain a digraph from a graph G by specifying an order for the endpoints of every edge; the resulting digraph is then called an **orientation** of G. Conversely, if D is a digraph and G is the graph obtained from D by replacing every arc by and edge with the same pair of endpoints, then G is the **underlying graph** of the directed graph D.

In general, the terminology for directed graphs is very similar to the terminology for graphs. For example, an **arc progression** W (from x_1 to x_{k+1}) in a digraph D is a finite sequence $(x_1, a_1, x_2, \ldots, x_k, a_k, x_{k+1})$ such that $k \geq 0$, and $a_i = (x_i, x_{i+1}) \in A(D)$ for $i = 1, \ldots, k$. If in addition $a_i \neq a_j$ for all $1 \leq i < j \leq k$, W is called a **directed walk** in D.

If x and y are the initial and terminal vertices of a directed walk W, we refer to W as a directed x-y-walk. We say that a vertex y is **reachable** from a vertex x if there exists a directed x-y-walk. We say that two vertices are **strongly connected** if x is reachable from y, and symetrically, y is reachable from x. Given a digraph D, the

property of strong connectivity induces a partition of $V(D)$, and the subdigraphs of D induced by each part of the partition are called the **strong components** of D.

If X and Y are sets of vertices (not necessarily disjoint) of a digraph $D = (V, A)$, we denote by $A(X, Y)$ to the set of arcs with tail in X and head in Y. When $Y = V \setminus X$, the set $A(X, Y)$ is denoted by $\partial^+(X)$ and is called the **outcut** of D associated with X. Analogously, the set $A(Y, X)$ is the **incut** of D associated with X and is denoted by $\partial^-(X)$. An **x-y-cut** in a digraph D is an outcut $\partial^+(X)$ such that $x \in X$ and $y \in V \setminus X$, and we say that such a cut **separates** y from x. Summarizing, given $U, V \subseteq V(D)$, we define the following subsets of $A(D)$.

- $\partial^-_D(U) := \{(x, y) \in A(D) : x \notin U, y \in U\}$;
- $\partial^+_D(U) := \{(x, y) \in A(D) : x \in U, y \notin U\}$;
- $\partial_D(U) := \partial^-_D(U) \cup \partial^+_D(U)$;
- $A(U, V) := \{(x, y) \in A(D) : x \in U, y \in V\}$.

If U consists of a single element x (i.e., U is the singleton $\{x\}$), then we write $\partial^-_D(x)$, $\partial^+_D(x)$, $\partial_D(x)$, and $A(x, V)$, instead of $\partial^-_D(\{x\})$, $\partial^+_D(\{x\})$, $\partial_D(\{x\})$, and $A(\{x\}, V)$, respectively. Analogously, if $V = \{y\}$ we write $A(U, y)$ instead of $A(U, \{y\})$.

A **flow network** $N(\hat{s}, \hat{p})$ consists of a digraph $D = (V, A)$ (the digraph associated with $N(\hat{s}, \hat{p})$) with two distinguished vertices, a source $\hat{s}$ and a sink $\hat{p}$, together with a nonnegative function $cap : A \to \mathbb{R}_+$ called the **capacity function** of $N(\hat{s}, \hat{p})$. Given an arc $a \in A$ the weight $cap(a)$ is the **capacity** of a.

Let $N(\hat{s}, \hat{p})$ be a flow network with associated digraph $D = (V, A)$ and capacities $cap : A(D) \to \mathbb{R}_+$. A **flow** is a function $f : A(D) \to \mathbb{R}_+$ with $f(a) \leq cap(a)$ for all $a \in A(D)$. The **excess** of a flow f at $v \in V(D)$ is defined by

$$ex_f(v) = \sum_{a \in \partial^-_D(v)} f(a) - \sum_{a \in \partial^+_D(v)} f(a).$$

We say that f satisfies the **flow conservation rule** at vertex v if $ex_f(v) = 0$. A flow satisfying the flow conservation rule at each vertex is called a **circulation**. An $\hat{s}$-$\hat{p}$-flow is a flow satisfying $ex_f(\hat{s}) \leq 0$ and $ex_f(v) = 0$ for all $v \in V(D) \setminus \{\hat{s}, \hat{p}\}$. We define the **value** $\mathrm{val}(f)$ of an $\hat{s}$-$\hat{p}$-flow f by $\mathrm{val}(f) := -ex_f(\hat{s})$. We say that a flow f in a flow network $N(\hat{s}, \hat{p})$ is **maximum** if there is no flow in $N(\hat{s}, \hat{p})$ with value greater than $\mathrm{val}(f)$.

A **cut** in a flow network $N(\hat{s}, \hat{p})$ is an $\hat{s}$-$\hat{p}$-cut in its underlying digraph. The **capacity** of a cut $K = \partial^+(X)$ is defined as the sum of the capacities of its arcs and we denote this value by $\mathrm{cap}(K)$.

A.4 Algorithms and Complexity

In mathematics, one way of comparing real valued functions of a single variable is examining their behavior either around a particular value or when the variable takes arbitrary large (or low) values. In this latter case we talk about asymptotic analysis of functions.

In asymptotic analysis, we are usually interested in determining, for a given function $f : \mathbb{N} \to \mathbb{N}$ if one of the following cases holds:

1. $f(n)$ converges to a certain value when $n \to \infty$;
2. $f(n)$ does not converge, but remains bounded between certain values when $n \to \infty$;
3. $f(n)$ diverges, this is, for every $n \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $f(\ell) \geq n$ for every $\ell \geq m$.

In the third case, we are also interested in the "rate" of growing of f. This rate is usually determined by comparing $f(n)$ with the corresponding values of other well known functions like $\log(x), x, x^2, x^3, \ldots e^x$. To indicate such comparisons the so called **"big-Oh" notation** is frequently used.

Let f, g be functions from $\mathbb{N}$ to $\mathbb{R}_+$. The notation $f(n) = O(g(n))$ means that there exists constants n_0 and C such that for all $n \geq n_0$, the inequality $|f(n)| \leq C \cdot g(n)$ is satisfied.

Before than electronic computers became common, many mathematicians were inclined to consider that any finite algorithm was good enough for practical purposes and some problems were considered as solved after finding a finite algorithm. One of the first notions of computational complexity was introduced in Edmonds [78]. In this paper entitled "Flowers, paths and trees", Edmonds attracted the attention about the practical importance of the efficiency of an algorithm, an he also gave the first "efficient" algorithm to compute a matching of maximum cardinality in a general graph.

In Jünger et al. [135], William R. Pulleyblank gives a nice account of the historical context in which Edmonds published a series of articles giving the necessary details to achieve the polinomiality of certain well-known algorithms.

The notions of computational complexity were eventually formalized and gave rise to a formal concept of "polynomial time algorithm" an to the $\mathcal{NP}$-completness theory, that was introduced independently by Cook [52] and Levin [148].

To formalize the concept of polynomial time algorithm, it is necessary to introduce a theoretical machine model (this is, a theoretical computer in which we are going to "execute" algorithms). There exists several of such models like Turing machines or random access machines, and they offer different degrees of flexibility to implement algorithms. However, it can be shown that most of these machine models are equivalent, in the sense that, we can execute the same algorithms and obtain the same results in amounts of time that are not significatively different (i.e., if an algorithm takes a polynomial number of steps in one machine model, then it takes also a polynomial number of steps in the other models). For this reason, we omit these details relative to machine models and we refer the reader to the book of Garey et al. [100] (which describes a one-tape Turing machine), the chapter about $\mathcal{NP}$-completness in Korte and Vygen [142] (which also shows the equivalence between one-tape Turing machines and two-tape Turing machines), or the book of Aho and Hopcroft [3] (which describes random access machines and several Turing machines).

In the rest of this section, we follow chapter 8 of Bondy and Murty [34] and chapter 9 of Cook et al. [53].

First of all, consider a finite set Σ. We will call this particular set Σ the **alphabet** and its elements are called **symbols** or **letters**. Any ordered finite sequence of symbols in Σ is called a **word** or a **string**. The size of a word w will be defined as the number of symbols used un w including multiplicities, and this value will be denoted by size(w).

Also, we will denote by Σ^* the set of all the words of symbols taken from Σ.

The following example shows that we can encode mathematical objects like rational numbers or graphs as words.

Example A.4 - Encoding a graph as a word
Consider the alphabets $\Sigma_1 = \{a, b, c, d, e, f, (,), \{, \}, ,\}$ (i.e., letters from a to f, left parenthesis symbol, right parenthesis symbol, left curly brace symbol, right curly brace symbol, and the comma symbol), and $\Sigma_2 = \{0, 1, \#\}$. The graph in the Figure can be encoded in Σ_1, Σ_2, respectively, as the words:

$w_1 = (\{a, b, c, d, e\}, \{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, a\}\})$

$w_2 = 100\#000\#001\#000\#010\#010\#011\#011\#100\#100\#000\#\#$

The interpretation of w_1 is straightforward because the chosen alphabet Σ_1 is part of the common mathematical notation. However, for interpreting w_2 we need to indicate the following enconding conventions that we have used for writing w_2.

- We have used the symbol # as a separator of binary sequences;

- all the binary sequences appearing in w_2 have the same length;
- each vertex is encoded by a unique binary sequence;
- the collection of binary sequences that encode vertices, can be arranged to follow a total ordering with respect to the lexicographic order;
- the first vertex is encoded by a sequence of zeros;
- the number of vertices (minus one) was encoded as the first binary sequence (from left to right) in w_2;
- the rest of binary sequences have to be read in consecutive pairs, and they correspond to the endpoints of the graph edges;
- the end of a word is indicated by two consecutive separator symbols, that is, by ##.

For decoding w_2, we start by reading the string w_2 from left to right until finding the first separator symbol #, that is, we read the subword 100#. According our encoding conventions, the leading binary sequence indicates the number of vertices in the instance minus one. So, in this particular case, the sequence 100 means that the encoded graph has five vertices, namely, 000, 001, 010, 011, and 100. Note that we can identify these binary sequences, respectively, with the vertices a, b, c, d, and e. Next, we continue reading w_2 until finding the separator symbol # in two occasions, that is, we read the sequence 000#001#, which tell us that the first edge has endpoints 000, and 001 (i.e., it is the encoding of the edge $\{a, b\}$). We continue reading all the remaining edges in this way until finding two consecutive separator symbols ##, which indicate the ending of the word w_2. □

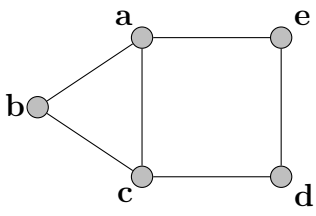


Figure A.4: The graph used in Example A.4.

As the above example shows, there can be several ways of choosing an alphabet or an encoding, and the choosing is quite arbitrary.

Clearly, we won't be interested in encodings containing redundancies, and so for the purpose of this work, we will assume that we are using an efficient encoding (i.e.,

one with length pollinomialy bounded by the minimum possible encoding length), and therefore we are going to consider that the different encodings of an object are equivalent.

Now, we discuss the concept of "problem". In an informal way, we can say that a problem is a question or task. We can classify problems in the following two types.

1. **Decision problems:** problems that can be answered with "yes" or "no", for example: "Is this given graph bipartite?";
2. **Finding problems:** problems in which we are required to find and object with certain properties, for example, "Find a perfect matching of minimal cost in this given graph."

Most of the complexity theory is restricted to decision problems because every problem of finding an object usually has associated a family of decision problems and, from a complexity point of view, the complexity of a finding problem and its associated familly of decision problems is the same.

Since we encode mathematical objects by words, we can also encode mathematical properties, so following Cook et al. [53], a **problem** is any subset Π of Σ^*, and any word w in Π is called an **instance**.[◇] In the particular case of decision problems, an instance whose answer is "yes" is called a **"yes"-instance**. Analogously, an instance whose answer is "no" is a **"no"-instance**.. In the rest of this section we consider only decision problems that are given by the set of all its "yes"-instances.

We need to be careful with above definitions: in the real life, we cannot decide that some given instance w is a "yes"-instance of a problem Π by simply checking if $w \in \Pi$ because the set Π is not usually known in an explicit way, and it may be huge or infinite. For deciding if $w \in \Pi$, we need to apply some ad hoc "algorithm" $\mathcal{A}$ that "solves" Π in the following way: $w \in \Pi$ if and only if after "applying" $\mathcal{A}$ to w, it can be determined mathematically that $\mathcal{A}$ "stops" after a finite number of "steps". The formal definitions behind these ideas will be clarified in the following paragraphs.

A **basic step** (in an algorithm) consists of replacing a subword u by a word u'. This is, if we have a word $w = tuv$, where t and v are also subwords, we replace w by the word $w' = tu'v$.

An **algorithm** is a finite list of basic steps, and it can be described by a set $\{(u_1, u_1'), \ldots, (u_n, u_n')\}$ where $u_1, u_1', \ldots, u_n, u_n'$ are words. The interpretation of such a set is the following: if a word w contains an element of $\{u_1, \ldots, u_n\}$ as a subword,

[◇]We can note, however, that Σ^* can have many words without sense, so in a more formal treatment we would require that we can determine in polynomial time (using a fixed machine model) whether an arbitrary word in Σ^* is an instance of Π or not.

then we choose the smallest index j such that w contains u_j as a subword, and we write $w = tu_jv$ in such a way that subword t is as short as possible. We call tu'_jv the **succesor** of w. We say that an algorithm stops at word w if w does not contain any of $u_1, \ldots, u_n$ as subwords.

For a fixed algorithm $\mathcal{A}$, we say that a finite or infinite sequence of words $w_0, w_1, \ldots,$ is *allowed* by $\mathcal{A}$ if w_{i+1} is the succesor of w_i $(i = 0, 1, \ldots)$ and in case that the sequence is finite, the algorithm *stops* at the last word of the sequence. We say that algorithm $\mathcal{A}$ *accepts* a word w if the allowed sequence starting with w is finite.

We say that algorithm $\mathcal{A}$ *solves* a decision problem $\Pi \subseteq \Sigma^*$ if Π is the set of words in Σ^* accepted by $\mathcal{A}$, and we say that $\mathcal{A}$ *solves* Π *in polynomial time* if there exists a polynomial $p(x)$ such that for any word $w \in \Sigma^*$ if $\mathcal{A}$ accepts w, then the allowed sequence starting with w contains at most $p(size(w))$ words and the size of each word in the sequence is upper bounded by $p(size(w))$. The class of decision problems for which there exists a polynomial-time algorithm for solving them is denoted by $\mathcal{P}$.

Another important class of decision problems is the class $\mathcal{NP}$. This class consists of those decision problems $\Pi \subseteq \Sigma^*$ for which there exist a decision problem Π' in $\mathcal{P}$ and a polynomial $p(x)$ such that for any word $w \in \Sigma^*$ we have that: $w \in \Pi$ if and only if there exists a word v such that $(w, v) \in \Pi'$ and such that size$(v) \leq p($size$(w))$. Such a word v is called a **certificate** that w belongs to Π.

Remark: The letters $\mathcal{NP}$ stand for "nondeterministic polynomial time", which is a concept related to the one of nondeterministic algorithm. As a matter of fact, a nondeterministic algorithm for a decision problem always answer "no" for a "no"-instance, and for a "yes"-instance, there is a positive probability that it answer "yes". In particular, $\mathcal{NP}$ does not mean "not polynomial time".

Example A.5 - Examples of problesm in $\mathcal{NP}$.
Consider the following two problems:

$\Pi_1 = \{G : G$ is a bipartite graph$\}$, and

$\Pi_2 = \{G : G$ is a Hamiltonian graph$\}$.

These problems belong to $\mathcal{NP}$ since the problems:

$\Pi'_1 = \{(G, (X, Y)) : G$ is a graph and (X, Y) is a bipartition for $G\}$ and

$\Pi'_2 = \{(G, H) : G$ is a graph and H is a hamiltonian cycle of $G\}$

belong to the class $\mathcal{P}$. This is because for Π'_1, the problem of deciding if two disjoint subsets of vertices X, Y of a graph G are the parts of G (i.e., deciding if G can be seen as a bipartite graph with parts X and Y), can be solved in polynomial

time (we only need to verify that each edge of G has one extreme in X and the other one in Y): Similarly, for Π_2' the problem of deciding if a given subgraph H of a graph G is a hamiltonian cycle of G, can be solved in polynomial time (we only need to verify that $V(H) = V(G)$ and $E(H) \subseteq E(G)$). □

Remark: We have that $\mathcal{P} \subseteq \mathcal{NP}$ because if $\Pi \in \mathcal{P}$, we can take the empty word $\diagup$ as a certificate, this is $\Pi' = \{(w, \diagup) : w \in \Pi\}$; which proves that $\Pi' \in \mathcal{P}$.

From the above remark, it is natural to ask whether this inclusion is strict. This is an unsolved question and it is also known as

Conjecture 1 (The Cook-Edmonds-Levin Conjecture.)**.**

$$\mathcal{P} \neq \mathcal{NP}.$$

A decision problem with set of "yes"-instances $\Pi \subseteq \Sigma^*$ belongs to the class co$\mathcal{NP}$ if the complementary problem $\bar{\Pi} = \Sigma^* \setminus \Pi$ belongs to $\mathcal{NP}$ [♡].

Example A.6 - A problem in co$\mathcal{NP}$.
The bipartite graph decision problem Π_1 from Example A.5 belongs to the class co$\mathcal{NP}$, because the problem

$\Pi_1'' = \{G : G$ is a graph, but G is not a bipartite graph$\}$

is equivalent to the problem

$\Pi_1''' = \{G : G$ is a graph containing and odd cycle.$\}$

and this last problem belongs to the class $\mathcal{P}$ because we can find an odd cycle in linear time, for example, by using a breadth-first search algorithm. □

From the above definitions, we can deduce that for any problem Π in $\mathcal{P}$, we have also that $\bar{\Pi}$ is in $\mathcal{P}$, and therefore $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$.

The problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ are those problems for which there exist certificates both in case the answer is "yes" and in case the answer is "no". A theorem giving us the certificates "yes" and "no" is called a **good characterization**.

For most of the problems that is known they are in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ it has been proved that they are also in $\mathcal{P}$; however, there exists some problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ for which we still do not know if they really belong to $\mathcal{P}$. This situation gives rise to the following conjecture.

[♡] Again, $\Sigma^* \setminus \Pi$ may contain words w without sense, so we assume that we have a polynomial time algorithm to decide if a word w is an instance.

Conjecture 2 (Edmonds Conjecture.)**.**

$$\mathcal{P} = \mathcal{NP} \cap co\mathcal{NP}.$$

The Primality Testing Problem (i.e., testing if a given number is prime or not) was a problem known to be in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ that could not be proved to be in $\mathcal{P}$ for many years, but a polynomial time algorithm for solving it, was finally found in 2003 by Agrawal et al. [2]. Another problem in $\mathcal{NP} \cap \text{co}\mathcal{NP}$, for which there is not known a polynomial time algorithm is the problem of determining if two given graphs are isomorphic.

An algorithm $\mathcal{A}$ is a **polynomial-time reduction** of a problem Π' to a problem Π if $\mathcal{A}$ is a polynomial-time algorithm such that for any allowed sequence starting with a word w and ending with a word v, we have that $w \in \Pi'$ if and only if $v \in \Pi$.

A problem Π is called $\mathcal{NP}$**-complete** if for each problem Π' in $\mathcal{NP}$ there exists a polynomial time reduction of Π' to Π.

It is not obvious at all that there exists $\mathcal{NP}$-complete problems. The first problem that was proved to be $\mathcal{NP}$-complete was the Boolean Satisfiability Problem (see, for example, Cook [52]).

Analogosuly, we can define the **co**$\mathcal{NP}$**-complete** class as the class of problems Π such that for each problem Π' in co$\mathcal{NP}$ there exists a polynomial time reduction of Π' to Π. A computational problem Π is called $\mathcal{NP}$**-hard** if all problems in $\mathcal{NP}$ polynomially reduce to Π. This is, $\mathcal{NP}$-hard problems are at least as hard as the hardest problems in $\mathcal{NP}$.

Figure A.5 sketches the two possibilities of inclusions for the classes $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{NP}$-complete for the cases of $\mathcal{P} \neq \mathcal{NP}$ and $\mathcal{P} \neq \mathcal{NP}$. Note that, if $\mathcal{P} = \mathcal{NP}$, some regions would collapse into a single one.

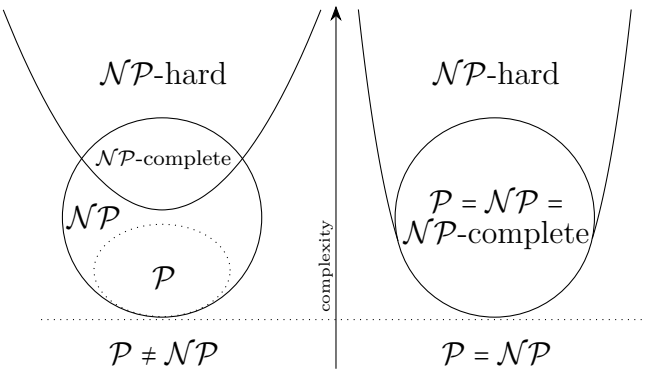


Figure A.5: The inclusions of complexity classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard for the two possibilities $\mathcal{P} \neq \mathcal{NP}$ and $\mathcal{P} = \mathcal{NP}$.

What if a problem is $\mathcal{NP}$-hard?

If we arrive to prove that a problem arising in some real world application is $\mathcal{NP}$-hard then we should stop searching for a polynomial time algorithm, unless we are willing to tackle the Cook-Edmonds-Levin conjecture.

However, we should not be discouraged yet: $\mathcal{NP}$-hardness should be viewed only as a limitation on what can be accomplished with a given problem, and it does not mean that a problem is intractable. In practice, there are many $\mathcal{NP}$-hard problems for which it is possible to compute optimal or good enough solutions in an acceptable amount of time.

Also, it is worth noting that not all the $\mathcal{NP}$-complete problems are equally hard (even when we know they are theoretically equivalent with respect to polynomial time transformations). The success for solving one of those $\mathcal{NP}$-complete problems may depend on the size/structure of the problem instance and on the existence of ad hoc algorithms with some theoretical/empirical performance.

We close this section with an example taken from the book of Bertsimas and Tsitsiklis [29]. The example shows what can be achieved sometimes with an approximated polynomial time algorithm.

Example A.7 - Computing a bound for the TSP in polynomial time
Given a graph $G = (X, E)$ with weights $c(e)$ for every edge $e \in E$, we aim to find a hamiltonian cycle (i.e., a cycle that visits all the vertices exactly once) of minimum weight (with respect to the weight c). Note, this problem is known to be $\mathcal{NP}$-complete.

For modeling the problem, we define for every edge $e \in E$ a variable z_e which is equal to 1 if edge e is included in the cycle, and 0 otherwise. Recall tha for every $U \subseteq X$ we use the notation $\partial(U) = \{e = (x, y) \in E : x \in U, y \notin U\}$, and that $\partial(x)$ is the set of edges incident to x.

Now, because in every cycle, each vertex is incident to exactly two edges, we have

$$\sum_{e \in \partial(\{x\})} z_e = 2,\ x \in X.$$

Also, if we partition the vertices into two nonempty sets U and $X \setminus U$, then in every hamiltonian cycle there are at least two edges conecting U and $X \setminus U$. Hence we have

$$\sum_{e \in \partial(U)} z_e \geq 2,\ U \subset X, U \neq \varnothing,\ U \neq X.$$

The following linear programming problem provides a lower bound to the optimal cost of the TSP:

$$\begin{array}{lrcll} \textit{minimize} & \sum_{e\in E} c(e)\cdot z_e & & & \\ \text{subject to} & \sum_{e\in\partial(x)} z_e & = & 2, & x\in X, \\ & \sum\limits_{e\in\partial(U)} z_e & \geq & 2, & U\subset X, U\neq\emptyset, U\neq X, \\ & z_e & \leq & 1 & \forall e\in E \\ & z_e & \geq & 0 & \forall e\in E \end{array} \tag{A.1}$$

This problem has an exponential number of constraints because there are $2^{|X|}-2$ nonempty proper subsets of vertices. However we can solve this problem in polynomial time in the following way.

Given a vector $\mathbf{z}^*$, we need to check whether it satisfies all the above constraints and if not, to exhibite a violated inequality.

Note that all the constraints can be checked by enumeration in polynomial time, except the constraints involving the subsets $U\subset X$.

However, we can check them also in polynomial time using a classical max-flow-min-cut algorithm. For that, we consider the graph G and assign a capacity z_e^* to every edge e of E. Then we compute a minimumc cut, where the minimum cut is also taken over all choices of the source and sink vertices. Let U_0 be a minimum cut. If the capacity of U_0 is larger than or equal to 2, then the point z^* is feasible because for all U, $\sum_{e\in\partial(U)} z_e^* \geq \sum_{e\in\partial(U_0)} z_e^* \geq 2$.

If not, then the inequality corresponding to U_0 is a violated, that is

$$\sum_{e\in\partial(U)} z_e^* < 2.$$

Finding a minimum cut in a directed graph is equivalent to solving a Maximum Flow Problem, and this can be done in polynomial time. Thus, if we combine this separation procedure with a polynomial time linear programming algorithm (like the ellipsoid algorithm [137]), we can compute this bound in polynomial time.

How good can be such a bound?

Figure A.6 shows the fourteen iterations for solving the above linear programming formulation for an Euclidean TSP instance (arising from the 32 capitals of the states of Mexico) using the method that we have just described. It results that the computed lower bound provides us with an optimal solution. Of course, we have been very lucky; there exist instances where the bound is not that tight. □

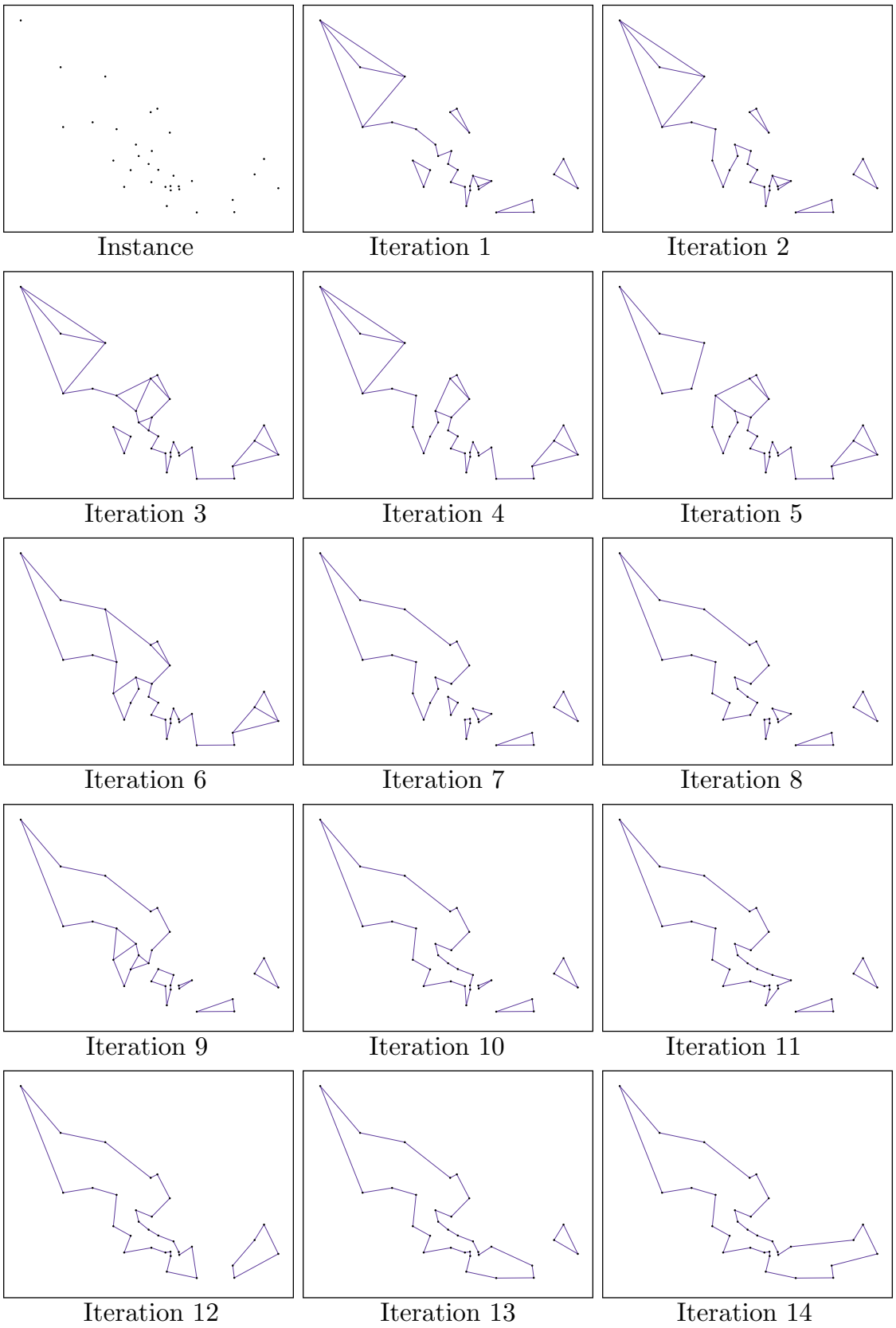


Figure A.6: Computing a lower bound in polynomial time for an Euclidean TSP instance arising from the 32 capitals of the states of Mexico.

A.5 Some Data Structures and Algorithms

In this section we describe some of the data structures and algorithms that were used for carrying out this work.

A.5.1 Indexed Priority Queue

The indexed priority queue is a data structure that allows to maintain a priority queue, with the possibility of changing the priority of the elements at any moment. The complexity of inserting, changing or deleting an element is logarithmic on the number of elements in the priority queue. On the other hand, we can perform in constant time some operations like checking if an element is already present in the priority queue, or reading the element with the highest priority.

The classical priority queues usually allow two basic operations: one *push* operation which consists in the insertion of a new element into the priority queue, and one *pop* operation which consists of reading an element with the highest priority, followed by removing it from the priority queue. Priority queues are commonly implemented by using a binary heap and forbid to the users the possibility of modifying the entries already present at the data structure. This last characteristic is known as *immutability of keys* and it is related to the fact that the modification of a single entry might invalidate the heap-order invariant on which is based the whole data structure. For example, the priority queues that are implemented in the STL library of the C++ language, have immutable keys.

Still, there are some algorithms in which we obtain more information during the execution, and we would like to have the possibility of updating the priority of some elements, and avoid the construction of a new data structure with the new priorities.

As typical examples we can mention some implementations of the Dijkstra's algorithm to compute the shortest paths starting at a given vertex of graph, and the Jarnik-Prim algorithm to compute the minimum spanning tree of a weighted graph. In both cases, we maintain a boolean array `explored` indexed over the set of vertices, and we use it to keep track of the explored vertices. At a first step, we start by putting a specified vertex into the priority queue. Then, at the general step, we pop a highest priority vertex x from the priority queue, and we mark it as explored. Next, we proceed to visit all its non-explored neighbors. This means, to consider every edge $e = (x, y)$ such that y has not been marked as explored, and

- if y is in the priority queue, and e improves its priority, then we update the priority of y;

- if y is not into the priority queue, then we push y into the priority queue, giving it a priority according to the weight of e.

For some applications in competitive environments, there can be a non-neglecting difference of performance between an indexed priority queue and a classical priority queue. On the other hand, there are more specialized data structures (e.g., Fibonacci heaps or d-ary heaps) with the same capabilities of an indexed priority queue, and with slightly better theoretical complexities. However, these structures are usually much more difficult of implementing and sometimes are not as efficient in practice when compared with the theoretically less efficient forms of heaps (see [97]).

An Implementation of Indexed Priority Queue

We use a constant `maxN` as an upper bound for the number of the elements that we are going to push into the priority queue. Then, we use a variable `N` to keep track of the number of elements that are present into the priority queue. The internal structure of the priority queue relies on three arrays:

1. an array `keys` to store the elements (keys, numbers, or other objects) that will be inserted into the priority queue. Note that, those elements must be taken from a set with a total ordering $\preceq$. The ordering $\preceq$ is used to establish the priorities of the elements in the priority queue;

2. an array `heap` of nonnegative integers encoding the complete binary tree that it is being used by the min-indexed priority queue;

3. an array `index` of nonnegative integers such that, `index[i]` is the index in the array `heap`, of the element `keys[i]`.

Example A.8 - An example of min-indexed priority queue
The min-indexed priority queue depicted in Figure A.7, follows an alphabetic ordering and it is implemented with the help of the following arrays.

i	0	1	2	3	4	5	6	7	8
keys[i]	A	S	O	R	T	I	(N)	G	-
heap[i]	-	0	(6)	7	2	1	5	4	3
index[i]	1	5	4	8	7	6	(2)	3	-

We can check that:

- `heap[1]`=0 means that the element `keys[0]` (the letter "A") has priority 1;
- `heap[2]`=6 means that the element `keys[6]` (the letter "N") has priority 2;
- `heap[3]`=7 means that the element `keys[7]` (the letter "G") has priority 3;
- `heap[4]`=2 means that the element `keys[2]` (the letter "O") has priority 4;
- `heap[5]`=1 means that the element `keys[1]` (the letter "S") has priority 5;
- `heap[6]`=5 means that the element `keys[5]` (the letter "I") has priority 6;
- `heap[7]`=4 means that the element `keys[4]` (the letter "T") has priority 7;
- `heap[8]`=3 means that the element `keys[3]` (the letter "R") has priority 8;
- `index[0]`=1 means that `heap[1]`=0;
- `index[1]`=5 means that `heap[5]`=1;
- `index[2]`=4 means that `heap[4]`=2;
- `index[3]`=8 means that `heap[8]`=3;
- `index[4]`=7 means that `heap[7]`=4;
- `index[5]`=6 means that `heap[6]`=5;
- `index[6]`=2 means that `heap[2]`=6;
- `index[7]`=3 means that `heap[3]`=7.

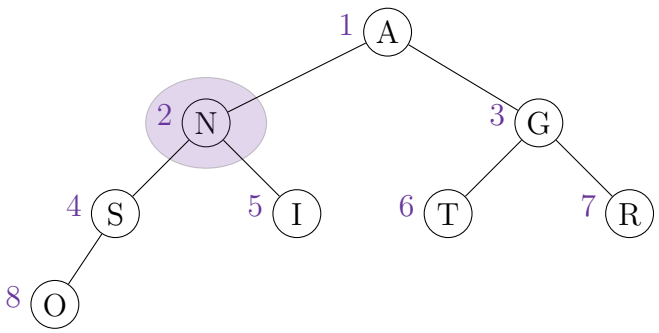


Figure A.7: The min-indexed priority queue used in Example A.8.

Private Methods.

- `swap`. This private method receives two indices `i` and `j`, and exchanges the entries of the array `heap` with indices `i` and `j`. Then, it also updates the array `index` by redefining `index[heap[i]]`=`i`, and `index[heap[j]]`=`j`. This method can be used by other methods (e.g., `swim`, `sink`, and `delMin`) to perform the necessary changes in the array `heap`. Note that it is declared as private method because if it is used in an incorrect way, it can destroy the heap order.

- `swim`. When a node's key becomes smaller and the heap order is violated (i.e., the node is smaller than its parent), we can restore the heap order by interchanging the node with its parent. Such an exchange fix the heap order at current level (because the node is smaller than both its children). But the node may still be smaller than its parent, and in that case, we need to repeat

the process. If we continue in this way, moving up the heap, we eventually restore the heap order.

- `sink`. When a node's key becomes larger and the heap order is violated (i.e., the node is larger than one or both of its children), we can restore the heap order by interchanging the node with its smaller child. Such an exchange fix the heap order at current level. But the node may still be larger than one or both of its children, and in that case, we need to repeat the process. If we continue in an iterative way, moving downd the heap, we eventually restore the heap order. This method is called during the `delMin` operation.

Public Methods.

- `contains`. This method receives an index `i` (i.e., a non-negative integer), and returns a boolean `true` value if the key with index `i` is in the priority queue; otherwise, this method returns a boolean `false` value.
- `isEmpty`. This method does not receive parameters. It returns a boolean `true` value if the priority queue is empty;, otherwise, it returns a boolean `false` value.
- `size`. This method does not receive parameters. It only returns the number of elements that are currently into the priority queue.
- `insert`. This method receives an index `i` and a new key `k`. We add the new key at the end of the array `keys`, increment `N` (i.e., the size of the heap), and then swim up through the heap with that key to restore the heap order.
- `delMin`. We take the smallest key off the top, put the item from the end of the heap at the top, decrement `N` (i.e., the size of the heap), and then sink down through the heap with that key to restore the heap order.
- `decreaseKey`. This method receives an index `i` (i.e., a non-negative integer) and a key element `k` that is smaller than the element in `keys[i]`. The method updates the array `keys` by doing `keys[i]=k`, as a result the entry in the array `heap` corresponding to the element `k`, can violate the heap order, therefore the method calls the `swim` with the parameter `index[i]`, to restore the heap order.

Implementation in the C++ Language

The following implementation is based on the description given by Robert Sedgewick and Kevin Wayne in [192].

```
1  //  IndexMinPQ.h
2  #ifndef IndexMinPQ_h
3  #define IndexMinPQ_h
4  #include <stdio.h>
5  #include <vector>
6
7  class IndexMinPQ
8  {
9   private:
10  int maxN, N, *heap, *index, *keys;
11
12  // swap the entries in the given indices of heap and index arrays
13  void swap(int, int);
14
15  // bottom-up reheapify
16  void swim(int);
17
18  // top-down reheapify
19  void sink(int);
20
21  public:
22  // constructor
23  IndexMinPQ(int);
24
25  // destructor
26  ~IndexMinPQ();
27
28  // is i an index on the priority queue?
29  bool contains(int);
30
31  // is the priority queue empty?
32  bool isEmpty();
33
34  // number of items in the priority queue
35  int size();
36
37  // associate key with index i
38  void insert(int, int);
39
40  // remove a minimal item and returns its index
41  int delMin();
42
43  // decrease the key associated with index i
44  void decreaseKey(int, int);
45 };
46 #endif /* IndexMinPQ_h */
```

```
1  //  IndexMinPQ.cpp
2  #include "IndexMinPQ.h"
3
4  IndexMinPQ::IndexMinPQ(int maxN){
5   this->maxN = maxN;
6   N = 0;
7   keys = new int[maxN + 1];
8   heap = new int[maxN + 1];
9   index = new int[maxN + 1];
10
11  for(int i = 0; i <= maxN; ++i){
12   index[i] = -1;
13  }
14 }
15
16 IndexMinPQ::~IndexMinPQ(){
17  delete [] keys;
18  delete [] heap;
19  delete [] index;
20 }
21
22 bool IndexMinPQ::contains(int i){
23  return index[i] != -1;
24 }
25 bool IndexMinPQ::isEmpty(){
26  return N == 0;
27 }
28
29 int IndexMinPQ::size(){
30  return N;
31 }
32
33 void IndexMinPQ::swap(int i, int j){
34  int t = heap[i];
35  heap[i] = heap[j];
36  heap[j] = t;
37  index[heap[i]] = i;
38  index[heap[j]] = j;
39 }
40
41 void IndexMinPQ::swim(int k){
42  while(k > 1 && keys[heap[k/2]] > keys[heap[k]]){
43   swap(k, k/2);
44   k = k/2;
45  }
46 }
47
```

```
48  void IndexMinPQ::sink(int k){
49   int j;
50   while(2*k <= N){
51    j = 2*k;
52    if(j < N && keys[heap[j]] > keys[heap[j+1]]){
53     j++;
54    }
55    if(keys[heap[k]] <= keys[heap[j]]){
56     break;
57    }
58    swap(k, j);
59    k = j;
60   }
61  }
62
63  void IndexMinPQ::insert(int i, int key){
64   N++;
65   index[i] = N;
66   heap[N] = i;
67   keys[i] = key;
68   swim(N);
69  }
70
71  int IndexMinPQ::delMin(){
72   int min = heap[1];
73   swap(1, N--);
74   sink(1);
75   index[min] = -1;
76   heap[N+1] = -1;
77   return min;
78  }
79
80  void IndexMinPQ::decreaseKey(int i, int key){
81   keys[i] = key;
82   swim(index[i]);
83  }
```

A.5.2 Topological Sorting

A topological sort is a digraph traversal in which each vertex is visited only after all its inneighbours are visited. A topological ordering is possible if and only if the graph has no directed cycles. Any acyclic digraph has at least one topological ordering, and it is possible to find a topological ordering in linear time.

Algorithm 12: Topological sort algorithm

Input : A digraph $G = (X, A)$.
Output: A topological order for the vertices of G.

```
1  L ← Nil.                                              [List for the sorted elements]
2  S ← vertices with no incoming arcs.
3  while S ≠ ∅ do                                                          [Main loop]
4      Remove a vertex x from S.
5      Add x to L.
6      for each vertex y with a = (x, y) ∈ G do
7          Remove arc a from G.
8          if ∂⁻_G(y) = ∅ then
9              Insert y into S.
10     if G has arcs then                                          [G is not acyclic]
11         return error "The input digraph is not acyclic"
12     else                                        [Return a topologically sorted order]
13         return L.
```

A.5.3 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding minimum weight paths of a digraph with nonnegative weight on the arcs. For a given source vertex in the digraph, the algorithm finds the minimum weight path between that vertex and every other. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later [70].

Algorithm 13: Dijkstra's algorithm

Input : A digraph G, weight function $c : A(G) \to \mathbb{R}_+$ and a vertex $s \in V(G)$.
Output: Shortest paths from s to all $v \in V(G)$ with their lengths. More precisely, we get the outputs $l(v) \in \mathbb{R}_+$ and $p(v) \in V(G)$ for all $v \in V(G) \setminus \{s\}$. $l(v)$ is the length of a shortest s-v-path, which consists of a shortest s-$p(v)$-path together with the arc $(p(v), v)$. If v is not reachable from s, then $l(v) = +\infty$ and $p(v)$ is undefined.

```
1  Set l(s) ← 0. Set l(v) ← ∞ for all v ∈ V(G) ∖ {s}. Set S ← ∅.   [Initialization]
2  Find a vertex v ∈ V(G) ∖ S such that l(v) = min_{w∈V(G)∖S} l(w).      [Next vertex]
3  Set S ← S ∪ {v}.                                                    [Update S]
4  for all w ∈ V(G) ∖ S such that (v, w) ∈ A(G) do                  [Explore ∂⁺(v)]
5      if l(w) > l(v) + c((v, w)) then                          [Improved distance?]
6          set l(w) ← l(v) + c((v, w)) and p(w) ← v.          [Update l(w) and p(w)]
7  if S ≠ V(G) then go to the step 2.                                    [Done?]
```

A.5.4 Ford-Fulkerson Algorithm

The Ford–Fulkerson algorithm is a greedy algorithm that computes the maximum flow in capacited network. It is sometimes called a "method" instead of an "algorithm" because the way for finding augmenting paths in a residual graph is not fully specified or it is specified in several implementations with different running times. The algorithm was published in 1956 by L. R. Ford Jr. and D. R. Fulkerson [93].

In order to describe the algorithm, we need the following definitions.

For a given digraph $G = (X, A)$, we define a digraph $\overleftrightarrow{G} = (X, A \sqcup \{\overleftarrow{a} : a \in A\})$, where for $a = (x, y) \in A(G)$ we define $\overleftarrow{a}$ to be a new arc from y to x. We call $\overleftarrow{a}$ the *reverse arc* of a and vice versa.

Given a digraph G with capacities $cap : A(G) \to \mathbb{R}_+$ and a flow f, we define *residual capacities* $cap_f : A(\overleftrightarrow{G}) \to \mathbb{R}_+$ by $cap_f(a) = cap(a) - f(a)$ and $cap_f(a) = f(a)$ for all $a \in A(G)$. The *residual digraph* G_f is the digraph $(X, \{a \in A(\overleftrightarrow{G}) : cap_f(a) > 0\})$.

Given a flow f and a path P in G_f, to *augment* f along P by a positive quantity ϑ means to do the following for each $a \in A(P)$: if $a \in A(G)$ then increase $f(a)$ by ϑ, otherwise, if $a = \overleftrightarrow{a_0}$ for $a_0 \in A(G)$ then decrease $f(a_0)$ by ϑ.

Given a network $N(\hat{s}, \hat{p})$ on a digraph $G = (X, A)$ and an $\hat{s}$-$\hat{p}$-flow f, an *f-augmenting path* is an $\hat{s}$-$\hat{p}$-path in the residual digraph G_f.

Algorithm 14: Ford-Fulkerson Algorithm

Input : A network $N(\hat{s}, \hat{p})$ on a digraph $G = (X, A)$ with a capacity function $cap : A(G) \to \mathbb{R}_+$.

Output: An $\hat{s}$-$\hat{p}$-flow f of maximum value.

1 Set $f(a) = 0$ for all $a \in A(G)$. `[Initialization]`

2 Find an f-augmenting path P. **If** none exists **then stop**.

3 Compute $\vartheta = \min_{a \in A(P)} cap_f(a)$. Augment f along P by ϑ and go to the step 2.

It is known that if the capacities of a network are integers, then there exists an integral maximum flow that can be found with the Ford-Fulkerson algorithm.

Still, the Ford-Fulkerson algorithm may attain an exponential complexity (or even iterate infinitely if the capacities are not integral) if we do not select carefully the augmenting paths for augmenting the flow at each iteration.

For that reason, Edmonds and Karp [80] developed the following two heuristics for selecting augmenting paths and avoid the worst-case scenarios.

- The "fat pipe" heuristic: choose the augmenting path with the largest bottleneck value (i.e., the value ϑ computed in the step 3 of the algorithm) at each iteration.
- The "fewest pipes" heuristic: choose an augmenting path with a minimum number of arcs at each iteration.

Using one of these heuristics the Ford-Fulkerson algorithm becomes a polynomial-time algorithm.

Appendix B

Detailed computational results

Numerical results

Tables legends

Id: Index of the instance.

n: Parameter for the size of the integer grid used for the instance. A natural number n in this column indicates that, the points of the instance were taken from the set $\{(a, b) \in \mathbb{Z} \times \mathbb{Z}, 0 \leq a, b, \leq n\}$.

$|X|$: Numbers of points in the instance.

$|V|$: Number of tours in the instance.

Ω: Upper limit of the time horizon.

$V\ Av$: Average number of vertices by tour.

o_r: Coordinates of the origin o_r of the request

d_r: Coordinates of the destination d_r of the request

ℓ_r: load of the request

$A\ Av$: Average number of available arcs by tour (i.e., arcs with enough capacity to transport the load of the request)

tour weight av.: Average *time* weight of a tour.

ρ: $|V|$ times the maximum time available for traversing a tour divided by the sum of the tours *time* weight

$|A'|$: Number of pre-existing transfers in the instance

strat: Strategy used for generating the instance.

No Transfer The cost of the solution without transfers computed through enumeration.

VAL A* The value computed by the Virtual A* algorithm.

CPU A* The running time used for computing VAL A* with the Virtual A* algorithm.

Transfer A* Number of transfers in the optimal solution found with the Virtual A* algorithm.

VAL DI The minimum cost between the best cost of a solution without transfers and the solution computed by Dijkstra 1-PDPT.

CPU DI The running time used for computing VAL DI with the Dijkstra 1-PDPT algorithm.

Transfer DI Number of transfers in the solution with cost VAL DI.

V min Minimum number of vertices in a tour.

V max Maximum number of vertices in a tour.

$LoadAv$ Average load in a tour.

$Load$ min Minimum load in a tour.

$Load$ max Maximum load in a tour.

$DistAv$ Average distance of a tour.

$Dist$ min Minimum distance of a tour.

$Dist$ max Maximum distance of a tour.

$TimeAv$ Average duration of a tour (including waiting times).

$Time$ min Minimum duration of a tour (including waiting times).

$Time$ max Maximum duration of a tour (including waiting times).

$|R|$ Number of requests in the instance.

NA: This legend means the data cannot be obtained because either there is no solution, or the algorithms can't be initialized (i.e. if there aren't any arcs with enough capacity to pick up or deliver the current request).

CG10: Best cost solution obtained from 10 iterations of the naive GRASP search algorithm.

RG10: Number of inserted requests in the solution with cost CG10.

TG10: Running time (in seconds) used for computing CG10.

CG100: Best cost solution obtained from 100 iterations of the naive GRASP search algorithm.

RG100: Number of inserted requests in the solution with cost CG100.

TG100: Running time (in seconds) used for computing CG100.

CG1K: Best cost solution obtained from 1000 iterations of the naive GRASP search algorithm.

RG1K: Number of inserted requests in the solution with cost CG1K.

TG1K: Running time (in seconds) used for computing CG1K.

CW10: Best cost solution obtained from 10 iterations of the random walk search algorithm.

RW10: Number of inserted requests in the solution with cost CW10.

TW10: Running time (in seconds) used for computing CW10.

CW100: Best cost solution obtained from 100 iterations of the random walk search algorithm.

RW100: Number of inserted requests in the solution with cost CW100.

TW100: Running time (in seconds) used for computing CW100.

CW1K: Best cost solution obtained from 1000 iterations of the random walk search algorithm.

RW1K: Number of inserted requests in the solution with cost CW1K.

TW1K: Running time (in seconds) used for computing CW1K.

CD10: Best cost solution obtained from 10 iterations of the descent search algorithm.

RD10: Number of inserted requests in the solution with cost CD10.

TD10: Running time (in seconds) used for computing CD10.

CD100: Best cost solution obtained from 100 iterations of the descent search algorithm.

RD100: Number of inserted requests in the solution with cost CD100.

TD100: Running time (in seconds) used for computing CD100.

CD1K: Best cost solution obtained from 1000 iterations of the descent search algorithm.

RD1K: Number of inserted requests in the solution with cost CD1K.

TD1K: Running time (in seconds) used for computing CD1K.

CSA: Best cost solution obtained with the threshold accepting algorithm with parameters: `batch_size` = 15, `temperature` = $4 \cdot n$, `cooling _factor` = 0.85, and `freezing_point` = n; where n is the size of the integral grid that was considered for constructing the instance..

RSA: Number of inserted requests in the solution with cost CSA.

TSA: Running time (in seconds) used for computing CSA.

Table B.1: 1-Request Insertion PDPT, Instance Characteristics

Id	n	$\|X\|$	$\|V\|$	Ω	$V\ Av$	o_r	d_r	ℓ_r	$A\ Av$	taxicab		E
										tour weight av.	ρ	tour wei
1	25	80	3	75	8.00	23,4	21,15	3	6.33	42.67	1.76	39.3
2	25	80	3	75	8.00	23,6	6,10	1	7.00	42.67	1.76	38.3
3	25	80	3	75	8.00	12,12	19,16	2	7.67	44.67	1.68	39.0
4	25	80	3	75	8.00	24,4	9,24	1	8.00	44.67	1.68	37.6
5	25	80	3	75	8.00	9,16	7,10	2	5.33	46.00	1.63	39.0
6	25	80	3	75	8.00	2,17	13,6	4	7.00	48.00	1.56	43.0
7	25	80	3	75	8.00	14,12	14,17	2	6.67	46.67	1.61	41.3
8	25	80	3	75	8.00	1,13	23,9	4	5.00	47.33	1.58	41.6
9	25	80	3	75	8.30	23,20	9,16	2	6.00	47.33	1.58	43.0
10	25	80	3	75	8.00	21,10	24,4	1	7.67	42.67	1.76	39.6
11	30	90	4	90	9.00	24,27	20,25	1	8.00	55.00	1.64	47.5
12	30	90	4	90	9.00	26,5	11,20	2	7.50	54.50	1.65	46.7
13	30	90	4	90	9.00	15,11	23,19	2	8.00	55.00	1.64	47.7
14	30	90	4	90	9.00	24,11	8,23	3	6.25	56.50	1.59	48.7
15	30	90	4	90	9.00	25,26	26,7	3	7.00	50.50	1.78	44.7
16	30	90	4	90	9.00	24,18	12,6	4	5.75	54.50	1.65	48.0
17	30	90	4	90	9.00	27,17	3,27	3	7.00	55.00	1.64	47.0
18	30	90	4	90	9.00	19,23	15,28	2	7.00	56.50	1.59	50.2
19	30	90	4	90	9.00	9,27	4,28	2	6.75	53.50	1.68	46.2
20	30	90	4	90	9.00	20,3	6,11	1	8.25	57.50	1.57	51.0
21	35	100	5	105	10.00	24,27	33,5	4	5.40	65.60	1.60	58.2
22	35	100	5	105	10.00	6,26	17,12	2	8.20	65.60	1.60	56.4
23	35	100	5	105	10.00	26,4	32,14	2	8.20	62.80	1.67	54.6
24	35	100	5	105	10.00	34,20	25,22	3	6.60	65.60	1.60	57.2
25	35	100	5	105	10.00	16,20	34,33	3	7.20	62.40	1.68	55.2
26	35	100	5	105	10.00	25,26	7,5	1	8.80	63.20	1.66	55.8
27	35	100	5	105	10.00	6,18	14,12	2	8.60	65.20	1.61	58.2
28	35	100	5	105	10.00	27,17	15,15	1	9.60	64.80	1.62	56.2
29	35	100	5	105	10.00	3,3	13,26	2	7.60	65.60	1.60	57.2
30	35	100	5	105	10.00	30,26	2,15	2	8.80	66.00	1.59	57.4
31	40	110	6	120	11.17	14,26	35,16	4	7.00	76.00	1.58	67.3
32	40	110	6	120	11.00	31,30	13,32	2	9.00	76.67	1.57	66.5
33	40	110	6	120	11.00	26,31	28,21	1	10.17	75.00	1.60	68.0
34	40	110	6	120	11.00	23,28	8,2	3	8.17	75.67	1.59	65.1
35	40	110	6	120	11.17	27,22	38,19	1	9.83	75.00	1.60	65.3
36	40	110	6	120	11.17	39,28	31,9	3	9.50	75.33	1.59	66.6
37	40	110	6	120	11.17	23,6	13,0	1	10.17	75.67	1.59	65.6
38	40	110	6	120	11.00	21,12	34,0	1	10.67	76.00	1.58	70.1
39	40	110	6	120	11.00	37,29	15,33	4	8.33	72.67	1.65	66.5
40	40	110	6	120	11.17	6,23	21,15	1	10.67	74.33	1.61	66.0
41	45	120	7	135	9.00	16,13	4,8	2	7.71	111.71	1.21	91.5
42	45	120	7	135	9.14	37,14	12,8	2	7.71	98.29	1.37	83.5
43	45	120	7	135	9.00	11,19	35,43	3	7.00	106.29	1.27	87.5
44	45	120	7	135	9.00	8,3	2,24	4	5.86	104.29	1.29	88.5
45	45	120	7	135	9.14	13,32	1,41	2	7.71	107.14	1.26	89.1
46	45	120	7	135	9.00	22,29	38,36	4	5.86	99.14	1.36	82.2
47	45	120	7	135	9.29	35,19	35,1	1	8.29	105.71	1.28	90.2
48	45	120	7	135	9.29	25,40	41,30	3	6.57	106.86	1.26	91.2
49	45	120	7	135	9.00	32,23	41,28	1	8.57	104.86	1.29	89.0
50	45	120	7	135	9.00	16,19	0,11	2	7.86	96.86	1.39	78.5

continued on next page...

Table B.1 – Continued

Id	n	$\|X\|$	$\|V\|$	Ω	$V\ Av$	o_r	d_r	ℓ_r	$A\ Av$	taxicab		E
										tour weight av.	ρ	tour weig
51	50	130	8	150	9.00	33,39	32,20	2	7.00	115.75	1.30	96.5
52	50	130	8	150	9.13	9,15	25,20	2	7.25	112.75	1.33	94.2
53	50	130	8	150	9.25	5,26	44,18	1	8.75	110.75	1.35	95.7
54	50	130	8	150	9.25	9,16	26,48	4	5.13	117.25	1.28	96.2
55	50	130	8	150	9.00	46,17	31,42	4	6.50	113.25	1.32	94.6
56	50	130	8	150	9.00	2,31	31,28	2	7.13	113.00	1.33	93.7
57	50	130	8	150	9.25	46,44	11,39	2	8.00	119.25	1.26	102.
58	50	130	8	150	9.00	25,3	9,38	1	7.50	119.25	1.26	99.8
59	50	130	8	150	9.13	26,33	11,1	2	8.25	119.50	1.26	102.
60	50	130	8	150	9.13	10,19	44,17	4	6.13	112.50	1.33	98.3
61	55	140	9	165	9.00	32,49	3,52	3	6.56	130.44	1.26	107.
62	55	140	9	165	9.11	21,41	26,10	2	7.56	124.22	1.33	103.
63	55	140	9	165	9.11	19,6	18,33	3	6.67	124.89	1.32	105.
64	55	140	9	165	9.00	12,1	28,15	3	6.78	118.44	1.39	99.7
65	55	140	9	165	9.00	22,40	5,49	1	8.00	126.44	1.30	102.
66	55	140	9	165	9.22	18,13	45,50	4	7.22	132.00	1.25	110.
67	55	140	9	165	9.11	12,19	15,1	2	7.33	124.44	1.33	101.
68	55	140	9	165	9.22	28,11	5,17	2	7.67	119.56	1.38	102.
69	55	140	9	165	9.11	31,52	46,37	1	8.33	127.78	1.29	106.
70	55	140	9	165	9.00	8,1	17,5	2	7.89	133.33	1.24	113.
71	60	150	10	180	9.00	49,0	49,30	2	7.50	137.40	1.31	117.
72	60	150	10	180	9.10	39,12	26,16	1	8.60	137.80	1.31	114.
73	60	150	10	180	9.20	49,9	59,4	4	6.00	138.60	1.30	115.
74	60	150	10	180	9.10	22,53	41,41	1	8.40	140.20	1.28	117.
75	60	150	10	180	9.10	7,14	37,40	4	5.60	142.60	1.26	116.
76	60	150	10	180	9.40	42,13	31,18	4	6.70	143.40	1.26	119.
77	60	150	10	180	9.10	57,15	13,33	2	7.70	139.40	1.29	116.
78	60	150	10	180	9.10	35,13	11,13	1	8.50	133.40	1.35	114.
79	60	150	10	180	9.00	52,2	17,28	2	7.10	136.60	1.32	116.
80	60	150	10	180	9.00	26,8	40,55	1	8.00	135.00	1.33	115.
81	65	160	11	195	9.27	29,41	62,32	3	7.73	146.00	1.34	120.
82	65	160	11	195	9.09	20,8	38,2	3	7.00	152.00	1.28	127.
83	65	160	11	195	9.09	3,11	23,53	1	8.64	154.00	1.27	131.
84	65	160	11	195	9.09	18,40	40,13	2	7.73	150.73	1.29	128.
85	65	160	11	195	9.09	34,14	22,6	1	8.64	149.46	1.30	123.
86	65	160	11	195	9.27	16,35	10,60	1	8.27	148.73	1.31	123.
87	65	160	11	195	9.18	2,57	28,61	1	8.27	149.46	1.30	122.
88	65	160	11	195	9.00	2,25	25,3	3	7.55	152.73	1.28	126.
89	65	160	11	195	9.18	3,64	20,57	4	6.73	145.64	1.34	125.
90	65	160	11	195	9.09	61,48	8,38	3	6.91	155.46	1.25	128.
91	70	170	12	210	9.08	7,64	59,25	3	6.25	166.00	1.27	140.
92	70	170	12	210	9.08	55,21	57,54	1	8.17	159.17	1.32	133.
93	70	170	12	210	9.00	52,0	28,23	2	7.42	163.67	1.28	134.
94	70	170	12	210	9.00	60,6	61,45	2	7.42	164.67	1.28	134.
95	70	170	12	210	9.42	22,45	60,35	4	6.83	158.00	1.33	132.
96	70	170	12	210	9.08	51,62	21,50	3	6.25	165.33	1.27	140.
97	70	170	12	210	9.08	14,37	39,4	2	7.50	151.33	1.39	123.
98	70	170	12	210	9.17	43,62	59,1	1	8.42	161.50	1.30	135.
99	70	170	12	210	9.25	5,38	23,35	1	8.58	158.67	1.32	134.
100	70	170	12	210	9.08	29,58	38,53	2	7.92	157.00	1.34	132.

continued on next page...

Table B.1 – Continued

Id	n	$\|X\|$	$\|V\|$	Ω	V Av	o_r	d_r	ℓ_r	A Av	taxicab		E
										tour weight av.	ρ	tour weig
101	25	80	3	75	8.33	16,5	20,17	3	6.67	55.33	1.36	48.0
102	25	80	3	75	8.00	14,13	22,9	1	6.33	49.33	1.52	46.0
103	25	80	3	75	8.00	7,22	7,14	2	6.33	53.33	1.41	46.0
104	25	80	3	75	8.00	8,24	6,10	1	7.67	52.00	1.44	45.3
105	25	80	3	75	7.67	20,1	18,5	3	6.33	53.33	1.41	47.6
106	25	80	3	75	7.67	20,23	6,21	4	4.33	52.00	1.44	45.0
107	25	80	3	75	7.33	24,23	23,19	2	7.00	49.33	1.52	46.6
108	25	80	3	75	7.67	3,23	16,11	1	7.67	51.33	1.46	45.0
109	25	80	3	75	7.67	14,8	14,17	2	6.33	52.00	1.44	45.3
110	25	80	3	75	7.67	10,11	17,4	2	7.67	52.00	1.44	44.0
111	30	90	4	90	7.75	12,11	3,3	2	6.75	75.00	1.20	67.0
112	30	90	4	90	8.00	20,8	16,18	1	7.50	67.00	1.34	59.0
113	30	90	4	90	7.75	13,17	24,7	4	6.25	71.00	1.27	59.5
114	30	90	4	90	7.50	9,22	29,21	1	6.75	68.00	1.32	58.5
115	30	90	4	90	8.25	14,0	5,2	3	6.50	60.50	1.49	51.7
116	30	90	4	90	8.00	10,22	9,4	4	5.50	66.50	1.35	58.5
117	30	90	4	90	8.00	27,19	14,10	4	4.50	63.00	1.43	55.2
118	30	90	4	90	8.00	24,13	4,17	4	6.50	66.00	1.36	55.0
119	30	90	4	90	8.00	7,24	9,16	2	6.25	68.50	1.31	61.0
120	30	90	4	90	8.00	11,9	16,26	2	7.00	70.00	1.29	56.2
121	35	100	5	105	8.40	32,13	33,20	2	7.00	76.40	1.37	64.4
122	35	100	5	105	8.80	28,21	6,17	1	8.60	83.60	1.26	72.2
123	35	100	5	105	8.60	15,26	25,22	2	7.20	76.80	1.37	67.0
124	35	100	5	105	8.80	14,11	24,12	3	7.40	84.00	1.25	70.2
125	35	100	5	105	8.60	15,23	34,4	1	8.20	79.60	1.32	70.4
126	35	100	5	105	8.60	8,12	26,29	2	7.80	88.00	1.19	76.4
127	35	100	5	105	8.60	30,11	14,18	4	5.80	72.40	1.45	62.0
128	35	100	5	105	8.40	2,3	27,4	4	6.40	82.80	1.27	69.6
129	35	100	5	105	8.40	17,28	4,12	4	5.00	81.60	1.29	67.8
130	35	100	5	105	8.80	27,4	20,15	3	6.80	82.40	1.27	70.6
131	40	110	6	120	8.83	5,23	18,30	4	6.50	90.67	1.32	78.3
132	40	110	6	120	9.33	1,7	9,12	1	8.50	95.00	1.26	79.8
133	40	110	6	120	9.00	16,22	2,30	4	5.83	93.00	1.29	80.1
134	40	110	6	120	9.67	19,10	19,35	4	7.00	96.00	1.25	81.3
135	40	110	6	120	9.00	21,12	33,22	1	8.33	100.33	1.20	84.1
136	40	110	6	120	8.33	33,5	16,4	2	6.50	102.00	1.18	85.8
137	40	110	6	120	8.67	21,16	17,22	3	6.67	101.67	1.18	88.6
138	40	110	6	120	8.00	29,3	21,24	2	7.17	98.33	1.22	81.8
139	40	110	6	120	8.67	13,13	23,22	1	8.00	96.67	1.24	83.1
140	40	110	6	120	9.17	11,4	15,31	3	7.17	93.67	1.28	79.5
141	45	120	7	135	9.29	34,9	22,6	2	8.29	110.00	1.23	95.4
142	45	120	7	135	8.71	24,14	10,22	4	6.71	112.57	1.20	90.5
143	45	120	7	135	9.14	30,33	32,7	2	7.86	104.29	1.29	90.1
144	45	120	7	135	9.14	15,38	6,10	1	8.29	105.14	1.28	90.0
145	45	120	7	135	9.43	14,34	9,18	3	7.86	107.14	1.26	93.7
146	45	120	7	135	9.71	37,26	23,31	1	9.43	108.29	1.25	93.8
147	45	120	7	135	9.71	39,35	17,5	2	8.14	106.00	1.27	91.4
148	45	120	7	135	8.71	28,30	13,24	4	6.00	111.71	1.21	95.0
149	45	120	7	135	8.71	3,31	20,11	2	8.29	113.71	1.19	94.1
150	45	120	7	135	9.43	17,18	8,22	4	6.86	113.71	1.19	98.8

continued on next page...

Table B.1 – Continued

Id	n	$\|X\|$	$\|V\|$	Ω	$V\ Av$	o_r	d_r	ℓ_r	$A\ Av$	taxicab		E
										tour weight av.	ρ	tour weig
151	50	130	8	150	8.63	41,5	13,19	2	6.88	126.50	1.19	108.
152	50	130	8	150	8.13	47,34	44,21	2	6.75	121.50	1.23	104.
153	50	130	8	150	8.50	41,35	45,25	3	6.63	127.00	1.18	106.
154	50	130	8	150	8.88	8,46	8,13	3	6.88	124.00	1.21	105.
155	50	130	8	150	9.00	5,12	28,43	2	8.00	127.75	1.17	106.
156	50	130	8	150	8.88	34,20	18,39	2	8.00	118.50	1.27	105.
157	50	130	8	150	9.88	35,19	13,45	3	7.25	116.25	1.29	100.
158	50	130	8	150	9.00	33,39	27,2	1	8.38	127.00	1.18	110.
159	50	130	8	150	9.63	23,3	38,6	1	9.00	126.50	1.19	109.
160	50	130	8	150	9.13	27,42	44,17	1	8.38	122.50	1.22	104.
161	55	140	9	165	9.33	38,39	53,15	2	8.11	134.89	1.22	114.
162	55	140	9	165	8.56	45,29	30,0	3	6.11	131.78	1.25	110.
163	55	140	9	165	8.33	22,0	42,6	3	7.11	129.11	1.28	110.
164	55	140	9	165	8.33	29,35	5,19	1	8.00	139.78	1.18	118.
165	55	140	9	165	9.33	52,41	6,21	3	7.56	132.44	1.25	111.
166	55	140	9	165	8.89	16,10	12,25	1	8.33	126.67	1.30	111.
167	55	140	9	165	8.67	24,6	40,32	3	6.56	133.56	1.24	116.
168	55	140	9	165	9.11	47,18	33,8	3	6.78	132.67	1.24	113.
169	55	140	9	165	9.00	39,14	13,34	2	7.22	140.00	1.18	119.
170	55	140	9	165	9.00	32,7	24,42	1	8.67	137.11	1.20	117.
171	60	150	10	180	8.90	2,21	27,34	1	8.50	152.00	1.18	129.
172	60	150	10	180	9.10	40,11	46,20	3	7.10	150.60	1.20	128.
173	60	150	10	180	8.20	20,35	9,9	1	7.80	142.80	1.26	122.
174	60	150	10	180	8.70	14,55	3,41	1	8.20	147.20	1.22	125.
175	60	150	10	180	8.20	29,55	36,4	1	7.90	140.20	1.28	120.
176	60	150	10	180	8.40	22,40	13,14	3	6.50	145.20	1.24	127.
177	60	150	10	180	8.50	46,27	55,33	2	7.10	155.40	1.16	132.
178	60	150	10	180	7.80	21,33	49,1	1	7.30	142.00	1.27	122.
179	60	150	10	180	9.10	55,7	43,32	4	6.20	156.60	1.15	134.
180	60	150	10	180	8.50	27,48	51,49	1	7.40	148.00	1.22	125.
181	65	160	11	195	9.09	2,31	22,41	3	6.55	163.46	1.19	138.
182	65	160	11	195	8.82	58,9	40,54	4	6.27	153.09	1.27	133.
183	65	160	11	195	8.55	18,26	35,6	4	5.82	163.64	1.19	141.
184	65	160	11	195	9.00	8,48	39,23	1	8.55	158.73	1.23	135.
185	65	160	11	195	8.91	36,56	24,44	3	6.82	170.00	1.15	148.
186	65	160	11	195	9.00	38,35	19,17	3	7.36	156.36	1.25	132.
187	65	160	11	195	8.45	18,2	55,17	2	7.09	160.18	1.22	137.
188	65	160	11	195	8.55	37,11	49,5	2	7.27	167.27	1.17	143.
189	65	160	11	195	8.36	29,60	43,21	2	7.18	163.82	1.19	139.
190	65	160	11	195	9.18	7,47	48,44	1	7.82	163.27	1.19	138.
191	70	170	12	210	8.92	11,23	61,12	3	7.17	169.00	1.24	142.
192	70	170	12	210	8.92	39,39	69,45	2	6.92	172.17	1.22	149.
193	70	170	12	210	9.58	30,20	15,38	2	7.75	163.83	1.28	144.
194	70	170	12	210	8.25	62,31	12,23	4	6.17	177.50	1.18	153.
195	70	170	12	210	9.25	23,37	19,13	3	7.25	176.83	1.19	150.
196	70	170	12	210	8.42	40,17	34,32	3	6.00	169.67	1.24	144.
197	70	170	12	210	9.25	60,52	40,22	1	8.50	167.00	1.26	142.
198	70	170	12	210	8.50	28,32	66,1	2	7.00	164.83	1.27	143.
199	70	170	12	210	8.83	29,40	45,37	2	7.75	171.50	1.22	141.
200	70	170	12	210	8.42	64,4	31,55	3	6.83	174.00	1.21	146.

continued on next page...

Table B.1 – Continued

Id	n	$\|X\|$	$\|V\|$	Ω	V Av	o_r	d_r	ℓ_r	A Av	taxicab		E
										tour weight av.	ρ	tour weig
201	25	80	3	75	6.67	4,6	3,15	2	6.00	44.00	1.70	37.5
202	25	80	3	75	7.00	11,16	10,12	3	6.33	51.00	1.47	48.0
203	25	80	3	75	6.33	16,20	10,14	2	5.00	42.00	1.79	35.6
204	25	80	3	75	6.67	4,23	9,11	1	5.33	45.33	1.65	39.6
205	25	80	3	75	6.67	14,8	1,18	1	6.67	41.67	1.80	37.0
206	25	80	3	75	7.00	17,14	7,10	1	6.67	45.33	1.65	40.3
207	25	80	3	75	6.67	4,3	6,23	2	6.33	40.67	1.84	35.6
208	25	80	3	75	6.33	5,17	11,14	2	4.33	40.33	1.86	35.3
209	25	80	3	75	6.33	8,13	15,11	4	4.00	42.33	1.77	37.3
210	25	80	3	75	7.67	16,0	17,23	3	6.33	48.33	1.55	42.6
211	30	90	4	90	7.00	9,13	16,27	1	6.50	65.50	1.37	54.7
212	30	90	4	90	7.00	4,18	21,25	4	5.25	57.00	1.58	48.7
213	30	90	4	90	7.25	8,4	19,26	2	6.50	63.75	1.41	51.2
214	30	90	4	90	7.50	16,19	0,29	2	5.50	58.00	1.55	50.0
215	30	90	4	90	7.00	18,23	0,23	2	6.50	66.50	1.35	58.2
216	30	90	4	90	6.75	18,7	5,8	2	5.50	57.50	1.57	49.0
217	30	90	4	90	7.00	7,24	0,24	1	6.50	57.75	1.56	48.0
218	30	90	4	90	6.75	12,9	17,23	4	4.50	59.75	1.51	50.0
219	30	90	4	90	7.00	0,5	10,4	2	6.25	63.75	1.41	57.2
220	30	90	4	90	7.50	18,8	16,25	1	7.00	57.50	1.57	49.2
221	35	100	5	105	8.00	23,24	1,5	4	5.60	73.40	1.43	61.6
222	35	100	5	105	7.80	4,17	16,31	3	6.00	74.40	1.41	64.2
223	35	100	5	105	7.20	33,33	2,17	2	5.80	71.20	1.47	61.2
224	35	100	5	105	7.80	23,13	16,19	1	7.60	73.20	1.43	62.4
225	35	100	5	105	7.40	32,22	26,14	4	4.60	79.40	1.32	70.4
226	35	100	5	105	8.00	16,9	29,6	3	5.60	73.40	1.43	64.4
227	35	100	5	105	8.00	10,33	31,27	2	7.00	68.80	1.53	57.8
228	35	100	5	105	7.60	13,24	27,30	2	6.80	73.80	1.42	62.2
229	35	100	5	105	7.40	23,11	12,27	3	4.80	73.60	1.43	64.4
230	35	100	5	105	7.00	30,23	18,20	1	6.80	72.40	1.45	62.2
231	40	110	6	120	8.17	3,9	16,8	1	7.67	91.17	1.32	78.1
232	40	110	6	120	8.33	26,1	33,30	1	7.83	84.33	1.42	74.6
233	40	110	6	120	7.67	35,14	14,28	4	5.00	85.17	1.41	73.5
234	40	110	6	120	8.67	27,4	34,11	3	6.33	79.67	1.51	67.5
235	40	110	6	120	8.00	2,9	34,7	1	7.67	84.17	1.43	72.1
236	40	110	6	120	8.50	24,39	22,19	3	6.67	81.50	1.47	70.5
237	40	110	6	120	8.00	34,38	6,7	1	7.67	82.67	1.45	68.6
238	40	110	6	120	8.33	3,5	32,17	1	7.50	87.17	1.38	74.0
239	40	110	6	120	8.00	22,18	12,33	1	7.83	86.00	1.40	72.5
240	40	110	6	120	7.67	10,37	28,23	2	6.50	77.83	1.54	65.8
241	45	120	7	135	8.00	15,19	17,4	2	7.14	98.29	1.37	83.5
242	45	120	7	135	8.00	15,22	34,2	4	6.14	96.43	1.40	83.2
243	45	120	7	135	7.86	5,31	18,10	2	6.71	98.57	1.37	83.5
244	45	120	7	135	8.00	4,18	30,24	2	6.86	88.71	1.52	75.2
245	45	120	7	135	7.57	13,3	22,33	2	7.43	95.43	1.41	81.7
246	45	120	7	135	7.57	9,10	42,38	1	7.14	92.43	1.46	76.4
247	45	120	7	135	7.86	41,13	15,24	4	5.29	92.14	1.47	79.4
248	45	120	7	135	8.14	20,26	8,11	2	7.29	91.00	1.48	76.2
249	45	120	7	135	8.43	43,24	29,12	2	6.43	94.14	1.43	78.5
250	45	120	7	135	8.00	42,17	11,15	2	6.29	98.29	1.37	83.0

continued on next page...

Table B.1 – Continued

Id	n	$\|X\|$	$\|V\|$	Ω	$V\ Av$	o_r	d_r	ℓ_r	$A\ Av$	taxicab		E
										tour weight av.	ρ	tour weig
251	50	130	8	150	9.00	25,0	15,36	3	6.75	106.25	1.41	91.2
252	50	130	8	150	8.00	30,17	29,2	2	6.50	105.63	1.42	92.1
253	50	130	8	150	7.63	44,5	36,2	2	7.00	107.00	1.40	90.0
254	50	130	8	150	8.00	19,15	3,38	1	7.63	104.88	1.43	91.1
255	50	130	8	150	8.38	20,21	32,13	3	6.25	105.00	1.43	91.8
256	50	130	8	150	7.75	11,21	37,30	1	7.00	105.00	1.43	91.1
257	50	130	8	150	7.88	11,0	19,25	2	6.63	109.88	1.37	93.1
258	50	130	8	150	8.50	42,46	29,20	2	7.63	102.63	1.46	85.8
259	50	130	8	150	7.75	26,3	8,34	1	6.38	108.63	1.38	94.5
260	50	130	8	150	8.38	26,21	5,9	2	6.38	104.13	1.44	92.5
261	55	140	9	165	7.44	6,23	29,30	1	6.78	118.56	1.39	99.5
262	55	140	9	165	8.11	28,27	39,42	1	7.56	113.56	1.45	99.4
263	55	140	9	165	8.22	34,33	20,21	1	7.56	113.00	1.46	99.0
264	55	140	9	165	8.00	30,16	47,20	3	5.89	122.67	1.35	107.
265	55	140	9	165	7.78	46,50	25,8	1	7.00	119.89	1.38	100.
266	55	140	9	165	8.11	11,16	30,24	1	7.67	122.67	1.35	107.
267	55	140	9	165	7.89	39,33	7,6	4	4.78	121.33	1.36	103.
268	55	140	9	165	8.78	15,29	24,34	3	6.11	120.89	1.36	102.
269	55	140	9	165	7.67	23,40	48,18	2	6.44	121.56	1.36	104.
270	55	140	9	165	7.89	19,13	40,5	3	6.44	122.44	1.35	105.
271	60	150	10	180	8.30	47,23	25,22	1	7.50	133.40	1.35	114.
272	60	150	10	180	8.70	31,11	24,23	2	7.40	135.30	1.33	116.
273	60	150	10	180	8.10	18,54	40,45	4	5.20	132.00	1.36	112.
274	60	150	10	180	8.20	22,20	18,48	1	7.40	131.80	1.37	114.
275	60	150	10	180	8.00	3,37	19,24	3	6.40	131.40	1.37	115.
276	60	150	10	180	8.30	16,0	40,32	3	6.00	142.10	1.27	122.
277	60	150	10	180	7.80	14,26	16,49	4	5.30	133.50	1.35	114.
278	60	150	10	180	7.90	51,34	34,26	3	6.00	126.30	1.43	109.
279	60	150	10	180	7.70	39,57	51,27	3	6.00	134.90	1.33	116.
280	60	150	10	180	7.90	30,2	9,39	2	6.50	127.60	1.41	110.
281	65	160	11	195	8.09	58,5	12,2	3	6.45	137.64	1.42	116.
282	65	160	11	195	7.91	39,36	45,53	3	5.27	143.91	1.36	120.
283	65	160	11	195	8.00	10,55	62,35	4	5.64	138.82	1.40	119.
284	65	160	11	195	8.09	44,47	61,6	3	6.36	138.64	1.41	115.
285	65	160	11	195	7.27	53,56	36,62	4	5.09	135.91	1.43	120.
286	65	160	11	195	8.09	29,22	12,53	2	7.00	144.00	1.35	124.
287	65	160	11	195	8.09	38,54	4,22	2	7.18	138.55	1.41	119.
288	65	160	11	195	8.00	58,32	38,4	1	6.91	148.00	1.32	127.
289	65	160	11	195	8.45	50,47	6,18	2	6.82	137.64	1.42	121.
290	65	160	11	195	8.18	25,27	47,64	2	7.18	142.36	1.37	122.
291	70	170	12	210	7.92	38,10	34,25	2	6.58	160.08	1.31	138.
292	70	170	12	210	7.83	5,31	55,20	4	5.00	152.67	1.38	134.
293	70	170	12	210	7.58	43,13	31,25	4	5.25	153.67	1.37	130.
294	70	170	12	210	8.00	20,8	48,1	4	5.25	146.50	1.43	127.
295	70	170	12	210	8.08	16,4	57,59	2	6.83	149.92	1.40	126.
296	70	170	12	210	7.83	44,22	29,15	1	7.00	157.75	1.33	137.
297	70	170	12	210	7.58	39,8	12,34	2	6.33	149.75	1.40	129.
298	70	170	12	210	8.00	36,44	8,41	1	7.67	152.67	1.38	131.
299	70	170	12	210	7.83	10,28	49,49	1	7.50	147.58	1.42	124.
300	70	170	12	210	8.50	15,63	18,22	4	5.92	157.17	1.34	131.

Table B.2: Behavior of virtual A* and Dijkstra 1-PDPT without any ε-elegibility

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
1	13	13	0.006	0	13	0.001	0	12	12	0.004	
2	21	21	0.049	0	21	0.002	0	18	18	0.040	
3	11	11	0.011	0	11	0.002	0	9	9	0.012	
4	NA	35	0.086	1	NA	0.001	NA	NA	28	0.056	
5	8	8	0.005	0	8	0.001	0	7	7	0.008	
6	22	22	0.010	0	22	0.001	0	16	16	0.006	
7	5	5	0.007	0	5	0.002	0	5	5	0.004	
8	NA	28	0.006	1	28	0.001	1	NA	31	0.008	
9	18	18	0.009	0	18	0.001	0	18	18	0.006	
10	9	9	0.006	0	9	0.001	0	7	7	0.008	
11	6	6	0.010	0	6	0.003	0	5	5	0.013	
12	NA	30	0.359	1	30	0.004	1	22	22	0.156	
13	16	16	0.026	0	16	0.004	0	12	12	0.021	
14	NA	40	0.021	2	NA	0.001	NA	NA	34	0.017	
15	20	20	0.183	0	20	0.003	0	20	20	0.263	
16	24	24	0.004	0	24	0.001	0	17	17	0.005	
17	34	34	0.158	0	34	0.004	0	27	27	0.129	
18	9	9	0.015	0	9	0.003	0	7	7	0.017	
19	6	6	0.009	0	6	0.003	0	6	6	0.016	
20	22	22	0.079	0	22	0.005	0	17	17	0.040	
21	31	31	0.075	0	31	0.003	0	24	24	0.113	
22	25	25	0.036	0	25	0.003	0	18	18	0.029	
23	16	16	0.023	0	16	0.007	0	12	12	0.027	
24	11	11	0.013	0	11	0.002	0	10	10	0.024	
25	31	31	0.214	0	31	0.006	0	23	23	0.161	
26	NA	39	33.543	1	NA	0.012	NA	NA	32	15.697	
27	14	14	0.022	0	14	0.008	0	10	10	0.028	
28	14	14	0.063	0	14	0.011	0	13	13	0.112	
29	NA	39	0.567	1	NA	0.005	NA	27	27	0.348	
30	39	39	17.141	0	39	0.009	0	31	31	10.406	
31	31	31	0.518	0	31	0.009	0	24	24	0.300	
32	20	20	2.398	0	20	0.024	0	19	19	18.630	
33	12	12	0.458	0	12	0.024	0	11	11	0.423	
34	NA	41	100.400	1	41	0.015	1	31	31	52.275	
35	14	14	0.053	0	14	0.034	0	12	12	0.076	
36	27	27	3.980	0	27	0.025	0	21	21	1.984	
37	16	16	0.155	0	16	0.023	0	12	12	0.105	
38	25	25	3.463	0	25	0.025	0	18	18	1.088	
39	26	26	47.663	0	26	0.016	0	23	23	62.826	
40	23	23	0.217	0	23	0.024	0	17	17	0.066	
41	17	17	0.022	0	17	0.008	0	13	13	0.024	
42	31	31	23.110	0	31	0.016	0	26	26	17.347	
43	NA	48	0.888	1	48	0.011	1	40	37	1.426	
44	NA	31	0.036	1	NA	0.003	NA	22	22	0.034	
45	23	23	0.070	0	23	0.009	0	15	15	0.076	
46	23	23	0.036	0	23	0.007	0	18	18	0.034	
47	NA	34	1.146	1	34	0.009	1	18	18	0.293	
48	26	26	0.075	0	26	0.007	0	19	19	0.100	
49	14	14	0.023	0	14	0.009	0	11	11	0.043	
50	24	24	0.157	0	24	0.014	0	18	18	0.176	

continued on next page...

Table B.2 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
51	20	20	0.129	0	20	0.023	0	20	20	0.261	
52	21	21	0.116	0	21	0.014	0	17	17	0.140	
53	47	47	0.415	0	47	0.009	0	40	40	0.869	
54	NA	79	0.047	1	NA	0.005	NA	62	53	0.154	
55	40	40	0.172	0	40	0.009	0	30	30	0.146	
56	32	32	0.894	0	32	0.019	0	30	30	2.921	
57	NA	52	0.315	2	NA	0.004	NA	NA	41	0.400	
58	NA	53	1.312	1	53	0.013	1	39	39	2.891	
59	NA	49	11.997	1	NA	0.016	NA	36	36	5.384	
60	36	36	0.545	0	36	0.012	0	35	35	1.601	
61	114	58	0.889	1	58	0.009	1	30	30	0.141	
62	36	36	144.011	0	36	0.033	0	32	32	672.075	
63	28	28	0.247	0	28	0.019	0	28	28	0.572	
64	42	42	0.010	0	42	0.001	0	22	22	0.015	
65	26	26	0.101	0	26	0.019	0	20	20	0.139	
66	NA	64	0.697	2	NA	0.017	NA	52	48	1.476	
67	21	21	0.120	0	21	0.019	0	19	19	0.286	
68	29	29	0.079	0	29	0.016	0	24	24	0.165	
69	30	30	0.079	0	30	0.014	0	22	22	0.115	
70	13	13	0.012	0	13	0.003	0	10	10	0.020	
71	30	30	0.099	0	30	0.018	0	30	30	0.472	
72	17	17	0.104	0	17	0.029	0	14	14	0.135	
73	15	15	0.027	0	15	0.009	0	12	12	0.037	
74	31	31	0.043	0	31	0.008	0	23	23	0.075	
75	58	58	0.062	0	58	0.006	0	40	40	0.099	
76	16	16	0.040	0	16	0.009	0	13	13	0.073	
77	94	62	4.008	1	62	0.028	1	48	48	12.444	
78	24	24	0.133	0	24	0.034	0	24	24	0.387	
79	61	61	0.822	0	61	0.011	0	44	44	0.948	
80	NA	75	0.418	1	NA	0.007	NA	50	50	0.754	
81	42	42	0.189	0	42	0.023	0	35	35	0.647	
82	24	24	0.063	0	24	0.017	0	19	19	0.113	
83	NA	64	1.988	1	64	0.007	1	47	47	2.395	
84	49	49	0.909	0	49	0.015	0	35	35	0.643	
85	20	20	0.191	0	20	0.045	0	15	15	0.239	
86	31	31	3.050	0	31	0.046	0	26	26	4.497	
87	34	34	0.026	0	34	0.009	0	27	27	0.131	
88	45	45	16.745	0	45	0.034	0	32	32	6.702	
89	24	24	0.042	0	24	0.007	0	19	19	0.069	
90	111	67	0.350	1	111	0.007	0	54	54	1.121	
91	NA	93	1.826	2	NA	0.014	NA	72	68	2.387	
92	35	35	5.024	0	35	0.066	0	34	34	22.871	
93	NA	47	0.098	1	NA	0.004	NA	34	34	0.069	
94	40	40	0.281	0	40	0.025	0	40	40	1.427	
95	48	48	0.425	0	48	0.021	0	40	40	0.611	
96	42	42	0.059	0	42	0.013	0	33	33	0.100	
97	58	58	0.731	0	58	0.018	0	42	42	0.523	
98	NA	81	68.086	1	NA	0.018	NA	64	64	138.070	
99	21	21	0.022	0	21	0.008	0	19	19	0.053	
100	14	14	0.066	0	14	0.030	0	11	11	0.118	

continued on next page...

Table B.2 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
101	16	16	0.005	0	16	0.001	0	13	13	0.003	
102	12	12	0.003	0	12	0.001	0	9	9	0.004	
103	8	8	0.004	0	8	0.001	0	8	8	0.003	
104	16	16	0.003	0	16	0.000	0	15	15	0.004	
105	6	6	0.003	0	6	0.000	0	5	5	0.003	
106	16	16	0.004	0	16	0.000	0	15	15	0.007	
107	5	5	0.004	0	5	0.001	0	5	5	0.003	
108	47	37	0.064	1	47	0.001	0	18	18	0.022	
109	9	9	0.004	0	9	0.001	0	9	9	0.005	
110	14	14	0.006	0	14	0.001	0	10	10	0.006	
111	17	17	0.007	0	17	0.002	0	14	14	0.007	
112	14	14	0.008	0	14	0.002	0	11	11	0.006	
113	21	21	0.012	0	21	0.001	0	15	15	0.012	
114	23	23	0.006	0	23	0.001	0	22	22	0.011	
115	11	11	0.007	0	11	0.002	0	10	10	0.006	
116	NA	33	0.014	1	33	0.001	1	NA	23	0.010	
117	22	22	0.003	0	22	0.000	0	16	16	0.005	
118	24	24	0.014	0	24	0.002	0	21	21	0.025	
119	10	10	0.011	0	10	0.002	0	9	9	0.014	
120	30	30	0.007	0	30	0.001	0	18	18	0.014	
121	8	8	0.006	0	8	0.002	0	8	8	0.010	
122	26	26	0.049	0	26	0.003	0	23	23	0.129	
123	14	14	0.014	0	14	0.003	0	11	11	0.017	
124	11	11	0.007	0	11	0.002	0	11	11	0.015	
125	50	38	0.061	1	38	0.003	1	43	34	0.093	
126	55	55	0.010	0	55	0.001	0	56	56	0.015	
127	23	23	0.010	0	23	0.002	0	18	18	0.022	
128	58	58	0.039	0	58	0.001	0	50	50	0.090	
129	NA	39	0.022	1	NA	0.002	NA	21	21	0.019	
130	18	18	0.006	0	18	0.001	0	14	14	0.018	
131	20	20	0.019	0	20	0.003	0	15	15	0.020	
132	95	35	0.034	1	95	0.002	0	10	10	0.028	
133	22	22	0.008	0	22	0.002	0	17	17	0.014	
134	NA	37	0.232	1	NA	0.004	NA	27	27	0.219	
135	58	30	0.061	1	30	0.003	1	16	16	0.026	
136	18	18	0.013	0	18	0.001	0	18	18	0.014	
137	10	10	0.006	0	10	0.001	0	8	8	0.014	
138	29	29	0.008	0	29	0.001	0	23	23	0.009	
139	19	19	0.014	0	19	0.003	0	14	14	0.022	
140	53	53	0.012	0	53	0.001	0	28	28	0.014	
141	15	15	0.010	0	15	0.002	0	13	13	0.014	
142	22	22	0.023	0	22	0.005	0	17	17	0.022	
143	NA	56	1.250	2	56	0.011	2	NA	51	2.894	
144	37	37	0.455	0	37	0.009	0	30	30	0.588	
145	21	21	0.016	0	21	0.004	0	17	17	0.027	
146	19	19	0.042	0	19	0.007	0	15	15	0.056	
147	NA	58	0.220	1	NA	0.008	NA	NA	47	0.576	
148	21	21	0.008	0	21	0.002	0	17	17	0.030	
149	37	37	0.134	0	37	0.005	0	27	27	0.303	
150	13	13	0.009	0	13	0.003	0	10	10	0.023	

continued on next page...

Table B.2 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
151	42	42	0.017	0	42	0.002	0	32	32	0.027	
152	16	16	0.012	0	16	0.003	0	14	14	0.015	
153	14	14	0.016	0	14	0.006	0	11	11	0.027	
154	49	43	0.089	1	43	0.007	1	33	33	0.104	
155	58	58	0.683	0	58	0.008	0	39	39	1.491	
156	35	35	0.073	0	35	0.008	0	25	25	0.093	
157	48	48	0.057	0	48	0.008	0	35	35	0.090	
158	75	57	0.790	2	75	0.010	0	40	40	0.616	
159	20	20	0.012	0	20	0.003	0	16	16	0.024	
160	48	48	1.118	0	48	0.014	0	31	31	0.768	
161	125	97	5.745	1	97	0.015	1	29	29	0.985	
162	NA	68	0.186	1	68	0.003	1	33	33	0.054	
163	26	26	0.014	0	26	0.004	0	21	21	0.019	
164	40	40	0.072	0	40	0.007	0	29	29	0.322	
165	NA	66	41.914	1	66	0.014	1	NA	54	56.727	
166	19	19	0.046	0	19	0.011	0	16	16	0.056	
167	46	46	0.024	0	46	0.004	0	31	31	0.026	
168	24	24	0.020	0	24	0.006	0	18	18	0.030	
169	46	46	0.054	0	46	0.006	0	33	33	0.105	
170	NA	73	0.440	2	NA	0.004	NA	49	45	0.522	
171	40	40	0.042	0	40	0.004	0	29	29	0.064	
172	15	15	0.011	0	15	0.002	0	11	11	0.025	
173	39	39	0.054	0	39	0.012	0	29	29	0.089	
174	25	25	0.104	0	25	0.013	0	18	18	0.177	
175	NA	80	16.952	1	NA	0.025	NA	53	53	10.523	
176	35	35	0.056	0	35	0.010	0	28	28	0.097	
177	15	15	0.023	0	15	0.006	0	11	11	0.042	
178	60	60	0.342	0	60	0.010	0	43	43	0.361	
179	NA	49	0.010	1	49	0.001	1	28	28	0.023	
180	25	25	0.050	0	25	0.012	0	25	25	0.093	
181	30	30	0.027	0	30	0.008	0	23	23	0.068	
182	NA	73	0.026	1	73	0.002	1	49	49	0.029	
183	37	37	0.020	0	37	0.005	0	27	27	0.050	
184	104	58	6.432	1	58	0.030	1	40	40	6.500	
185	24	24	0.112	0	24	0.015	0	17	17	0.134	
186	37	37	0.077	0	37	0.020	0	27	27	0.120	
187	100	58	0.124	2	100	0.006	0	40	40	0.160	
188	18	18	0.016	0	18	0.004	0	14	14	0.039	
189	53	53	3.788	0	53	0.023	0	42	42	4.737	
190	44	44	0.243	0	44	0.014	0	42	42	1.174	
191	63	63	0.028	0	63	0.005	0	52	52	0.203	
192	36	36	0.030	0	36	0.011	0	31	31	0.077	
193	33	33	0.031	0	33	0.011	0	24	24	0.059	
194	58	58	0.033	0	58	0.004	0	51	51	0.068	
195	58	58	0.076	0	58	0.009	0	25	25	0.082	
196	21	21	0.015	0	21	0.004	0	17	17	0.032	
197	NA	80	91.754	2	114	0.026	1	37	37	2.380	
198	77	69	0.175	1	69	0.008	1	50	50	0.480	
199	19	19	0.074	0	19	0.035	0	17	17	0.155	
200	NA	124	0.160	1	NA	0.006	NA	61	61	0.104	

continued on next page...

Table B.2 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
201	10	10	0.004	0	10	0.001	0	10	10	0.004	
202	5	5	0.004	0	5	0.001	0	5	5	0.003	
203	12	12	0.004	0	12	0.001	0	9	9	0.005	
204	17	17	0.004	0	17	0.001	0	13	13	0.004	
205	23	23	0.026	0	23	0.002	0	17	17	0.013	
206	14	14	0.004	0	14	0.001	0	11	11	0.005	
207	24	24	0.004	0	24	0.001	0	22	22	0.004	
208	9	9	0.003	0	9	0.000	0	7	7	0.008	
209	9	9	0.004	0	9	0.001	0	8	8	0.003	
210	NA	28	0.011	1	30	0.001	1	NA	25	0.010	
211	21	21	0.013	0	21	0.001	0	16	16	0.011	
212	24	24	0.010	0	24	0.001	0	19	19	0.009	
213	NA	33	0.010	1	NA	0.001	0	25	25	0.008	
214	38	38	0.010	0	38	0.001	0	19	19	0.007	
215	18	18	0.005	0	18	0.001	0	18	18	0.008	
216	14	14	0.004	0	14	0.001	0	14	14	0.006	
217	7	7	0.007	0	7	0.002	0	7	7	0.008	
218	19	19	0.004	0	19	0.000	0	15	15	0.006	
219	11	11	0.004	0	11	0.001	0	11	11	0.004	
220	19	19	0.022	0	19	0.002	0	18	18	0.046	
221	41	41	0.016	0	41	0.002	0	31	31	0.022	
222	26	26	0.006	0	26	0.001	0	19	19	0.007	
223	47	47	0.022	0	47	0.001	0	46	46	0.030	
224	13	13	0.010	0	13	0.002	0	10	10	0.016	
225	14	14	0.004	0	14	0.001	0	10	10	0.005	
226	16	16	0.009	0	16	0.002	0	14	14	0.010	
227	27	27	0.006	0	27	0.001	0	22	22	0.008	
228	20	20	0.019	0	20	0.003	0	16	16	0.031	
229	27	27	0.006	0	27	0.001	0	20	20	0.005	
230	15	15	0.007	0	15	0.002	0	13	13	0.011	
231	14	14	0.010	0	14	0.003	0	14	14	0.015	
232	36	36	0.028	0	36	0.003	0	30	30	0.035	
233	35	35	0.007	0	35	0.001	0	26	26	0.008	
234	14	14	0.019	0	14	0.005	0	10	10	0.023	
235	70	34	0.036	1	46	0.003	1	33	33	0.104	
236	22	22	0.007	0	22	0.002	0	21	21	0.021	
237	NA	59	1.002	1	59	0.005	1	48	43	0.753	
238	65	49	13.584	1	65	0.006	0	32	32	5.252	
239	25	25	0.038	0	25	0.006	0	19	19	0.040	
240	32	32	0.082	0	32	0.004	0	23	23	0.079	
241	17	17	0.013	0	17	0.005	0	16	16	0.033	
242	39	39	0.021	0	39	0.003	0	28	28	0.027	
243	NA	34	0.068	1	NA	0.007	NA	25	25	0.045	
244	32	32	0.030	0	32	0.004	0	27	27	0.042	
245	NA	53	0.046	1	NA	0.002	0	32	32	0.022	
246	61	61	1.361	0	61	0.005	0	44	44	0.921	
247	NA	41	0.030	1	41	0.002	1	29	29	0.025	
248	27	27	0.065	0	27	0.009	0	20	20	0.060	
249	26	26	0.009	0	26	0.003	0	19	19	0.026	
250	NA	77	0.576	2	NA	0.004	NA	32	32	0.186	

continued on next page...

Table B.2 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
251	56	56	0.047	0	56	0.003	0	38	38	0.036	
252	16	16	0.016	0	16	0.005	0	16	16	0.025	
253	11	11	0.020	0	11	0.006	0	9	9	0.041	
254	39	39	0.280	0	39	0.011	0	29	29	0.137	
255	20	20	0.025	0	20	0.006	0	15	15	0.028	
256	35	35	0.035	0	35	0.006	0	28	28	0.045	
257	33	33	0.021	0	33	0.003	0	27	27	0.036	
258	39	39	0.167	0	39	0.008	0	30	30	0.266	
259	NA	61	0.070	1	61	0.003	1	36	36	0.033	
260	NA	89	1.942	2	NA	0.006	NA	NA	65	3.215	
261	30	30	0.039	0	30	0.007	0	25	25	0.057	
262	26	26	0.055	0	26	0.020	0	19	19	0.076	
263	26	26	0.048	0	26	0.018	0	19	19	0.068	
264	21	21	0.023	0	21	0.006	0	18	18	0.031	
265	71	63	3.165	1	63	0.010	1	47	47	2.312	
266	27	27	0.066	0	27	0.013	0	21	21	0.070	
267	101	101	0.059	0	101	0.005	0	42	42	0.027	
268	14	14	0.025	0	14	0.005	0	11	11	0.041	
269	47	47	0.077	0	47	0.007	0	34	34	0.084	
270	29	29	0.013	0	29	0.005	0	23	23	0.022	
271	23	23	0.034	0	23	0.013	0	23	23	0.075	
272	19	19	0.032	0	19	0.007	0	14	14	0.052	
273	31	31	0.024	0	31	0.006	0	24	24	0.026	
274	32	32	0.043	0	32	0.014	0	29	29	0.084	
275	29	29	0.033	0	29	0.014	0	21	21	0.043	
276	NA	66	0.231	1	NA	0.006	NA	40	40	0.101	
277	25	25	0.033	0	25	0.010	0	24	24	0.052	
278	25	25	0.029	0	25	0.012	0	19	19	0.041	
279	42	42	0.061	0	42	0.006	0	33	33	0.093	
280	NA	98	0.410	1	98	0.004	1	43	43	0.052	
281	49	49	0.197	0	49	0.009	0	47	47	0.700	
282	23	23	0.033	0	23	0.014	0	19	19	0.054	
283	72	72	0.250	0	72	0.009	0	57	57	0.436	
284	58	58	0.161	0	58	0.012	0	45	45	0.169	
285	23	23	0.029	0	23	0.009	0	19	19	0.029	
286	48	48	0.084	0	48	0.019	0	36	36	0.091	
287	66	66	4.478	0	66	0.028	0	47	47	3.014	
288	48	48	0.031	0	48	0.005	0	35	35	0.080	
289	73	73	11.290	0	73	0.023	0	53	53	8.846	
290	97	67	33.430	1	97	0.020	0	44	44	3.783	
291	19	19	0.054	0	19	0.011	0	16	16	0.059	
292	61	61	0.068	0	61	0.008	0	52	52	0.096	
293	24	24	0.033	0	24	0.010	0	17	17	0.035	
294	35	35	0.024	0	35	0.007	0	29	29	0.033	
295	NA	102	1.849	2	NA	0.017	NA	73	73	2.810	
296	22	22	0.057	0	22	0.019	0	17	17	0.064	
297	65	65	0.043	0	65	0.004	0	38	38	0.047	
298	31	31	0.156	0	31	0.028	0	29	29	0.299	
299	60	60	0.541	0	60	0.022	0	45	45	0.569	
300	44	44	0.058	0	44	0.008	0	42	42	0.376	

Table B.3: Behavior of virtual A* and Dijkstra 1-PDPT without any ε-elegibility restriction a

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
1	13	13	0.004	0	13	0.001	0	12	12	0.004	
2	21	21	0.041	0	21	0.002	0	18	18	0.039	
3	11	11	0.011	0	11	0.002	0	9	9	0.010	
4	NA	35	0.071	1	NA	0.001	NA	NA	28	0.080	
5	8	8	0.006	0	8	0.001	0	7	7	0.008	
6	22	22	0.006	0	22	0.001	0	16	16	0.006	
7	5	5	0.004	0	5	0.001	0	5	5	0.005	
8	NA	28	0.006	1	28	0.001	1	NA	31	0.008	
9	18	18	0.008	0	18	0.001	0	18	18	0.004	
10	9	9	0.005	0	9	0.001	0	7	7	0.005	
11	6	6	0.008	0	6	0.002	0	5	5	0.010	
12	NA	30	0.197	1	30	0.003	1	22	22	0.101	
13	16	16	0.021	0	16	0.004	0	12	12	0.016	
14	NA	40	0.018	2	NA	0.001	NA	NA	34	0.016	
15	20	20	0.139	0	20	0.003	0	20	20	0.203	
16	24	24	0.004	0	24	0.001	0	17	17	0.007	
17	34	34	0.102	0	34	0.002	0	27	27	0.112	
18	9	9	0.016	0	9	0.003	0	7	7	0.014	
19	6	6	0.008	0	6	0.003	0	6	6	0.009	
20	22	22	0.072	0	22	0.004	0	17	17	0.041	
21	31	31	0.060	0	31	0.003	0	24	24	0.140	
22	25	25	0.036	0	25	0.003	0	18	18	0.026	
23	16	16	0.022	0	16	0.007	0	12	12	0.024	
24	11	11	0.012	0	11	0.003	0	10	10	0.022	
25	31	31	0.180	0	31	0.007	0	23	23	0.146	
26	NA	39	3.482	1	NA	0.008	NA	NA	32	2.875	
27	14	14	0.023	0	14	0.008	0	10	10	0.028	
28	14	14	0.060	0	14	0.011	0	13	13	0.106	
29	NA	39	0.484	1	NA	0.005	NA	27	27	0.289	
30	39	39	2.506	0	39	0.009	0	31	31	2.727	
31	31	31	0.302	0	31	0.008	0	24	24	0.253	
32	20	20	1.146	0	20	0.023	0	19	19	4.369	
33	12	12	0.367	0	12	0.023	0	11	11	0.367	
34	NA	41	8.272	1	41	0.015	1	31	31	8.227	
35	14	14	0.050	0	14	0.030	0	12	12	0.078	
36	27	27	2.045	0	27	0.023	0	21	21	1.203	
37	16	16	0.126	0	16	0.022	0	12	12	0.093	
38	25	25	1.351	0	25	0.024	0	18	18	0.625	
39	26	26	4.314	0	26	0.015	0	23	23	6.167	
40	23	23	0.132	0	23	0.018	0	17	17	0.054	
41	17	17	0.017	0	17	0.006	0	13	13	0.026	
42	31	31	3.169	0	31	0.015	0	26	26	3.744	
43	NA	48	0.500	1	48	0.007	1	40	37	0.881	
44	NA	31	0.036	1	NA	0.005	NA	22	22	0.032	
45	23	23	0.061	0	23	0.010	0	15	15	0.058	
46	23	23	0.034	0	23	0.008	0	18	18	0.045	
47	NA	34	0.589	1	34	0.009	1	18	18	0.284	
48	26	26	0.068	0	26	0.008	0	19	19	0.113	
49	14	14	0.022	0	14	0.009	0	11	11	0.040	
50	24	24	0.128	0	24	0.010	0	18	18	0.132	

continued on next page...

Table B.3 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
51	20	20	0.104	0	20	0.021	0	20	20	0.244	
52	21	21	0.099	0	21	0.013	0	17	17	0.131	
53	47	47	0.303	0	47	0.008	0	40	40	0.692	
54	NA	79	0.039	1	NA	0.003	NA	62	53	0.103	
55	40	40	0.149	0	40	0.009	0	30	30	0.143	
56	32	32	0.465	0	32	0.015	0	30	30	1.329	
57	NA	52	0.247	2	NA	0.004	NA	NA	41	0.325	
58	NA	53	0.803	1	53	0.011	1	39	39	1.645	
59	NA	49	3.118	1	NA	0.015	NA	36	36	2.187	
60	36	36	0.335	0	36	0.013	0	35	35	0.784	
61	114	58	0.581	1	58	0.008	1	30	30	0.129	
62	36	36	7.352	0	36	0.026	0	32	32	15.289	
63	28	28	0.205	0	28	0.019	0	28	28	0.489	
64	42	42	0.010	0	42	0.001	0	22	22	0.016	
65	26	26	0.096	0	26	0.018	0	20	20	0.119	
66	NA	64	0.533	2	NA	0.012	NA	52	48	1.055	
67	21	21	0.107	0	21	0.021	0	19	19	0.283	
68	29	29	0.075	0	29	0.016	0	24	24	0.172	
69	30	30	0.075	0	30	0.010	0	22	22	0.130	
70	13	13	0.012	0	13	0.003	0	10	10	0.020	
71	30	30	0.090	0	30	0.016	0	30	30	0.405	
72	17	17	0.098	0	17	0.026	0	14	14	0.142	
73	15	15	0.025	0	15	0.009	0	12	12	0.043	
74	31	31	0.036	0	31	0.008	0	23	23	0.073	
75	58	58	0.050	0	58	0.004	0	40	40	0.104	
76	16	16	0.035	0	16	0.007	0	13	13	0.073	
77	94	62	1.706	1	62	0.022	1	48	48	4.352	
78	24	24	0.120	0	24	0.030	0	24	24	0.401	
79	61	61	0.481	0	61	0.007	0	44	44	0.837	
80	NA	75	0.327	1	NA	0.006	NA	50	50	0.609	
81	42	42	0.178	0	42	0.021	0	35	35	0.652	
82	24	24	0.053	0	24	0.017	0	19	19	0.130	
83	NA	64	1.202	1	64	0.008	1	47	47	1.600	
84	49	49	0.907	0	49	0.016	0	35	35	0.613	
85	20	20	0.176	0	20	0.041	0	15	15	0.252	
86	31	31	2.271	0	31	0.065	0	26	26	2.455	
87	34	34	0.042	0	34	0.014	0	27	27	0.100	
88	45	45	3.516	0	45	0.051	0	32	32	2.279	
89	24	24	0.058	0	24	0.010	0	19	19	0.063	
90	111	67	0.322	1	111	0.006	0	54	54	0.825	
91	NA	93	0.927	2	NA	0.011	NA	72	68	1.728	
92	35	35	2.305	0	35	0.059	0	34	34	6.898	
93	NA	47	0.087	1	NA	0.006	NA	34	34	0.064	
94	40	40	0.266	0	40	0.025	0	40	40	1.266	
95	48	48	0.328	0	48	0.021	0	40	40	0.566	
96	42	42	0.059	0	42	0.013	0	33	33	0.097	
97	58	58	0.581	0	58	0.017	0	42	42	0.447	
98	NA	81	6.569	1	NA	0.012	NA	64	64	21.036	
99	21	21	0.023	0	21	0.009	0	19	19	0.055	
100	14	14	0.060	0	14	0.026	0	11	11	0.108	

continued on next page...

Table B.3 – Continued

Id	Taxicab distance								Euclidean							
	No Transfer	VAL	A^*	CPU	A^*	Transfer	A^*	VAL	DI	CPU	DI	Transfer	DI	No Transfer	VAL	A^*	CPU	A^*	Transfer	A^*	VAL	DI	CPU	DI	Transfer	DI
101	16	16		0.004		0		16		0.001		0		13	13		0.005		0		16		0.005		0	
102	12	12		0.004		0		12		0.001		0		9	9		0.004		0		12		0.004		0	
103	8	8		0.003		0		8		0.001		0		8	8		0.002		0		8		0.002		0	
104	16	16		0.004		0		16		0.001		0		15	15		0.004		0		16		0.004		0	
105	6	6		0.004		0		6		0.001		0		5	5		0.003		0		6		0.003		0	
106	16	16		0.003		0		16		0.001		0		15	15		0.003		0		16		0.003		0	
107	5	5		0.005		0		5		0.001		0		5	5		0.004		0		5		0.004		0	
108	47	37		0.059		1		47		0.001		0		18	18		0.022		0		47		0.022		0	
109	9	9		0.004		0		9		0.001		0		9	9		0.005		0		9		0.005		0	
110	14	14		0.006		0		14		0.001		0		10	10		0.006		0		14		0.006		0	
111	17	17		0.007		0		17		0.002		0		14	14		0.007		0		17		0.007		0	
112	14	14		0.004		0		14		0.001		0		11	11		0.005		0		14		0.005		0	
113	21	21		0.013		0		21		0.002		0		15	15		0.012		0		21		0.012		0	
114	23	23		0.006		0		23		0.001		0		22	22		0.011		0		23		0.011		0	
115	11	11		0.004		0		11		0.001		0		10	10		0.006		0		11		0.006		0	
116	NA	33		0.012		1		33		0.001		1		NA	23		0.016		0		33		0.016		0	
117	22	22		0.003		0		22		0.001		0		16	16		0.006		0		22		0.006		0	
118	24	24		0.013		0		24		0.001		0		21	21		0.016		0		24		0.016		0	
119	10	10		0.012		0		10		0.002		0		9	9		0.013		0		10		0.013		0	
120	30	30		0.007		0		30		0.001		0		18	18		0.008		0		30		0.008		0	
121	8	8		0.007		0		8		0.002		0		8	8		0.009		0		8		0.009		0	
122	26	26		0.047		0		26		0.003		0		23	23		0.139		0		26		0.139		0	
123	14	14		0.015		0		14		0.004		0		11	11		0.018		0		14		0.018		0	
124	11	11		0.007		0		11		0.002		0		11	11		0.010		0		11		0.010		0	
125	50	38		0.057		1		38		0.004		1		43	34		0.100		1		38		0.100		1	
126	55	55		0.009		0		55		0.001		0		56	56		0.018		0		55		0.018		0	
127	23	23		0.010		0		23		0.002		0		18	18		0.013		0		23		0.013		0	
128	58	58		0.035		0		58		0.001		0		50	50		0.067		0		58		0.067		0	
129	NA	39		0.024		1		NA		0.003		NA		21	21		0.018		0		NA		0.018		0	
130	18	18		0.006		0		18		0.001		0		14	14		0.014		0		18		0.014		0	
131	20	20		0.017		0		20		0.003		0		15	15		0.022		0		20		0.022		0	
132	95	35		0.034		1		95		0.002		0		10	10		0.013		0		95		0.013		0	
133	22	22		0.007		0		22		0.002		0		17	17		0.014		0		22		0.014		0	
134	NA	37		0.162		1		NA		0.005		NA		27	27		0.172		0		NA		0.172		0	
135	58	30		0.057		1		30		0.003		1		16	16		0.021		0		30		0.021		0	
136	18	18		0.011		0		18		0.002		0		18	18		0.015		0		18		0.015		0	
137	10	10		0.006		0		10		0.001		0		8	8		0.008		0		10		0.008		0	
138	29	29		0.009		0		29		0.001		0		23	23		0.023		0		29		0.023		0	
139	19	19		0.015		0		19		0.003		0		14	14		0.023		0		19		0.023		0	
140	53	53		0.012		0		53		0.001		0		28	28		0.014		0		53		0.014		0	
141	15	15		0.010		0		15		0.002		0		13	13		0.015		0		15		0.015		0	
142	22	22		0.017		0		22		0.003		0		17	17		0.026		0		22		0.026		0	
143	NA	56		0.774		2		56		0.011		2		NA	51		1.602		2		56		1.602		2	
144	37	37		0.374		0		37		0.009		0		30	30		0.690		0		37		0.690		0	
145	21	21		0.014		0		21		0.004		0		17	17		0.026		0		21		0.026		0	
146	19	19		0.038		0		19		0.007		0		15	15		0.055		0		19		0.055		0	
147	NA	58		0.218		1		NA		0.008		NA		NA	47		0.606		1		NA		0.606		1	
148	21	21		0.010		0		21		0.002		0		17	17		0.030		0		21		0.030		0	
149	37	37		0.135		0		37		0.005		0		27	27		0.197		0		37		0.197		0	
150	13	13		0.010		0		13		0.004		0		10	10		0.021		0		13		0.021		0	

continued on next page...

Table B.3 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
151	42	42	0.015	0	42	0.002	0	32	32	0.018	
152	16	16	0.011	0	16	0.003	0	14	14	0.017	
153	14	14	0.015	0	14	0.004	0	11	11	0.031	
154	49	43	0.090	1	43	0.007	1	33	33	0.120	
155	58	58	0.517	0	58	0.008	0	39	39	0.923	
156	35	35	0.072	0	35	0.008	0	25	25	0.084	
157	48	48	0.059	0	48	0.008	0	35	35	0.097	
158	75	57	0.360	2	75	0.007	0	40	40	0.421	
159	20	20	0.012	0	20	0.003	0	16	16	0.030	
160	48	48	0.719	0	48	0.014	0	31	31	0.676	
161	125	97	2.553	1	97	0.015	1	29	29	0.774	
162	NA	68	0.125	1	68	0.003	1	33	33	0.064	
163	26	26	0.014	0	26	0.004	0	21	21	0.026	
164	40	40	0.118	0	40	0.012	0	29	29	0.244	
165	NA	66	5.390	1	66	0.014	1	NA	54	9.734	
166	19	19	0.046	0	19	0.013	0	16	16	0.064	
167	46	46	0.024	0	46	0.003	0	31	31	0.024	
168	24	24	0.021	0	24	0.006	0	18	18	0.025	
169	46	46	0.055	0	46	0.005	0	33	33	0.098	
170	NA	73	0.327	2	NA	0.004	NA	49	45	0.470	
171	40	40	0.042	0	40	0.005	0	29	29	0.064	
172	15	15	0.018	0	15	0.003	0	11	11	0.025	
173	39	39	0.051	0	39	0.012	0	29	29	0.089	
174	25	25	0.104	0	25	0.013	0	18	18	0.166	
175	NA	80	5.252	1	NA	0.022	NA	53	53	4.055	
176	35	35	0.059	0	35	0.010	0	28	28	0.086	
177	15	15	0.019	0	15	0.006	0	11	11	0.037	
178	60	60	0.286	0	60	0.010	0	43	43	0.318	
179	NA	49	0.010	1	49	0.001	1	28	28	0.016	
180	25	25	0.044	0	25	0.013	0	25	25	0.082	
181	30	30	0.027	0	30	0.009	0	23	23	0.058	
182	NA	73	0.031	1	73	0.002	1	49	49	0.025	
183	37	37	0.020	0	37	0.005	0	27	27	0.039	
184	104	58	2.522	1	58	0.024	1	40	40	2.015	
185	24	24	0.110	0	24	0.015	0	17	17	0.103	
186	37	37	0.079	0	37	0.017	0	27	27	0.110	
187	100	58	0.170	2	100	0.009	0	40	40	0.132	
188	18	18	0.017	0	18	0.004	0	14	14	0.034	
189	53	53	1.107	0	53	0.018	0	42	42	1.678	
190	44	44	0.232	0	44	0.018	0	42	42	0.906	
191	63	63	0.028	0	63	0.006	0	52	52	0.175	
192	36	36	0.039	0	36	0.012	0	31	31	0.064	
193	33	33	0.028	0	33	0.010	0	24	24	0.051	
194	58	58	0.034	0	58	0.005	0	51	51	0.066	
195	58	58	0.106	0	58	0.015	0	25	25	0.070	
196	21	21	0.016	0	21	0.004	0	17	17	0.031	
197	NA	80	4.203	2	114	0.025	1	37	37	1.102	
198	77	69	0.175	1	69	0.010	1	50	50	0.435	
199	19	19	0.070	0	19	0.032	0	17	17	0.145	
200	NA	124	0.150	1	NA	0.005	NA	61	61	0.099	

continued on next page...

Table B.3 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
201	10	10	0.005	0	10	0.001	0	10	10	0.003	
202	5	5	0.003	0	5	0.001	0	5	5	0.003	
203	12	12	0.004	0	12	0.001	0	9	9	0.004	
204	17	17	0.006	0	17	0.001	0	13	13	0.004	
205	23	23	0.020	0	23	0.001	0	17	17	0.012	
206	14	14	0.005	0	14	0.001	0	11	11	0.006	
207	24	24	0.004	0	24	0.001	0	22	22	0.004	
208	9	9	0.003	0	9	0.001	0	7	7	0.003	
209	9	9	0.003	0	9	0.001	0	8	8	0.003	
210	NA	28	0.008	1	30	0.001	1	NA	25	0.011	
211	21	21	0.011	0	21	0.001	0	16	16	0.011	
212	24	24	0.010	0	24	0.001	0	19	19	0.008	
213	NA	33	0.012	1	NA	0.001	0	25	25	0.008	
214	38	38	0.011	0	38	0.001	0	19	19	0.007	
215	18	18	0.007	0	18	0.001	0	18	18	0.008	
216	14	14	0.004	0	14	0.001	0	14	14	0.005	
217	7	7	0.005	0	7	0.001	0	7	7	0.007	
218	19	19	0.005	0	19	0.001	0	15	15	0.006	
219	11	11	0.004	0	11	0.001	0	11	11	0.003	
220	19	19	0.023	0	19	0.002	0	18	18	0.044	
221	41	41	0.018	0	41	0.002	0	31	31	0.014	
222	26	26	0.007	0	26	0.002	0	19	19	0.007	
223	47	47	0.017	0	47	0.001	0	46	46	0.028	
224	13	13	0.010	0	13	0.003	0	10	10	0.010	
225	14	14	0.005	0	14	0.001	0	10	10	0.005	
226	16	16	0.009	0	16	0.002	0	14	14	0.009	
227	27	27	0.007	0	27	0.002	0	22	22	0.008	
228	20	20	0.019	0	20	0.003	0	16	16	0.024	
229	27	27	0.007	0	27	0.001	0	20	20	0.006	
230	15	15	0.008	0	15	0.002	0	13	13	0.012	
231	14	14	0.011	0	14	0.003	0	14	14	0.015	
232	36	36	0.021	0	36	0.003	0	30	30	0.035	
233	35	35	0.010	0	35	0.002	0	26	26	0.008	
234	14	14	0.017	0	14	0.005	0	10	10	0.017	
235	70	34	0.036	1	46	0.003	1	33	33	0.081	
236	22	22	0.007	0	22	0.002	0	21	21	0.017	
237	NA	59	0.487	1	59	0.007	1	48	43	0.498	
238	65	49	1.683	1	65	0.008	0	32	32	1.378	
239	25	25	0.032	0	25	0.006	0	19	19	0.041	
240	32	32	0.075	0	32	0.004	0	23	23	0.065	
241	17	17	0.013	0	17	0.004	0	16	16	0.021	
242	39	39	0.022	0	39	0.003	0	28	28	0.024	
243	NA	34	0.056	1	NA	0.005	NA	25	25	0.043	
244	32	32	0.026	0	32	0.005	0	27	27	0.042	
245	NA	53	0.044	1	NA	0.003	0	32	32	0.019	
246	61	61	0.763	0	61	0.009	0	44	44	0.681	
247	NA	41	0.029	1	41	0.002	1	29	29	0.024	
248	27	27	0.082	0	27	0.013	0	20	20	0.059	
249	26	26	0.011	0	26	0.003	0	19	19	0.018	
250	NA	77	0.312	2	NA	0.004	NA	32	32	0.152	

continued on next page...

Table B.3 – Continued

Id	Taxicab distance							Euclidea			
	No Transfer	VAL A^*	CPU A^*	Transfer A^*	VAL DI	CPU DI	Transfer DI	No Transfer	VAL A^*	CPU A^*	Tra
251	56	56	0.051	0	56	0.003	0	38	38	0.030	
252	16	16	0.015	0	16	0.005	0	16	16	0.028	
253	11	11	0.021	0	11	0.007	0	9	9	0.030	
254	39	39	0.233	0	39	0.010	0	29	29	0.151	
255	20	20	0.030	0	20	0.010	0	15	15	0.023	
256	35	35	0.050	0	35	0.007	0	28	28	0.037	
257	33	33	0.018	0	33	0.003	0	27	27	0.031	
258	39	39	0.176	0	39	0.009	0	30	30	0.212	
259	NA	61	0.061	1	61	0.003	1	36	36	0.029	
260	NA	89	0.621	2	NA	0.008	NA	NA	65	1.225	
261	30	30	0.037	0	30	0.008	0	25	25	0.049	
262	26	26	0.053	0	26	0.016	0	19	19	0.059	
263	26	26	0.051	0	26	0.018	0	19	19	0.063	
264	21	21	0.040	0	21	0.010	0	18	18	0.032	
265	71	63	1.743	1	63	0.011	1	47	47	1.417	
266	27	27	0.066	0	27	0.012	0	21	21	0.068	
267	101	101	0.058	0	101	0.006	0	42	42	0.025	
268	14	14	0.033	0	14	0.008	0	11	11	0.038	
269	47	47	0.082	0	47	0.009	0	34	34	0.075	
270	29	29	0.012	0	29	0.005	0	23	23	0.022	
271	23	23	0.028	0	23	0.008	0	23	23	0.074	
272	19	19	0.031	0	19	0.008	0	14	14	0.048	
273	31	31	0.018	0	31	0.004	0	24	24	0.022	
274	32	32	0.044	0	32	0.013	0	29	29	0.075	
275	29	29	0.028	0	29	0.008	0	21	21	0.034	
276	NA	66	0.216	1	NA	0.008	NA	40	40	0.100	
277	25	25	0.049	0	25	0.013	0	24	24	0.044	
278	25	25	0.024	0	25	0.008	0	19	19	0.035	
279	42	42	0.057	0	42	0.006	0	33	33	0.082	
280	NA	98	0.240	1	98	0.004	1	43	43	0.044	
281	49	49	0.178	0	49	0.010	0	47	47	0.456	
282	23	23	0.030	0	23	0.009	0	19	19	0.052	
283	72	72	0.198	0	72	0.010	0	57	57	0.392	
284	58	58	0.167	0	58	0.014	0	45	45	0.165	
285	23	23	0.027	0	23	0.008	0	19	19	0.029	
286	48	48	0.073	0	48	0.016	0	36	36	0.086	
287	66	66	2.754	0	66	0.023	0	47	47	2.218	
288	48	48	0.030	0	48	0.005	0	35	35	0.059	
289	73	73	2.684	0	73	0.020	0	53	53	3.105	
290	97	67	4.662	1	97	0.021	0	44	44	1.714	
291	19	19	0.054	0	19	0.011	0	16	16	0.047	
292	61	61	0.063	0	61	0.008	0	52	52	0.094	
293	24	24	0.029	0	24	0.009	0	17	17	0.035	
294	35	35	0.029	0	35	0.008	0	29	29	0.030	
295	NA	102	1.280	2	NA	0.013	NA	73	73	1.963	
296	22	22	0.054	0	22	0.015	0	17	17	0.063	
297	65	65	0.037	0	65	0.004	0	38	38	0.041	
298	31	31	0.143	0	31	0.028	0	29	29	0.251	
299	60	60	0.446	0	60	0.022	0	45	45	0.448	
300	44	44	0.054	0	44	0.010	0	42	42	0.184	

Table B.4: PDPT with multiple requests, Instance Characteristics

Id	n	$\|X\|$	$\|V\|$	Ω	VAv	V min	V max	$LoadAv$	$Load$ min	$Load$ max	$\|A'\|$	$DistAv$	$Dist$ min	$Dist$ m
1	20	15	2	68	6.00	6	6	6.50	6	7	0	42.00	38	
2	15	22	3	45	7.67	7	8	35.67	30	43	1	20.00	18	
3	15	22	3	52	5.00	5	5	22.67	17	26	0	18.00	12	
4	15	22	4	55	4.50	2	6	18.50	0	29	1	17.00	0	
5	25	25	5	90	5.00	4	6	12.80	9	17	2	41.20	30	
6	20	15	2	60	2.00	2	2	0.00	0	0	0	0.00	0	
7	15	22	3	38	2.00	2	2	0.00	0	0	0	0.00	0	
8	15	22	3	38	2.00	2	2	0.00	0	0	0	0.00	0	
9	15	22	4	40	2.00	2	2	0.00	0	0	0	0.00	0	
10	25	25	5	75	2.00	2	2	0.00	0	0	0	0.00	0	
11	25	80	3	140	8.00	8	8	31.33	26	38	1	43.33	40	
12	30	90	4	110	7.75	7	8	33.75	27	47	1	62.25	52	
13	35	100	5	140	8.00	8	8	35.80	28	48	2	71.40	63	
14	40	110	6	150	11.17	11	12	54.67	38	69	3	74.83	65	
15	45	120	7	135	9.29	7	11	33.43	12	47	3	101.71	70	1
16	50	130	8	160	9.00	7	11	42.50	21	63	4	86.50	63	1
17	55	140	9	230	9.00	9	9	40.67	28	52	5	130.44	116	1
18	60	150	10	175	8.90	7	14	41.50	27	64	3	119.90	82	1
19	65	160	11	195	8.45	6	11	32.00	20	44	6	142.82	117	1
20	70	170	12	220	9.08	9	10	42.17	33	57	7	139.33	131	1
21	30	90	4	150	2.00	2	2	0.00	0	0	0	0.00	0	
22	35	100	5	110	2.00	2	2	0.00	0	0	0	0.00	0	
23	40	110	6	120	2.00	2	2	0.00	0	0	0	0.00	0	
24	45	120	7	135	2.00	2	2	0.00	0	0	0	0.00	0	
25	50	130	8	150	2.00	2	2	0.00	0	0	0	28.38	4	
26	55	140	9	150	2.00	2	2	0.67	0	3	0	13.78	0	
27	10	121	3	60	15.33	13	17	29.67	20	45	0	21.33	20	
28	10	121	4	65	7.50	7	8	24.00	18	30	2	40.50	36	
29	70	113	3	440	18.67	18	19	0.00	0	0	1	196.67	172	2
30	20	324	5	130	2.00	2	2	0.00	0	0	0	0.00	0	
31	20	324	5	90	11.80	10	14	0.00	0	0	0	33.60	32	

Table B.5: Virtual A* with cost parameters $\alpha = 1$, $\beta = 0$, and $\gamma = 0$

Id	CG10	RG10	TG10	CG100	RG100	TG100	CG1K	RG1K	TG1K	CW10	RW10	TW10	CW100	RW100	TW100	CW1K	RW1K	TW1K	CD10	RD10	TD10	CD100
1	50	3	0.13	50	3	0.32	50	3	6.06	50	3	0.03	50	3	0.28	50	3	2.58	50	3	0.05	50
2	28	4	0.20	28	4	0.64	28	4	6.66	28	4	0.05	28	4	0.56	28	4	4.85	28	4	0.10	28
3	69	5	0.20	69	5	0.57	69	5	5.54	54	4	0.06	54	4	0.34	69	5	4.01	64	4	0.09	64
4	67	5	0.28	67	5	0.69	67	5	7.51	67	5	0.06	67	5	0.69	67	5	5.54	67	5	0.14	67
5	88	5	0.24	88	5	1.15	88	5	12.39	88	5	0.09	88	5	0.81	88	5	9.39	88	5	0.08	88
6	91	5	0.17	91	5	0.47	91	5	5.11	91	5	0.04	91	5	0.34	91	5	3.72	91	5	0.06	91
7	52	4	0.17	82	5	0.49	82	5	5.26	82	5	0.05	82	5	0.37	82	5	3.83	82	5	0.16	82
8	67	5	0.21	67	5	0.53	67	5	5.05	67	5	0.04	67	5	0.37	67	5	3.55	67	5	0.08	67
9	75	6	0.28	75	6	0.60	75	6	6.32	75	6	0.05	75	6	0.39	75	6	4.50	75	6	0.07	75
10	141	8	0.31	141	8	0.81	141	8	8.50	141	8	0.06	141	8	0.50	141	8	5.74	141	8	0.10	141
11	230	9	0.66	176	9	4.07	229	10	41.29	222	9	0.30	184	9	2.22	251	10	26.55	188	9	1.30	188
12	184	6	0.41	199	7	2.31	261	8	23.81	227	7	0.15	271	8	1.78	257	8	17.21	148	5	0.13	148
13	359	9	0.60	296	9	4.45	276	9	40.80	245	8	0.29	314	9	2.71	292	9	24.94	237	8	0.31	237
14	233	10	4.36	228	10	33.07	223	10	429.46	266	10	2.58	230	10	22.26	223	10	255.59	188	9	0.67	188
15	220	8	1.52	180	8	15.38	180	8	127.76	157	7	4.94	180	8	9.72	180	8	99.02	157	7	0.35	157
16	296	10	1.68	296	10	12.84	296	10	127.32	307	10	0.95	296	10	8.55	296	10	88.65	307	10	0.46	307
17	383	10	5.34	383	10	54.36	383	10	519.71	383	10	4.10	383	10	31.01	383	10	311.57	383	10	1.59	383
18	391	10	4.88	350	10	118.47	350	10	1491.61	350	10	12.02	350	10	177.90	350	10	1288.57	456	10	7.59	456
19	345	9	4.17	444	10	36.79	418	10	354.23	335	9	2.62	502	10	26.61	428	10	255.14	335	9	0.54	335
20	369	10	17.43	369	10	147.03	369	10	1472.63	339	9	11.97	369	10	107.82	369	10	1002.32	339	9	3.24	339
21	474	18	0.34	442	19	2.26	442	19	24.51	420	18	0.24	413	18	1.75	441	19	18.11	392	17	1.04	392
22	383	17	0.27	444	18	1.89	425	18	20.07	319	15	0.16	335	16	1.35	373	17	13.83	310	15	0.41	310
23	472	16	0.30	473	17	1.91	572	18	20.76	394	15	0.20	414	16	1.58	444	17	16.87	406	15	0.39	406
24	520	23	0.43	605	24	3.29	637	25	34.12	585	24	0.29	705	25	2.52	672	25	24.87	585	24	0.15	585
25	673	25	0.50	908	27	4.48	1016	28	43.80	684	24	0.35	879	26	2.79	1012	27	33.29	627	22	0.11	627
26	783	27	0.55	761	27	5.10	869	29	49.72	852	27	0.35	833	28	3.51	921	29	36.56	804	27	0.59	804
27	74	10	3.05	74	10	36.58	74	10	340.49	82	10	1.80	76	10	27.08	74	10	229.16	74	10	1.21	74
28	75	10	0.53	75	10	4.42	75	10	44.91	75	10	0.30	75	10	3.37	75	10	34.59	75	10	0.15	75
29	460	8	31.30	440	8	250.13	440	8	2062.48	446	8	8.05	440	8	83.46	440	8	977.67	360	7	4.98	360
30	590	35	1.62	590	35	26.93	628	37	162.12	587	32	0.87	630	34	13.74	655	37	143.21	502	29	0.42	502
31	225	15	29.25	213	15	196.28	213	15	1730.25	215	15	9.32	215	15	78.98	215	15	752.01	215	15	7.05	215

Table B.6: Virtual A* with cost parameters $\alpha = 1$, $\beta = 0$, and $\gamma = 1$

Id	CG10	RG10	TG10	CG100	RG100	TG100	CG1K	RG1K	TG1K	CW10	RW10	TW10	CW100	RW100	TW100	CW1K	RW1K	TW1K	CD10	RD10	TD10	CD100
1	168	3	0.15	168	3	0.50	168	3	6.80	168	3	0.05	168	3	0.39	168	3	3.68	168	3	0.06	168
2	163	5	0.21	163	5	0.92	163	5	9.24	163	5	0.06	163	5	0.70	163	5	7.07	163	5	0.20	163
3	209	5	0.20	209	5	0.66	209	5	7.31	191	4	0.09	209	5	0.47	209	5	5.21	196	4	0.13	196
4	219	5	0.33	219	5	0.77	219	5	9.21	219	5	0.05	219	5	0.56	219	5	5.49	219	5	0.18	219
5	424	5	0.40	424	5	2.53	424	5	23.58	424	5	0.19	424	5	2.10	424	5	19.20	424	5	0.17	424
6	209	5	0.22	209	5	0.48	209	5	4.59	209	5	0.04	209	5	0.32	209	5	3.33	209	5	0.06	209
7	184	5	0.19	184	5	0.47	184	5	4.70	184	5	0.04	184	5	0.44	184	5	3.31	184	5	0.19	184
8	169	5	0.20	169	5	0.49	169	5	4.97	169	5	0.04	169	5	0.36	169	5	3.48	169	5	0.07	169
9	181	6	0.22	179	6	0.72	179	6	6.81	181	6	0.05	179	6	0.48	179	6	4.73	191	6	0.11	191
10	371	8	0.41	371	8	0.85	371	8	8.23	381	8	0.06	371	8	0.53	371	8	5.75	413	8	0.15	413
11	559	10	0.81	557	10	5.12	557	10	53.04	569	10	0.50	557	10	4.14	557	10	42.15	589	10	0.32	589
12	617	7	0.57	737	9	3.75	736	9	40.28	698	8	0.26	659	8	2.61	709	9	27.94	720	9	0.35	720
13	1061	10	0.86	1019	10	5.67	1008	10	57.76	937	9	0.42	913	9	3.68	1019	10	40.74	1003	9	0.48	1003
14	953	10	6.17	953	10	55.09	953	10	536.82	993	10	3.94	962	10	43.93	953	10	378.65	1019	10	2.28	1019
15	1082	8	0.98	1062	8	10.64	1062	8	109.31	1000	7	3.88	1062	8	10.52	1062	8	82.66	1000	7	0.61	1000
16	1334	10	1.56	1320	10	15.51	1320	10	150.82	1320	10	0.93	1320	10	10.37	1320	10	100.79	1320	10	0.69	1320
17	1949	10	12.54	1949	10	163.86	1947	10	1577.97	1977	10	10.26	1947	10	117.41	1947	10	1269.34	1983	10	8.78	1983
18	1906	10	16.68	1890	10	199.44	1890	10	1781.31	2018	10	10.94	1890	10	120.21	1890	10	1251.24	1963	10	16.47	1963
19	2387	10	8.85	2353	10	83.84	2346	10	810.60	2378	10	4.10	2347	10	56.86	2347	10	571.12	2353	10	6.34	2353
20	2477	10	9.93	2471	10	97.96	2471	10	967.66	2495	10	6.91	2471	10	77.46	2471	10	744.56	2539	10	2.87	2539
21	1025	19	0.37	1055	20	2.91	1035	20	29.07	1055	19	0.25	1028	19	1.88	1033	20	21.57	984	17	0.45	984
22	933	17	0.29	971	18	2.15	921	18	21.12	933	16	0.18	917	17	1.41	951	18	14.20	915	17	0.53	915
23	1146	17	0.30	1236	18	2.48	1180	18	24.29	1195	17	0.23	1125	17	2.05	1222	18	19.58	1138	16	0.26	1138
24	1504	25	0.55	1467	25	5.16	1437	25	48.38	1537	25	0.45	1478	25	4.01	1467	25	36.68	1491	24	0.53	1491
25	1979	28	0.73	2077	29	7.45	2168	30	73.20	2138	29	0.87	2101	29	6.18	2044	29	55.33	2014	27	0.34	2014
26	2156	30	0.74	2155	30	6.84	2106	30	68.77	2208	30	0.43	2153	30	5.31	2118	30	49.95	2273	30	1.40	2273
27	224	10	8.66	222	10	76.30	222	10	750.35	228	10	4.02	226	10	57.71	222	10	523.47	232	10	2.60	232
28	287	10	0.84	287	10	8.87	287	10	88.51	297	10	0.99	287	10	7.66	287	10	74.68	297	10	0.35	297
29	1496	8	18.89	1414	8	178.44	1414	8	1838.44	1496	8	12.40	1414	8	127.25	1414	8	1331.27	1496	8	1.66	1496
30	1256	36	1.56	1292	38	16.40	1342	39	201.15	1287	36	1.32	1294	38	12.18	1322	39	214.51	1287	36	0.98	1287
31	593	15	74.93	573	15	663.24	567	15	6920.56	589	15	55.82	577	15	550.96	569	15	5306.05	589	15	28.36	589

Table B.7: Virtual A* with cost parameters $\alpha = 1$, $\beta = 1$, and $\gamma = 0$

Id	CG10	RG10	TG10	CG100	RG100	TG100	CG1K	RG1K	TG1K	CW10	RW10	TW10	CW100	RW100	TW100	CW1K	RW1K	TW1K	CD10	RD10	TD10	CD10
1	168	3	0.15	168	3	0.47	168	3	7	168	3	0.05	168	3	0.48	168	3	4.72	168	3	0.07	16
2	163	5	0.25	163	5	0.84	163	5	12.06	163	5	0.06	163	5	0.6	163	5	7.95	163	5	0.21	16
3	209	5	0.22	209	5	0.71	209	5	7.83	191	4	0.06	209	5	0.52	209	5	6.1	196	4	0.13	19
4	195	5	0.25	195	5	0.84	195	5	8.95	195	5	0.06	195	5	0.59	195	5	7.35	195	5	0.20	19
5	410	5	0.45	404	5	3.02	404	5	29.63	406	5	0.31	404	5	2.36	404	5	24.15	406	5	0.19	40
6	209	5	0.18	209	5	0.48	209	5	4.54	209	5	0.04	209	5	0.34	209	5	3.77	209	5	0.07	20
7	184	5	0.22	184	5	0.47	184	5	4.62	184	5	0.04	184	5	0.34	184	5	3.81	184	5	0.12	18
8	169	5	0.22	169	5	0.49	169	5	4.89	169	5	0.04	169	5	0.38	169	5	3.95	169	5	0.06	16
9	181	6	0.28	179	6	0.72	179	6	6.59	181	6	0.05	179	6	0.48	179	6	5.58	191	6	0.11	19
10	371	8	0.33	371	8	0.84	371	8	8.41	381	8	0.06	371	8	0.54	371	8	6.52	413	8	0.12	41
11	557	10	1.32	553	10	8.76	553	10	78.37	585	10	0.57	557	10	6.77	557	10	61.5	607	10	0.82	60
12	609	7	0.6	732	9	4.04	732	9	40.74	631	7	0.29	670	8	2.63	655	8	30.45	717	9	0.31	71
13	921	9	1.23	1015	10	9.09	1009	10	86.88	901	9	0.73	1063	10	5.92	1013	10	59.5	1033	10	1.21	103
14	946	10	12.72	946	10	125.81	946	10	1302.3	985	10	6.71	962	10	92.51	946	10	873.5	967	10	2.84	96
15	1082	8	1.09	1062	8	12.48	1062	8	125.43	1015	7	4.01	1062	8	11.11	1062	8	90.65	1015	7	0.74	101
16	1307	10	2.51	1297	10	22.12	1293	10	213.62	1307	10	1.08	1307	10	13.68	1293	10	147.96	1307	10	0.54	130
17	1951	10	48.38	1947	10	904.31	1939	10	7303	1963	10	28.43	1947	10	461.06	1939	10	6169.93	1947	10	15.09	194
18	1856	10	19.35	1856	10	351.75	1856	10	2892.29	1954	10	15.69	1880	10	214.17	1856	10	2001.95	1990	10	3.17	199
19	2345	10	14.72	2339	10	132.23	2339	10	1207.41	2395	10	5.31	2339	10	83.02	2339	10	866.32	2353	10	7.38	235
20	2469	10	32.86	2465	10	316.66	2465	10	3082.64	2493	10	28.49	2465	10	274.08	2465	10	2560.76	2506	10	11.63	250
21	1023	19	0.39	1055	20	2.75	1033	20	31.48	1055	19	0.26	1027	19	1.98	1033	20	22.99	1004	19	1.00	100
22	933	17	0.27	971	18	2.13	921	18	23.58	933	16	0.18	917	17	1.43	951	18	16.28	915	17	0.46	91
23	1146	17	0.33	1226	18	2.31	1198	18	26.3	1195	17	0.24	1125	17	1.73	1174	18	21.19	1138	16	0.11	113
24	1504	25	0.56	1467	25	4.35	1430	25	52.16	1537	25	0.33	1478	25	4.04	1442	25	39.6	1491	24	0.48	149
25	1979	28	0.82	2077	29	7.22	2168	30	79.7	2134	29	0.87	2101	29	6.24	2044	29	62.12	2009	27	0.33	200
26	2156	30	0.74	2155	30	7.09	2106	30	73.16	2208	30	0.42	2153	30	5.15	2118	30	53.57	2208	30	1.07	220
27	222	10	10.24	222	10	100.14	222	10	1001.28	228	10	5.8	222	10	70.17	222	10	630.86	228	10	2.50	22
28	287	10	1.9	281	10	19.26	281	10	203.67	287	10	1.86	281	10	15.12	281	10	168.08	287	10	0.59	28
29	1374	8	27.38	1374	8	259.32	1374	8	2858.6	1374	8	21	1374	8	226.51	1374	8	2018.86	1434	8	9.06	143
30	1322	37	1.76	1292	38	19.01	1314	40	256.29	1287	36	1.41	1327	38	17.16	1304	39	266.46	1287	36	0.98	128
31	601	15	165.27	571	15	1436.83	565	15	17127.2	603	15	138.36	573	15	1601.28	565	15	13233.5	603	15	194.34	60

Table B.8: Virtual A* with cost parameters $\alpha = 1$, $\beta = 1$, and $\gamma = 1$

Id	CG10	RG10	TG10	CG100	RG100	TG100	CG1K	RG1K	TG1K	CW10	RW10	TW10	CW100	RW100	TW100	CW1K	RW1K	TW1K	CD10	RD10	TD10	CD
1	284	3	0.15	284	3	0.46	284	3	7.66	284	3	0.10	284	3	0.35	284	3	4.08	284	3	0.05	
2	283	5	0.21	283	5	0.96	283	5	9.32	283	5	0.13	283	5	0.62	283	5	6.91	283	5	0.22	
3	349	5	0.22	349	5	0.64	349	5	7.13	327	4	0.07	349	5	0.64	349	5	5.40	336	4	0.22	
4	355	5	0.24	355	5	0.81	355	5	8.75	375	5	0.07	355	5	0.75	355	5	6.52	375	5	0.08	
5	762	5	0.43	750	5	3.03	750	5	30.91	750	5	0.26	750	5	2.38	750	5	24.11	750	5	0.18	
6	327	5	0.18	327	5	0.55	327	5	5.13	327	5	0.04	327	5	0.40	327	5	3.56	327	5	0.07	
7	286	5	0.31	286	5	0.49	286	5	5.13	286	5	0.04	286	5	0.33	286	5	3.60	286	5	0.07	
8	271	5	0.24	271	5	0.47	271	5	5.13	271	5	0.04	271	5	0.33	271	5	3.65	271	5	0.06	
9	277	6	0.24	277	6	0.63	277	6	7.05	277	6	0.06	277	6	0.50	277	6	5.17	287	6	0.11	
10	589	8	0.32	589	8	0.80	589	8	8.73	591	8	0.06	589	8	0.52	589	8	6.14	657	8	0.13	
11	901	10	0.96	891	10	6.52	883	10	66.34	899	10	0.54	893	10	5.28	883	10	53.19	913	10	0.34	
12	1161	9	0.78	1151	9	4.94	1137	9	50.40	1069	8	0.42	1151	9	3.05	1147	9	34.00	1068	8	0.50	1
13	1528	9	1.06	1655	10	7.92	1655	10	81.87	1580	9	0.67	1744	10	5.09	1670	10	54.58	1580	9	0.33	1
14	1673	10	9.45	1664	10	85.91	1659	10	867.18	1702	10	6.89	1680	10	68.15	1659	10	588.84	1702	10	5.15	1
15	1902	8	1.57	1902	8	20.31	1902	8	203.40	2015	8	4.51	1902	8	16.06	1902	8	128.69	2015	8	0.85	2
16	2325	10	1.98	2311	10	21.53	2311	10	216.67	2311	10	1.26	2311	10	14.70	2311	10	145.54	2311	10	0.65	2
17	3441	10	26.60	3441	10	310.15	3441	10	3470.83	3489	10	26.37	3441	10	348.14	3441	10	2950.82	3511	10	15.00	3
18	3451	10	44.05	3384	10	373.48	3378	10	3573.01	3581	10	22.41	3384	10	271.89	3373	10	2775.78	3488	10	24.13	3
19	4229	10	17.05	4229	10	147.26	4193	10	1453.41	4419	10	5.99	4207	10	105.55	4193	10	971.74	4383	10	5.66	4
20	4562	10	32.73	4547	10	376.53	4534	10	3740.65	4617	10	38.52	4547	10	276.23	4534	10	3012.58	4639	10	8.79	4
21	1610	19	0.34	1627	20	3.17	1601	20	29.90	1627	19	0.22	1705	20	1.97	1617	20	21.60	1610	19	1.05	1
22	1461	17	0.26	1469	18	2.03	1456	18	22.01	1475	17	0.17	1557	18	1.33	1468	18	15.69	1452	17	0.63	1
23	1765	17	0.38	1962	18	2.65	1836	18	24.78	1895	17	0.24	1653	17	1.74	1920	18	20.08	1823	17	0.40	1
24	2389	25	0.65	2324	25	4.97	2218	25	47.55	2433	25	0.34	2322	25	4.03	2216	25	35.89	2444	25	0.60	2
25	3042	28	1.00	3346	29	7.76	3313	30	76.57	3288	28	0.72	3258	29	5.72	3152	29	58.65	3147	27	0.27	3
26	3348	30	0.64	3288	30	7.05	3224	30	70.12	3428	29	0.47	3250	30	5.18	3238	30	51.44	3505	27	0.13	3
27	384	10	13.05	372	10	125.20	370	10	1246.28	384	10	6.75	378	10	90.30	372	10	836.37	388	10	4.46	
28	499	10	2.22	493	10	18.53	491	10	191.64	497	10	1.95	493	10	14.70	491	10	164.07	497	10	0.70	
29	2410	8	24.55	2380	8	307.03	2380	8	3169.47	2482	8	21.98	2380	8	228.54	2380	8	2192.75	2482	8	9.22	2
30	1980	37	1.91	1978	38	27.40	1998	39	292.87	1964	35	1.56	2022	37	14.51	1966	39	266.67	2005	36	1.63	2
31	909	15	142.65	909	15	1505.31	897	15	18653.80	933	15	254.19	915	15	1386.76	903	15	13367.90	931	15	137.66	

References

[1] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223(2):295–303, 2012. ISSN 0377-2217.

[2] M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of mathematics*, pages 781–793, 2004.

[3] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1974. ISBN 0201000296.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms and Applications*. Prentice hall, Englewood Cliffs, N.J., 1993.

[5] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, and M. Reddy. Chapter 1 applications of network optimization. In *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science*, pages 1–83. Elsevier, 1995.

[6] Z. Al Chami, H. Manier, and M.-A. Manier. New model for a variant of pick up and delivery problem. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1708–1713, 10 2016.

[7] Z. Al Chami, H. Manier, and M. Manier. A lexicographic approach for the bi-objective selective pickup and delivery problem with time windows and paired demands. *Annals of Operations Research volume*, 85:237–255, 2017.

[8] Z. Al Chami, H. Manier, M.-A. Manier, and Z. Peng. A hybrid particle swarm optimization for the selective pickup and delivery problem with transfers. *Engineering Applications of Artificial Intelligence*, 85:99–111, 2019. ISSN 0952-1976.

[9] L. Alexander, S. Allen, N. Bindoff, F.-M. Breon, J. Church, U. Cubasch, S. Emori, P. Forster, P. Friedlingstein, N. Gillett, J. Gregory, D. Hartmann, E. Jansen, B. Kirtman, R. Knutti, K. Kanikicharla, P. Lemke, J. Marotzke, V. Masson-Delmotte, and S.-P. Xie. *Climate change 2013: The physical science basis, in contribution of Working Group I (WGI) to the Fifth Assessment Report (AR5) of the Intergovernmental Panel on Climate Change (IPCC)*. Cambridge University Press, 09 2013.

[10] M. Andini, Y. Satria, and H. Burhan. Dynamic pickup and delivery problem with transfer in ridesharing to reduce congestion. *Journal of Physics: Conference Series*, 1218:012010, 05 2019.

[11] S. Anily and J. Bramel. Approximation algorithms for the capacitated traveling salesman problem with pickups and deliveries. *Naval research logistics*, 46:654–670, 1999.

[12] S. Anily and R. Hassin. The swapping problem. *Networks*, 22(4):419–433, 1992.

[13] C. Archetti and M. G. Speranza. The split delivery vehicle routing problem: A survey. In B. Golden, S. Raghavan, and E. Wasil, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, pages 103–122. Springer US, Boston, MA, 2008. ISBN 978-0-387-77778-8.

[14] J. Aronson. A survey of dynamic network flows. *Annals of Operations Research*, 20:1–66, 12 1989.

[15] S. R. Arora, A. V. Hill, and B. Kalantari. An algorithm for the traveling salesman problem with pickup and delivery customers. *European Journal of Operational Research*, 22(3):377–386, 1985. ISSN 0377-2217.

[16] N. Ascheuer, M. Fischetti, and M. Grötschel. A polyhedral study of the asymmetric traveling salesman problem with time windows. *Networks*, 36(2):69–79, 2000.

[17] K. Ashok and M. E. Ben-Akiva. Estimation and prediction of time-dependent origin-destination flows with a stochastic mapping to path flows and link flows. *Transportation Science*, 36(2):184–198, 2002.

[18] E. Balas and M. W. Padberg. Set partitioning: A survey. *SIAM Review*, 18(4):710–760, 10 1976.

[19] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996. ISSN 0167-6377.

[20] D. Banister. "Great Cities and Their Traffic": Michael Thomson Revisited. *Built Environment (1978-)*, 41(3):435–446, 2015. ISSN 02637960.

[21] C. Barnhart, E. Johnson, G. Nemhauser, M. Savelsbergh, and P. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46(3):316–329, 1994.

[22] R. G. Bartle and D. R. Sherbert. *Introduction to Real Analysis*. Oxford University Press, 2011.

[23] M. Battarra, J.-F. Cordeau, and M. Iori. Pickup-and-delivery problems for goods transportation. In P. Toth and D. Vigo, editors, *Vehicle Routing. Problems, Methods and Applications*, chapter 6, pages 161–191. Society for Industrial and Applied Mathematics, 2nd edition, 2014.

[24] J. Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society*, 55(4):299—-327, 1959.

[25] S. Belieres, M. Hewitt, N. Jozefowiez, and F. Semet. A time-expanded network reduction matheuristic for the logistics service network design problem. *Transportation Research Part E: Logistics and Transportation Review*, 147:102203, 2021. ISSN 1366-5545.

[26] M. Benchimol, P. Benchimol, B. Chappert, A. Taille, F. Laroche, F. Meunier, and L. Robinet. Balancing the stations of a self service "bike hire" system. *RAIRO - Operations Research*, 45:37–61, 2011.

[27] G. Berbeglia, J.-F. Cordeau, I. Gribkovskaia, and G. Laporte. Static pickup and delivery problems: A classification scheme and survey. *TOP*, 15:1–31, 02 2007. ISSN 1863-8279.

[28] G. Berbeglia, J.-F. Cordeau, and G. Laporte. Dynamic pickup and delivery problems. *European Journal of Operational Research*, 202(1):8–15, 2010. ISSN 0377-2217.

[29] D. Bertsimas and J. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

[30] L. Bigras, M. Gamache, and G. Savard. Time-indexed formulations and the total weighted tardiness problem. *INFORMS Journal of Computing*, 20(1), 2008.

[31] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. Mixed-integer programming: A progress report. In M. Grötschel, editor, *The Sharpest Cut*, chapter 18, pages 309–325. Society for Industrial and Applied Mathematics, 2004.

[32] N. Boland, M. Hewitt, L. Marshall, and M. Savelsbergh. The continuous-time service network design problem. *Operations Research*, 65(5):1303–1321, 2017.

[33] N. Boland, M. Hewitt, D. M. Vu, and M. Savelsbergh. Solving the traveling salesman problem with time windows through dynamically generated time-expanded networks. In D. Salvagnin and M. Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, pages 254–262. Springer, 2017. ISBN 978-3-319-59776-8.

[34] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer, 2008.

[35] R. Borndörfer, M. Grötschel, F. Klostermeier, and C. Küttner. Telebus berlin: Vehicle scheduling in a dial-a-ride system. In N. H. M. Wilson, editor, *Computer-Aided Transit Scheduling*, pages 391–422. Springer, 1999. ISBN 978-3-642-85970-0.

[36] P. Bouros, T. Dalamagas, D. Sacharidis, and T. Sellis. Dynamic pickup and delivery with transfers. *Proceedings of the 12th International Symposium on Spatial and Temporal Databases*, 08 2011.

[37] D. Bredström and M. Rönnqvist. Combined vehicle routing and scheduling with temporal precedence and synchronization constraints. *European Journal of Operational Research*, 191 (1):19–31, 2008. ISSN 0377-2217.

[38] C. Brezovec, G. Cornuéjols, and F. W. Glover. A matroid algorithm and its application to the efficient solution of two optimization problems on graphs. *Mathematical Programming*, 42:471–487, 1988.

[39] S. Bsaybes. *Modèles et algorithmes de gestion de flottes de véhicules vipa*. PhD thesis, Université Clermont Auvergne, 2017.

[40] S. Bsaybes, A. Quilliot, and A. K. Wagler. Fleet management for autonomous vehicles using multicommodity coupled flows in time-expanded networks. In *SEA*, 2018.

[41] S. Bsaybes, A. Quilliot, and A. Wagler. Fleet management for autonomous vehicles using flows in time-expanded networks. *TOP, Springer Verlag*, 27(2):288–311, 2019.

[42] L. S. Buriol and C. S. Sartori. A study on the pickup and delivery problem with time windows: Matheuristics and new instances. *Computers & Operations Research*, 124:105065, 2020. ISSN 0305-0548.

[43] P. Chalasani and R. Motwani. Approximating capacitated routing and delivery problems. *SIAM Journal on Computing*, 28(6):2133–2149, 1999.

[44] D. Chemla, F. Meunier, and R. Wolfler-Calvo. Bike sharing systems: solving the static rebalancing problem. *Discrete Optimization*, 10(2):120–146, 2012.

[45] D. Chemla, F. Meunier, T. Pradeau, R. Wolfler-Calvo, and H. Yahiaoui. Self-service bike sharing systems: simulation, repositioning, pricing. *hal-00824078*, 2013.

[46] J. Chifflet, P. Mahey, and V. Reynier. Proximal decomposition for multicommodity flow problems with convex costs. *Telecommunication Systems*, 3:1–10, 1994.

[47] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. *Carnegie Mellon University*, 3:10, 03 1976.

[48] B. Coltin and M. Veloso. Scheduling for transfers in pickup and delivery problems with very large neighborhood search. *Proceedings of the National Conference on Artificial Intelligence*, 3:2250–2256, 06 2014.

[49] M. Conforti, G. Cornuejols, and G. Zambelli. *Integer Programming*. Springer, 2014. ISBN 3319110071.

[50] C. Contardo, C. Cortés, and M. Matamala. The pickup and delivery problem with transfers: Formulation and a branch-and-cut solution method. *European Journal of Operational Research*, 200:711–724, 02 2010.

[51] C. Contardo, C. Morency, and L.-M. Rousseau. Balancing a dynamic public bike-sharing system. Technical report, CIRRELT-2012-09, CIRRELT, Montreal, Canada, 2012, 2012.

[52] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[53] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Ltd, 1998.

[54] J.-F. Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573–586, 2006.

[55] J.-F. Cordeau and G. Laporte. The dial-a-ride problem (darp): Variants, modeling issues and algorithms. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):89–101, 2003.

[56] J.-F. Cordeau and G. Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*, 37(6):579–594, 2003. ISSN 0191-2615.

[57] J.-F. Cordeau and G. Laporte. The dial-a-ride problem: models and algorithms. *Annals of operations research*, 153(1):29–46, 2007.

[58] F. H. Cullen, J. J. Jarvis, and H. D. Ratliff. Set partitioning based heuristics for interactive routing. *Networks*, 11(2):125–143, 1981.

[59] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Manage. Sci.*, 6(1):80—-91, 10 1959. ISSN 0025-1909.

[60] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32:505–536, 1985.

[61] S. Deleplanque and A. Quilliot. Dial-a-ride problem with time windows, transshipments, and dynamic transfer points. *IFAC Proceedings Volumes*, 46(9):1256–1261, 2013. ISSN 1474-6670.

[62] M. Dell'Amico, E. Hadjicostantinou, M. Iori, and S. Novellani. The bike sharing rebalancing problem: Mathematical formulations and benchmark instances. *Omega*, 45:7–19, 2014.

[63] DEMOGRAPHIA. Wendell Cox Consultancy. Demographia world urban areas. 17th annual edition: 2021 06. Technical report, 2021. URL `www.demographia.com`.

[64] J.-C. Deschamps, R. Dupas, and R. T. Takoudjou. A mip formulation for the pickup and delivery problem with time window and transshipment. *IFAC Proceedings Volumes*, 45(6): 333–338, 2012. ISSN 1474-6670.

[65] J. Desrosiers, P. Pelletier, and F. Soumis. Plus court chemin avec contraintes d'horaires. *Rairo-operations Research*, 17:357–377, 1983.

[66] J. Desrosiers, F. Soumis, and M. Desrochers. Routing with time windows by column generation. *Networks*, 14(4):545–565, 1984.

[67] J. Desrosiers, Y. Dumas, and F. Soumis. A dynamic programming solution of the large-scale single-vehicle dial-a-ride problem with time windows. *American Journal of Mathematical and Management Sciences*, 6, 2 1986.

[68] L. Di Gaspero, A. Rendl, and T. Urli. A hybrid ACO+CP for balancing bicycle sharing systems. In M. J. Blesa, C. Blum, P. Festa, A. Roli, and M. Sampels, editors, *Hybrid Metaheuristics*, pages 198–212. Springer, 2013. ISBN 978-3-642-38516-2.

[69] R. Diestel. *Graph theory.* Springer, 2018.

[70] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 1959.

[71] K. Doerner and J. J. Salazar González. Pickup-and-delivery problems for people transportation. In P. Toth and D. Vigo, editors, *Vehicle Routing. Problems, Methods and Applications*, chapter 7, pages 193–212. Society for Industrial and Applied Mathematics, 2nd edition, 2014. ISBN 978-1-61197-358-7.

[72] A. Downs. The law of peak-hour expressway congestion. *Traffic quarterly*, 16, 1962.

[73] M. Dror, D. Fortin, and C. Roucairol. Redistribution of self-service electric cars: A case of pickup and delivery. 01 1998.

[74] G. Dueck and T. Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90(1):161–175, 1990. ISSN 0021-9991.

[75] Y. Dumas, J. Desrosiers, and F. Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54(1):7–22, 1991. ISSN 0377-2217.

[76] J. D. Durand. Historical estimates of world population: An evaluation. *Population and Development Review*, 3(3):253–296, 1977. ISSN 00987921, 17284457.

[77] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995.

[78] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[79] J. Edmonds. Submodular functions, matroids, and certain polyhedra. In *Combinatorial Structures and Their Applications, Proceedings of Calgary International Conference*, pages 69–87, 1970.

[80] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association for Computing Machinery*, 19(2):248–264, 4 1972. ISSN 0004-5411.

[81] G. Erdoğan, M. Battarra, and R. Wolfler-Calvo. An exact algorithm for the static rebalancing problem arising in bicycle sharing systems. *European Journal of Operational Research*, 245 (3), 2015.

[82] K. Fagerholt and M. Christiansen. A combined ship scheduling and allocation problem. *Journal of the Operational Research Society*, 51(7):834–842, 2000.

[83] J. Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902(124):1–27, 1902.

[84] E. Ferguson. The rise and fall of the american carpool: 1970–1990. *Transportation*, 24: 349–376, 11 1997.

[85] J.-L. Figueroa, A. Quilliot, H. Toussaint, and A. Wagler. Optimal 1-request insertion for the pickup and delivery problem with transfers and time horizon. In *ICORES*, volume 1, pages 17–27. INSTICC, SciTePress, 2022. ISBN 978-989-758-548-7.

[86] J.-L. Figueroa, A. Quilliot, H. Toussaint, and A. Wagler. Optimal paths with impact on a constraint system: an application to the 1-request insertion for the pickup and delivery problem with transfers. In G. H. Parlier, F. Liberatore, and M. Demange, editors, *Operations Research and Enterprise Systems*, volume 1. Springer, 2022.

[87] J. L. Figueroa González, M. Baïou, A. Quilliot, H. Toussaint, and A. Wagler. Branch-and-cut for a 2-commodity flow relocation model with time constraints. In I. Ljubić, F. Barahona, S. S. Dey, and A. R. Mahjoub, editors, *Combinatorial Optimization*, pages 22–34, Cham, 2022. Springer International Publishing.

[88] J. L. Figueroa González, A. Quilliot, H. Toussaint, and A. Wagler. Managing time expanded networks through project and lift: the lift issue. *Procedia Computer Science*, 223: 241–249, 2023. ISSN 1877-0509. XII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS 2023).

[89] J. L. Figueroa González, A. Quilliot, H. Toussaint, and A. Wagler. Managing time expanded networks: The strong lift problem. In A. Brieden, S. Pickl, and M. Siegle, editors, *Graphs and Combinatorial Optimization: from Theory to Applications: CTW2023 Proceedings*, Cham, 2024. Springer International Publishing. ISBN 978-3-031-46826-1.

[90] F. Fischer and C. Helmberg. Dynamic graph generation for the shortest path problem in time expanded networks. *Mathematical Programming*, 143(1-2):257–297, 2014.

[91] L. Fleischer and M. Skutella. The quickest multicommodity flow problem. In W. J. Cook and A. S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 36–53. Springer, 2002. ISBN 978-3-540-47867-6.

[92] L. Fleischer and M. Skutella. Quickest flows over time. *SIAM Journal on Computing*, 36(6): 1600–1630, 2007.

[93] L. R. Ford and D. R. Fulkerson. *Maximal Flow through a Network.* RAND Corporation, 1954.

[94] L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6(3):419–433, 1958.

[95] L. R. Ford and F. D. R. *Flows in Networks.* Princeton University Press, 1962. ISBN 9780691651842.

[96] I. A. Forma, T. Raviv, and M. Tzur. A 3-step math heuristic for the static repositioning problem in bike-sharing systems. *Transportation Research Part B: Methodological*, 71:230–247, 2015. ISSN 0191-2615.

[97] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1), 1986.

[98] G. G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964. ISSN 0030364X, 15265463. URL `http://www.jstor.org/stable/167703`.

[99] T. Gallai. Maximum-minimum sätze über graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 9(3):395–434, 12 1958.

[100] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic Discrete Methods*, 1(2):216–227, 1980.

[101] D. Gavalas, C. Konstantopoulos, and G. Pantziou. Design and management of vehicle-sharing systems: A survey of algorithmic approaches. *Smart Cities and Homes*, pages 216–289, 2016.

[102] M. Gendreau, G. Laporte, and D. Vigo. Heuristics for the traveling salesman problem with pickup and delivery. *Computers & Operations Research*, 26(7):699–714, 1999. ISSN 0305-0548.

[103] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[104] P. C. Gilmore and R. E. Gomory. Sequencing a one state-variable machine: A solvable case of the traveling salesman problem. *Operations Research*, 12(5):655–679, 1964. ISSN 0030364X, 15265463.

[105] F. Gnegel and A. Fügenschuh. Branch-and-refine for solving time-expanded milp formulations. *Computers & Operations Research*, 149:106043, 2023. ISSN 0305-0548.

[106] M. Godinho, L. Gouveia, and P. Pesneau. Natural and extended formulations for the time-dependent traveling salesman problem. *Discrete Applied Mathematics*, 164(1):138–153, 2014.

[107] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 09 1958.

[108] Google. *Google Maps*. `https://maps.google.com`, 2023. Accessed on Feb 7, 2023.

[109] L. Gouveia, M. Leitner, and M. Ruthmair. Layered graph approaches for combinatorial optimization problems. *Computers & Operations Research*, 102:22–38, 2019. ISSN 0305-0548.

[110] M. Groß and M. Skutella. Generalized maximum flows over time. In R. Solis-Oba and G. Persiano, editors, *Approximation and Online Algorithms*, pages 247–260. Springer, 2012. ISBN 978-3-642-29116-6.

[111] S. I. Grossman. *Elementary Linear Algebra*. Brooks Cole, 2004. ISBN 0030973546.

[112] A. Hall, S. Hippler, and M. Skutella. Multicommodity flows over time: Efficient algorithms and complexity. *Theoretical Computer Science*, 379(3):387–404, 2007. ISSN 0304-3975. Automata, Languages and Programming.

[113] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10 (1):26–30, 1935.

[114] S. Hammouda, M. Taffar, and A. Lemouari. A simulated annealing for the resolution of "Dial-A-Ride-Problem with Transfer" using hybrid neighborhood methods. In *ICECOCS*, pages 1–6, 2020.

[115] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2): 100–107, 1968.

[116] P. Healy and R. Moll. A new extension of local search applied to the dial-a-ride problem. *European Journal of Operational Research*, 83(1):83–104, 1995. ISSN 0377-2217.

[117] H. Hernández-Pérez and J.-J. Salazar-González. The one-commodity pickup-and-delivery travelling salesman problem. In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *Combinatorial Optimization — Eureka, You Shrink!: Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001 Revised Papers*, pages 89–104. Springer, 2003. ISBN 978-3-540-36478-8.

[118] H. Hernández-Pérez and J.-J. Salazar-González. A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery. *Discrete Applied Mathematics*, 145(1):126–139, 2004. ISSN 0166-218X. Graph Optimization IV.

[119] H. Hernández-Pérez and J. J. Salazar González. Heuristics for the one-commodity pickup-and-delivery traveling salesman problem. *Transportation Science*, 38:245–255, 05 2004.

[120] R. J. Herrera and R. Garcia-Bertrand. Origin of modern humans. In *Ancestral DNA, Human Origins, and Migrations*, chapter 3, pages 61–103. Academic Press, 2018. ISBN 978-0-12-804124-6.

[121] H. Hersbach, B. Bell, P. Berrisford, S. Hirahara, A. Horányi, J. Muñoz-Sabater, J. Nicolas, C. Peubey, R. Radu, D. Schepers, A. Simmons, C. Soci, S. Abdalla, X. Abellan, G. Balsamo, P. Bechtold, G. Biavati, J. Bidlot, M. Bonavita, G. De Chiara, P. Dahlgren, D. Dee, M. Diamantakis, R. Dragani, J. Flemming, R. Forbes, M. Fuentes, A. Geer, L. Haimberger, S. Healy, R. J. Hogan, E. Hólm, M. Janisková, S. Keeley, P. Laloyaux, P. Lopez, C. Lupu, G. Radnoti, P. de Rosnay, I. Rozum, F. Vamborg, S. Villaume, and J.-N. Thépaut. The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society*, 146(730):1999–2049, 2020.

[122] S. C. Ho, Y.-H. Kuo, J. M. Leung, M. Petering, W. Szeto, and T. W. Tou. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological*, 111:395–421, 2018. ISSN 0191-2615.

[123] A. Hooper. Cost of congestion to the trucking industry: 2018 update. Technical report, American Transportation Research Institute, October 2018.

[124] Y. Hou, W. Zhong, L. Su, K. Hulme, A. Sadek, and C. Qiao. Taset: Improving the efficiency of electric taxis with transfer-allowed rideshare. *IEEE Transactions on Vehicular Technology*, 65:9518–9528, 12 2016.

[125] J.-J. Hublin, A. Ben-Ncer, S. E. Bailey, S. E. Freidline, S. Neubauer, M. M. Skinner, I. Bergmann, A. Le Cabec, S. Benazzi, K. Harvati, and P. Gunz. New fossils from jebel irhoud, morocco and the pan-african origin of homo sapiens. *Nature*, 546:289—-292, 2017.

[126] D. Huremović. *Brief History of Pandemics (Pandemics Throughout History)*, pages 7–35. Springer, 2019. ISBN 978-3-030-15346-5.

[127] INRIX Research. *Global Traffic Scorecard 2019*. Technical report, December 2019. URL `https://inrix.com/scorecard`.

[128] A. Insel, J. B. Fraleigh, and L. Spence. *Elementary Linear Algebra. A Matrix Approach.* Pearson Education, 2014.

[129] I. Ioachim, J. Desrosiers, Y. Dumas, M. M. Solomon, and D. Villeneuve. A request clustering algorithm for door-to-door handicapped transportation. *Transportation Science*, 29(1):63–78, 1995.

[130] S. Irnich, P. Toth, and D. Vigo. *The Family of Vehicle Routing Problems*, chapter 1, pages 1–33. 11 2014. ISBN 978-1-61197-358-7.

[131] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B: Methodological*, 20(3):243–257, 1986. ISSN 0191-2615.

[132] D. S. Johnson and L. A. McGeoch. *Experimental Analysis of Heuristics for the STSP*, pages 369–443. Springer, 2007. ISBN 978-0-306-48213-7.

[133] E. L. Johnson. Modeling and strong linear programs for mixed integer programming. In S. W. Wallace, editor, *Algorithms and Model Formulations in Mathematical Programming*, pages 1–43. Springer, 1989. ISBN 978-3-642-83724-1.

[134] D. E. Joslin and D. P. Clements. "Squeaky Wheel" Optimization. *J. Artif. Int. Res.*, 10(1): 353–373, 5 1999. ISSN 1076-9757.

[135] M. Jünger, G. Reinelt, and G. Rinaldi. *Combinatorial Optimization - Eureka, You Shrink!, Papers Dedicated to Jack Edmonds, 5th International Workshop, Aussois, France, March 5-9, 2001, Revised Papers*. Springer, 1 2003.

[136] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[137] L. G. Khachiyan. A polynomial algorithm in linear programming. In *Doklady Academii Nauk SSSR*, volume 244, pages 1093–1096, 1979.

[138] S. Kikuchi. Scheduling of demand-responsive transit vehicles. *Journal of Transportation Engineering*, 110(6):511–520, 1984.

[139] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–80, 06 1983.

[140] D. E. Knuth, T. Larrabee, and P. M. Roberts. *Mathematical Writing*, volume 14 of *MAA notes*. Mathematical Association of America, 1989. ISBN 978-0-88385-063-3.

[141] S. Kobayashi, Y. Ota, Y. Harada, A. Ebita, M. Moriya, H. Onoda, K. Onogi, K. Hirotaka, C. Kobayashi, H. Endo, K. Miyaoka, and K. Takahashi. The jra-55 reanalysis: General specifications and basic characteristics. *Journal of the Meteorological Society of Japan. Ser. II*, 93(1):5–48, 2015.

[142] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 6th edition, 2019. ISBN 9783662585665.

[143] E. F. Krause. *Taxicab Geometry*. Dover Publications, 2012.

[144] S. O. Krumke, A. Quilliot, A. K. Wagler, and J.-T. Wegener. Relocation in carsharing systems using flows in time-expanded networks. In J. Gudmundsson and J. Katajainen, editors, *Experimental Algorithms*, pages 87–98. Springer, 2014. ISBN 978-3-319-07959-2.

[145] M. Kubo and H. Kasugai. Heuristic algorithms for the single vehicle dial-a-ride problem. *Journal of the Operations Research Society of Japan*, 33(4):354–365, 1990.

[146] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. ISSN 00129682, 14680262.

[147] N. J. L. Lenssen, G. A. Schmidt, J. E. Hansen, M. J. Menne, A. Persin, R. Ruedy, and D. Zyss. Improvements in the gistemp uncertainty model. *Journal of Geophysical Research: Atmospheres*, 124(12):6307–6326, 2019.

[148] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3): 115–116, 1973.

[149] A. Lim, F. Wang, and Z. Xu. The capacitated traveling salesman problem with pickups and deliveries on a tree. In X. Deng and D.-Z. Du, editors, *Algorithms and Computation*, pages 1061–1070. Springer, 2005. ISBN 978-3-540-32426-3.

[150] F. Lokin. Procedures for travelling salesman problems with additional constraints. *European Journal of Operational Research*, 3(2):135–141, 1979. ISSN 0377-2217.

[151] L. Lozano and A. L. Medaglia. On an exact method for the constrained shortest path problem. *Computers & Operations Research*, 40(1):378–384, 2013. ISSN 0305-0548.

[152] C. Machado, N. Hue, F. Berssaneti, and J. Quintanilha. An overview of shared mobility. *Sustainability*, 10:4342, 11 2018.

[153] O. B. G. Madsen, H. F. Ravn, and J. M. Rygaard. A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives. *Annals of Operations Research*, 60:193–208, 1995.

[154] R. Masson, F. Lehuédé, and O. Péton. A tabu search algorithm for the dial-a-ride problem with transfers. *Proceedings of the IESM 2011 Conference*, 05 2011.

[155] R. Masson, F. Lehuédé, and O. Péton. Simple temporal problems in route scheduling for the dial–a–ride problem with transfers. In N. Beldiceanu, N. Jussien, and É. Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 275–291. Springer, 2012. ISBN 978-3-642-29828-8.

[156] R. Masson, F. Lehuédé, and O. Péton. Efficient feasibility testing for request insertion in the pickup and delivery problem with transfers. *Operations Research Letters*, 41(3):211–215, 2013. ISSN 0167-6377.

[157] R. Masson, F. Lehuédé, and O. Péton. An adaptive large neighborhood search for the pickup and delivery problem with transfers. *Transportation Science*, 47:344–355, 8 2013.

[158] R. Masson, F. Lehuédé, and O. Péton. The dial-a-ride problem with transfers. *Computers & Operations Research*, 41:12–23, 2014. ISSN 0305-0548.

[159] M. Minoux. Networks synthesis and optimum network design problems: Models, solution methods and applications. *Networks*, 19(3):313–360, 1989.

[160] S. Mitrović-Minić and G. Laporte. The pickup and delivery problem with time windows and transshipment. *INFOR: Information Systems and Operational Research*, 44(3):217–227, 2006.

[161] Y. Molenbruch, K. Braekers, and A. Caris. Typology and literature review for dial-a-ride problems. *Annals of Operations Research*, 259, 12 2017.

[162] C. P. Morice, J. J. Kennedy, N. A. Rayner, J. P. Winn, E. Hogan, R. E. Killick, R. J. H. Dunn, T. J. Osborn, P. D. Jones, and I. R. Simpson. An updated assessment of near-surface temperature change from 1850: The hadcrut5 data set. *Journal of Geophysical Research: Atmospheres*, 126(3), 2021.

[163] G. Mosheiov. The travelling salesman problem with pick-up and delivery. *European Journal of Operational Research*, 79(2):299–310, 1994. ISSN 0377-2217.

[164] Y. Nakao and H. Nagamochi. Worst case analysis for pickup and delivery problems with consecutive pickups and deliveries. In Y. Dong, D.-Z. Du, and O. Ibarra, editors, *Algorithms and Computation*, pages 554–563. Springer, 2009. ISBN 978-3-642-10631-6.

[165] National Aeronautics and Space Administration. Goddard Institute for Space Studies. World of change: Global temperatures. `https://earthobservatory.nasa.gov/world-of-change/global-temperatures`, 2022. Accessed on Nov 02, 2022.

[166] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[167] K. Nordback, S. Kothuri, T. Phillips, C. Gorecki, and M. Figliozzi. Accuracy of bicycle counting with pneumatic tubes in oregon. *Transportation Research Record*, 2593:8–17, 2016.

[168] M. Nourinejad and M. Roorda. A dynamic carsharing decision support system. *Transportation Research Part E: Logistics and Transportation Review*, 66:36–50, 2014.

[169] J. G. Oxley. Briefly, what is a matroid? 2018.

[170] S. Parragh, K. Doerner, and R. Hartl. A survey on pickup and delivery problems: Part ii: Transportation between pickup and delivery locations. *Journal für Betriebswirtschaft*, 58: 81–117, 06 2008.

[171] A. G. Percus and O. C. Martin. Finite size and dimensional dependence in the euclidean traveling salesman problem. *Phys. Rev. Lett.*, 76:1188–1191, Feb 1996.

[172] J. Pierotti and J. Theresia van Essen. Milp models for the dial-a-ride problem with transfers. *EURO Journal on Transportation and Logistics*, 10:100037, 2021. ISSN 2192-4376.

[173] J.-Y. Potvin and J.-M. Rousseau. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, 66 (3):331–340, 1993. ISSN 0377-2217.

[174] W. B. Powell, P. Jaillet, and A. Odoni. Chapter 3 stochastic and dynamic networks and routing. In *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, pages 141–295. Elsevier, 1995.

[175] Préfecture de la région Auvergne. Contrat de Plan État-Région Auvergne 2015-2020. `https://www.prefectures-regions.gouv.fr/auvergne-rhone-alpes/content/download/14008/97561/file/CPER_2015-2020_VERSION_signee.pdf`, 2015. Accessed on Nov 17, 2022.

[176] Préfecture de la région Auvergne-Rhône-Alpes. Projet de Contrat de Plan État-Région Auvergne-Rhône-Alpes 2021-2027. `https://www.prefectures-regions.gouv.fr/auvergne-rhone-alpes/content/download/89323/573572/file/01_%20Projet%20de%20CPER_ARA_%202021_2027.pdf`, 2021. Accessed on Nov 17, 2022.

[177] Préfecture de la région Rhône-Alpes. Contrat de Plan État-Région Auvergne 2015-2020. `https://www.rhone.gouv.fr/content/download/18856/105395/file/0-CPER_RA_2015_2020_mai2015.pdf`, 2015. Accessed on Nov 17, 2022.

[178] H. Psaraftis. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*, 14:130–154, 5 1980.

[179] H. Psaraftis. An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows. *Transportation Science*, 17(3):351–357, 1983.

[180] A. Quilliot, F. Bendali, and J. Mailfert. Flots entiers et multiflots fractionnaires couplés par une contrainte de capacité. *RAIRO Operations Research*, 39:185–224, 07 2005.

[181] M. Rainer-Harbach, P. Papazek, B. Hu, and G. R. Raidl. Balancing bicycle sharing systems: A variable neighborhood search approach. In M. Middendorf and C. Blum, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 121–132. Springer, 2013. ISBN 978-3-642-37198-1.

[182] T. Raviv, M. Tzur, and I. A. Forma. Static repositioning in a bike-sharing system: models and solution approaches. *EURO Journal on Transportation and Logistics*, 2(3):313–360, 2013.

[183] K. K. Rebibo. A computer controlled dial-a-ride system. In *Traffic control and transportation systems, Proceedings of 2nf IFAC/IFIP/IFORS Symposium Monte Carlo, September 1974*, pages 69–87. North-Holland, 9 1974.

[184] L. B. Reinhardt, T. Clausen, and D. Pisinger. Synchronized dial-a-ride transportation of disabled passengers at airports. *European Journal of Operational Research*, 225(1):106–117, 2013. ISSN 0377-2217.

[185] R. A. Rohde and Z. Hausfather. The berkeley earth land/ocean temperature record. *Earth System Science Data*, 12(4):3469–3479, 2020. doi: 10.5194/essd-12-3469-2020. URL `https://essd.copernicus.org/articles/12/3469/2020/`.

[186] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40:455–472, 11 2006.

[187] S. Ropke, J.-F. Cordeau, and G. Laporte. Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks: An International Journal*, 49(4):258–272, 2007.

[188] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Series in Artificial Inteligence, 4th edition, 2021. ISBN 780134610993.

[189] M. W. P. Savelsbergh and M. M. Sol. The general pickup and delivery problem. *Transportation Science*, 29(1):17–29, 1995.

[190] R. Schuijbroek, J. adn Hampshire and W. van Hoeve. Inventory rebalancing and vehicle routing in bike sharing systems. *European Journal of Operational Research*, 257(3), 2017.

[191] J. Schönberger. Scheduling constraints in dial-a-ride problems with transfers: a metaheuristic approach incorporating a cross-route scheduling procedure with postponement opportunities. *Public Transport*, 9, 07 2017.

[192] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011. ISBN 978-0-321-57351-3.

[193] T. R. Sexton and L. D. Bodin. Optimizing Single Vehicle Many-to-Many Operations with Desired Delivery Times: I. Scheduling. *Transportation Science*, 19(4):378–410, 1985. ISSN 00411655, 15265447.

[194] T. R. Sexton and L. D. Bodin. Optimizing Single Vehicle Many-to-Many Operations with Desired Delivery Times: II. Routing. *Transportation Science*, 19(4):411–435, 1985.

[195] S. Shaheen, S. Guzman, and H. Zhang. Bikesharing in europe, the americas, and asia: Past, present, and future. *Institute of Transportation Studies, Working Paper Series, UC Davis*, 2143, 2010.

[196] S. Shaheen, A. Cohen, I. Zohdy, and B. Kock. Shared Mobility: Current Practices and Guiding Principles Brief. Technical report, Institute of Transportation Studies, UC Berkeley, 4 2016.

[197] J. S. Shang and C. K. Cuff. Multicriteria pickup and delivery problem with transfer opportunity. *Computers & Industrial Engineering*, 30(4):631–645, 1996. ISSN 0360-8352.

[198] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35:254–265, 1987.

[199] D. M. Stein. An asymptotic, probabilistic analysis of a routing problem. *Mathematics of Operations Research*, 3(2):89–101, 1978. ISSN 0364765X, 15265471.

[200] D. M. Stein. Scheduling dial-a-ride transportation systems. *Transportation Science*, 12(3): 232–249, 1978. ISSN 00411655, 15265447.

[201] S. Thangiah, A. Fergany, and S. Awan. Real-time split-delivery pickup and delivery time window problems with transfers. *Central European Journal of Operations Research*, 15: 329—349, 2007.

[202] The R Foundation. *The R Project for Statistical Computing*. `https://www.r-project.org/`, 2019. Accessed on Dec 28, 2021.

[203] J. M. Thomson. Great cities and their traffic. 1977.

[204] F. A. Tillman and T. M. Cain. An upperbound algorithm for the single and multiple terminal delivery problem. *Management Science*, 18(11):664–682, 1972. ISSN 00251909, 15265501.

[205] P. Toth and D. Vigo. *Vehicle Routing: Problems, Methods, and Applications, Second Edition*. Society for Industrial and Applied Mathematics, 2014. ISBN 1611973589.

[206] United Nations. *The World's Cities in 2018*. Technical report, Department of Economic and Social Affairs, Population Division, 2019.

[207] United Nations. *World Population Prospects 2019:Highlights (ST/ESA/SER.A/423)*. Technical report, Department of Economic and Social Affairs, Population Division, 2019.

[208] United Nations. *World Urbanization Prospects: The 2018 Revision (ST/ESA/SER.A/420)*. Technical report, Department of Economic and Social Affairs, Population Division, 2019.

[209] United Nations. *Download Center*. `https://population.un.org/wpp/Download/Standard/Population/`, 2019. Accessed on Dec 28, 2021.

[210] United Nations. *WMO Provisional State of the Global Climate 2022*. Technical report, World Meteorological Organization, 2022.

[211] U.S. Census Bureau. 2021 american community survey 1-year estimates. `https://data.census.gov/cedsci/table?q=carpool&tid=ACSDT1Y2021.B08301`, 2021. Accessed on November 16, 2022.

[212] J. Van der Akker, C. Hurkens, and M. Savelsberg. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal of Computing*, 12(2):111–124, 2000.

[213] J. Viegas. Making urban road pricing acceptable and effective: searching for quality and equity in urban mobility. *Transport Policy*, 8(4):289–294, 2001. ISSN 0967-070X.

[214] D. M. Vu, M. Hewitt, N. Boland, and M. Savelsbergh. Dynamic discretization discovery for solving the time-dependent traveling salesman problem with time windows. *Transportation Science*, 54(3):703–720, 2020.

[215] Z. Wang and J.-B. Sheu. Vehicle routing problem with drones. *Transportation Research Part B: Methodological*, 122:350–364, 2019. ISSN 0191-2615.

[216] A. Waserhole and V. Jost. Vehicle Sharing System Pricing Regulation: A Fluid Approximation. working paper or preprint, 2012. URL `https://hal.science/hal-00727041`.

[217] H. Williams. *Model Building in Mathematical Programming*. Wiley, 2013. ISBN 9781118506189.

[218] H. Wilson, S. J., W. H., and H. B. *Scheduling algorithms for dial-a-ride system*. Tech report. Massachusetts Institute of Technology, Urban Systems Laboratory, 1971.

[219] N. Wilson, R. Weissberg, and J. Hauser. *Advanced Dial-a-ride Algorithms Research Project: Final Report*. Tech report. Massachusetts Institute of Technology, Department of Materials Science and Engineering, 1967.

[220] G. Xue. A two-stage heuristic solution for multi-depot collaborative pickup and delivery network with transfers to reduce carbon emissions. *Journal of Cleaner Production*, 373: 133839, 2022. ISSN 0959-6526.

[221] C. Yu, O. O'Brien, P. DeMaio, R. Rabello, S. Chou, and T. Benicchio. The Meddin Bike-sharing World Map Mid-2021 Report. Technical report, PBSC Urban Solutions, 10 2021.

[222] D. Zhang, C. Yu, J. Desai, H. Lau, and S. Srivathsan. A time-space network flow approach to dynamic repositioning in bicycle sharing systems. *Transportation Research Part B: Methodological*, 103:188–207, 2017. ISSN 0191-2615. Green Urban Transportation.

[223] H.-M. Zhang, H. B., L. J., M. M., and S. T. M. Noaa global surface temperature dataset (noaaglobaltemp), version 5.0. doi: doi:10.25921/9qth-2p70.

[224] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *J Optim Theory Appl*, 45:41–51, 1985.

Glossary

Black Death The Black Death was the most fatal pandemic recorded in human history. It was a bubonic plague pandemic occurring in Western Eurasia and North Africa from 1346 to 1353 . 4

clean hydrogen Also known as Green hydrogen, is hydrogen generated by powering the electrolysis of water with renewable energy or with low-carbon power. It has significantly lower carbon emissions than "grey hydrogen", which is primarily produced by steam reforming of natural gas . 8

electrolysis A technique that uses direct electric current to drive an otherwise non-spontaneous chemical reaction . 8

greenhouse gases Usually abbreviated GHG or GhG. Gases that absorb and emit radiant energy within the thermal infrared range. The primary greenhouse gases in Earth's atmosphere are water vapor (H_2O), carbon dioxide (CO_2), methane (CH_4), nitrous oxide (N_2O), and ozone (O_3). 3

Homo sapiens Homo sapiens (humans) are the most abundant and widespread species of primate. Although some scientists use the term "humans" with all members of the genus Homo, in common usage it generally refers to Homo sapiens, the only non-extinct member. 4

Industrial Revolution The Industrial Revolution was the transition to new manufacturing processes that started in Great Britain, continental Europe, and the United States. It occurred during the period from around 1760 to about 1820–1840 . 4

La Niña An oceanic and atmospheric phenomenon. During a La Niña period, the sea surface temperature across the eastern equatorial part of the central Pacific Ocean will be lower than normal by 3–5 °C (5.4–9 °F). An appearance of La Niña often persists for longer than five months. 3

megacity An urban agglomeration with more than 10 million inhabitants. 5

Index

Résumé en français

Introduction et contexte

Au cours des dernières décennies, une augmentation des températures a été observée dans le monde entier [29] et il a été estimé que plus de 50% de l'augmentation observée est très probablement due à l'augmentation des concentrations des *gaz à effet de serre* (GES) anthropiques dans l'atmosphère [1].

La principale source de ces émissions de GES anthropiques est la combustion de combustibles fossiles (comme le charbon, le pétrole et le gaz naturel), et est fortement influencée par certains phénomènes démographiques tels que les taux accélérés de croissance démographique et d'urbanisation. Ces problématiques sont généralement très complexes et entretiennent des relations avec d'autres activités humaines comme, par exemple, le transport.

Les problèmes de transport urbain ne sont pas particulièrement récents, en 1977 Michael Thomson [28] a publié une étude sur le trafic dans les grandes villes et a inventé l'expression « *plus la ville est grande, plus les problèmes sont importants et plus les coûts de transport sont élevés* ». De nos jours, les problèmes de transport persistent et sont devenus plus préoccupants.

En France, les *Contrats de plan État-Région* (CPER) sont des plans régionaux pour programmer et financer les grands projets de développement, tels que la création d'infrastructures et le soutien de secteurs clés pour l'avenir. Les CPER de la région Auvergne-Rhône-Alpes [23], [25] et [24] intègrent le financement d'opérations liées à l'écologie et à la transition énergétique.

En matière d'énergies renouvelables, l'un des objectifs du CPER-Auvergne-Rhône-Alpes 2021-2027 est de faire passer le recours aux énergies alternatives de 19% à 36% d'ici 2030. Aussi, ce CPER considère que l'hydrogène est un technologie clé de la transition écologique, et pour mener à bien le développement des technologies de l'hydrogène, le CPER se fixe les défis suivants.

1. La transition de l'industrie vers l'hydrogène propre.
2. Mobilité, infrastructures de distribution et véhicules (spécialement pour les poids lourds).
3. L'énergie hydrogène, la production par électrolyse à un prix compétitif, les infrastructures de stockage et de transport, les applications stationnaires, les services réseaux et marchands.

Cette thèse contribue à relever le troisième défi ci-dessus en fournissant des modèles pour résoudre les problèmes de tournées des véhicules avec des transferts et un horizon temporel, en renforçant la prise de décision informée sur la gestion des réseaux de transport.

Les transferts peuvent contribuer à diminuer le nombre de véhicules en circulation car généralement on peut satisfaire plus de demandes de transport avec le même nombre de véhicules ; les transferts peuvent également contribuer à économiser du carburant en réduisant la distance parcourue. Cependant, une meilleure gestion des contraintes de temps est nécessaire pour garantir qu'un transfert aura lieu.

D'autre part, en imposant un horizon de temps, nous pouvons concevoir des plans de transport à exécuter dans une période de temps particulière où des ressources peuvent être disponibles.

Dans cette thèse, deux problèmes de collecte et de livraison avec des transferts et un horizon temporel sont analysés et de nouvelles méthodes et algorithmes sont proposés pour traiter ces problèmes.

Problème de repositionnement d'objets avec transferts et horizon de temps

Le premier problème abordé est un *Problème de repositionnement d'objets avec transferts et horizon de temps*. Dans un premier temps, il est proposé un modèle multiflots sur un réseau étendu dans le temps. Ce modèle est significatif mais il est très difficile à résoudre dans la pratique. Pour cette raison, nous avons proposé une approche *Projection et Remontage* pour le gérer de manière flexible.

Projection. Le modèle multiflots est « *projeté* » sur le réseau de transport d'origine pour obtenir un modèle multiflots plus simple que nous avons appelé le *Modèle projété de repositionnement d'objets* (PIRP). Pour récupérer une partie des contraintes temporelles, nous avons introduit les *contraintes étendus de sous-tours* qui lient l'horizon temporel et le nombre de véhicules circulant à travers tout sous-ensemble de sommets non-dépôts. Nous avons montré comment ces contraintes peuvent être séparées en temps polynomial, et nous avons observé l'efficacité de l'algorithme de *séparation et coupe* ainsi que la qualité des solutions calculées.

Remontage. Les problèmes de *remontage* consistent à construire une solution au problème de repositionnement d'objets à partir d'une solution du modèle projeté. Nous avons introduit deux problèmes de *remontage* : le problème du *Remontage Fort* et le problème du *Remontage Faible*, qui se distinguent par leur degré de compatibilité avec une solution PIRP. De ces problèmes, nous avons déduit les contraintes

de *décomposition en chemins réalisables* qui peuvent être séparées et ajoutés pour renforcer le modèle projeté. Cette fois, le problème de séparation est NP-difficile, mais ces contraintes peuvent être séparées en pratique par génération de colonnes avec un effort de calcul modéré (complexité pseudo-polynomial).

Pour le problème du *Remontage Forte*, nous avons proposé un *programme linéaire mixte en nombres entiers* exacte. Cependant, nous avons observé que la plupart du temps, ce programme est non réalisable. D'autre part, nous avons proposé la méthode *Covering/Weak* pour gérer le problème de *Remontage Faible* de manière flexible. Nous avons introduit le concept de *remontage-faible-consistance* et nous avons conjecturé que, sous certains conditions, les modèles *Covering/Weak* peuvent donner des solutions optimales lorsque nous partons de solutions PIRP qui sont *remontage-faible-consistantes*.

La bibliographie qui a été plus relevante pour cette partie de la thèse est la suivante : Ford and Fulkerson [16], Hu [18], Aronson [2], Powell et al. [22], Fleischer and Skutella [15], Benchimol et al. [3], Chemla et al. [6], Rainer-Harbach et al. [26], Krumke et al. [19], Bsaybes et al. [5], and Gouveia et al. [17].

Les résultats de cette partie de la thèse ont été présentés au *7th International Symposium on Combinatorial Optimization* (ISCO 2022), au *24ème Congrès Annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision* (ROADEF 2023), au *19th Cologne-Twente Workshop on Graphs and Combinatorial Optimization* (CTW 2023) et au *XII Latin-American Algorithms, Graphs and Optimization Symposium* (LAGOS 2023). Certaines parties de ce travail ont été publiées dans *Lecture Notes in Computer Science* [12] (Springer), *Graphs and Combinatorial Optimization : from Theory to Applications* [14] (AIRO Springer Series 13), et *Procedia Computer Science* [13] (Elsevier).

Comme travaux futurs, il reste à identifier d'autres problèmes où les contraintes étendus de sous-tour ou la borne inférieure fournie par le coût d'une solution PIRP optimale peuvent être appliquées. De plus, il serait intéressant de trouver de nouveaux types de contraintes pour renforcer le modèle projeté et augmenter la probabilité d'obtenir des solutions donnant des problèmes du *Remontage Fort* réalisables. En outre, l'approche proposée peut être utilisée pour d'autres problèmes impliquant des multiflots sur des réseaux étendus dans le temps.

Problème de tournées de véhicules avec collecte et livraison, transferts et horizon de temps

Le deuxième problème abordé est le *1-Request Insertion PDPT* qui se présente comme un sous-problème du *Problème de tournées de véhicules avec collecte et livraison, transferts et horizon de temps* (PDPT). Nous avons montré que ce problème peut être vu comme un cas particulier d'une problématique plus générale que nous avons appelée le « *problème du chemin virtuel* » et qui consiste à rechercher un chemin optimal parmi une collection de chemins qui satisfont et impactent une système de contraintes donnée. Nous avons proposé l'algorithme Virtual A*, qui est un algorithme de type A* pour résoudre le problème de chemin virtuel, et nous avons montré comment l'adapter pour résoudre le 1-Request Insertion PDPT. Puisque la complexité de cet algorithme est exponentielle, nous avons montré comment le modifier pour obtenir des heuristiques avec une complexité polynomiale.

Nous avons testé l'algorithme Virtual A* de manière intensive sur un ensemble de 300 instances aléatoires, et nous avons analysé son comportement. Nous avons confirmé que la plupart du temps, le nombre de transferts dans une solution optimale est petit et que les temps d'exécution présentent une forte variance. Nous avons également confirmé comment les temps d'exécution sont réduits lorsque nous limitons le nombre de transferts à une petite constante (par exemple deux ou trois). Nous avons également observé l'impact sur la qualité de la solution lorsque nous filtrons les arcs de transfert en fonction d'un paramètre de seuil de poids.

Aussi, nous avons montré comment l'algorithme Virtual A* peut être combiné avec des heuristiques de recherche classiques pour obtenir des algorithmes d'insertion pour gérer le PDPT. Nous avons implémenté ces algorithmes et analysé leur comportement. En conséquence, nous disposons d'un solveur capable de gérer des instances PDPT de taille petite/moyenne.

La bibliographie qui a été plus relevante pour cette partie de la thèse est la suivante : Shang and Cuff [27], Laporte and Mitrović-Minić [20], [7], Deschamps et al. [9], Bouros et al. [4], Lehuédé et al. [21], Chemla et al. [6], and Deleplanque and Quilliot [8].

Les résultats de cette partie de la thèse ont été présentés pour la première fois au *22ème Congrès Annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision* (ROADEF 2021), et au *11th International Conference on Operations Research and Enterprise Systems* (ICORES 2022). Deux articles ont été publiés : l'un dans les *ICORES 2022 proceedings* [10] et l'autre dans *Operations Research and Enterprise Systems* (Springer Communications in Computer and Information Science series)[11].

En tant que futures lignes de recherche, il reste à identifier d'autres problèmes où l'algorithme Virtual A * peut être appliqué. Il reste également à analyser le cas des fortes contraintes de synchronisation qui interviennent lorsque des véhicules doivent se croiser dans une fenêtre temporelle donnée pour effectuer un transfert. Aussi, il serait intéressant d'analyser le comportement des algorithmes de recherche proposés lorsqu'ils sont pourvus d'un composant (e.g., liste tabou) pour pénaliser la réinsertion de requêtes difficiles/non réalisables.

Commentaires finaux

Dans ce travail nous nous sommes limités aux problèmes génériques. Cependant, les modèles et algorithmes que nous avons décrits peuvent être appliqués dans des contextes pratiques. Par conséquent, il reste également à identifier le type d'applications concrètes où les modèles et algorithmes proposés peuvent être appliqués.

Bibliographie

[1] L. Alexander, S. Allen, N. Bindoff, F.-M. Breon, J. Church, U. Cubasch, S. Emori, P. Forster, P. Friedlingstein, N. Gillett, J. Gregory, D. Hartmann, E. Jansen, B. Kirtman, R. Knutti, K. Kanikicharla, P. Lemke, J. Marotzke, V. Masson-Delmotte, and S.-P. Xie. *Climate change 2013 : The physical science basis, in contribution of Working Group I (WGI) to the Fifth Assessment Report (AR5) of the Intergovernmental Panel on Climate Change (IPCC)*. Cambridge University Press, 09 2013.

[2] J. Aronson. A survey of dynamic network flows. *Annals of Operations Research*, 20 :1–66, 12 1989.

[3] M. Benchimol, P. Benchimol, B. Chappert, A. Taille, F. Laroche, F. Meunier, and L. Robinet. Balancing the stations of a self service "bike hire" system. *RAIRO - Operations Research*, 45 :37–61, 2011.

[4] P. Bouros, T. Dalamagas, D. Sacharidis, and T. Sellis. Dynamic pickup and delivery with transfers. *Proceedings of the 12th International Symposium on Spatial and Temporal Databases*, 08 2011.

[5] S. Bsaybes, A. Quilliot, and A. Wagler. Fleet management for autonomous vehicles using flows in time-expanded networks. *TOP, Springer Verlag*, 27(2) : 288–311, 2019.

[6] D. Chemla, F. Meunier, and R. Wolfler-Calvo. Bike sharing systems : solving the static rebalancing problem. *Discrete Optimization*, 10(2) :120–146, 2012.

[7] C. Contardo, C. Morency, and L.-M. Rousseau. Balancing a dynamic public bike-sharing system. Technical report, CIRRELT-2012-09, CIRRELT, Montreal, Canada, 2012, 2012.

[8] S. Deleplanque and A. Quilliot. Dial-a-ride problem with time windows, transshipments, and dynamic transfer points. *IFAC Proceedings Volumes*, 46(9) : 1256–1261, 2013. ISSN 1474-6670.

[9] J.-C. Deschamps, R. Dupas, and R. T. Takoudjou. A mip formulation for the pickup and delivery problem with time window and transshipment. *IFAC Proceedings Volumes*, 45(6) :333–338, 2012. ISSN 1474-6670.

[10] J.-L. Figueroa, A. Quilliot, H. Toussaint, and A. Wagler. Optimal 1-request insertion for the pickup and delivery problem with transfers and time horizon. In *ICORES*, volume 1, pages 17–27. INSTICC, SciTePress, 2022. ISBN 978-989-758-548-7.

[11] J.-L. Figueroa, A. Quilliot, H. Toussaint, and A. Wagler. Optimal paths with impact on a constraint system : an application to the 1-request insertion for the pickup and delivery problem with transfers. In G. H. Parlier, F. Liberatore, and M. Demange, editors, *Operations Research and Enterprise Systems*, volume 1. Springer, 2022.

[12] J. L. Figueroa González, M. Baïou, A. Quilliot, H. Toussaint, and A. Wagler. Branch-and-cut for a 2-commodity flow relocation model with time constraints. In I. Ljubić, F. Barahona, S. S. Dey, and A. R. Mahjoub, editors, *Combinatorial Optimization*, pages 22–34, Cham, 2022. Springer International Publishing.

[13] J. L. Figueroa González, A. Quilliot, H. Toussaint, and A. Wagler. Managing time expanded networks through project and lift : the lift issue. *Procedia Computer Science*, 223 :241–249, 2023. ISSN 1877-0509. XII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS 2023).

[14] J. L. Figueroa González, A. Quilliot, H. Toussaint, and A. Wagler. Managing time expanded networks : The strong lift problem. In A. Brieden, S. Pickl, and M. Siegle, editors, *Graphs and Combinatorial Optimization : from Theory to Applications : CTW2023 Proceedings*, Cham, 2024. Springer International Publishing. ISBN 978-3-031-46826-1.

[15] L. Fleischer and M. Skutella. The quickest multicommodity flow problem. In W. J. Cook and A. S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 36–53. Springer, 2002. ISBN 978-3-540-47867-6.

[16] L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6(3) :419–433, 1958.

[17] L. Gouveia, M. Leitner, and M. Ruthmair. Layered graph approaches for combinatorial optimization problems. *Computers & Operations Research*, 102 :22–38, 2019. ISSN 0305-0548.

[18] T. C. Hu. Multi-commodity network flows. *Operations Research*, 11(3) :344–360, 1963.

[19] S. O. Krumke, A. Quilliot, A. K. Wagler, and J.-T. Wegener. Relocation in carsharing systems using flows in time-expanded networks. In J. Gudmundsson and J. Katajainen, editors, *Experimental Algorithms*, pages 87–98. Springer, 2014. ISBN 978-3-319-07959-2.

[20] G. Laporte and S. Mitrović-Minić. The pickup and delivery problem with time windows and transshipment. *INFOR : Information Systems and Operational Research*, 44(3) :217–227, 2006.

[21] F. Lehuédé, R. Masson, and O. Péton. Efficient feasibility testing for request insertion in the pickup and delivery problem with transfers. *Operations Research Letters*, 41(3) :211–215, 2013. ISSN 0167-6377.

[22] W. B. Powell, P. Jaillet, and A. Odoni. Chapter 3 stochastic and dynamic networks and routing. In *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, pages 141–295. Elsevier, 1995.

[23] Préfecture de la région Auvergne. Contrat de Plan État-Région Auvergne 2015-2020. `https://www.prefectures-regions.gouv.fr/auvergne-rhone-alpes/content/download/14008/97561/file/CPER_2015-2020_VERSION_signee.pdf`, 2015. Accessed on Nov 17, 2022.

[24] Préfecture de la région Auvergne-Rhône-Alpes. Projet de Contrat de Plan État-Région Auvergne-Rhône-Alpes 2021-2027. `https://www.prefectures-regions.gouv.fr/auvergne-rhone-alpes/content/download/89323/573572/file/01_%20Projet%20de%20CPER_ARA_%202021_2027.pdf`, 2021. Accessed on Nov 17, 2022.

[25] Préfecture de la région Rhône-Alpes. Contrat de Plan État-Région Auvergne 2015-2020. `https://www.rhone.gouv.fr/content/download/18856/105395/file/0-CPER_RA_2015_2020_mai2015.pdf`, 2015. Accessed on Nov 17, 2022.

[26] M. Rainer-Harbach, P. Papazek, B. Hu, and G. R. Raidl. Balancing bicycle sharing systems : A variable neighborhood search approach. In M. Middendorf and C. Blum, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 121–132. Springer, 2013. ISBN 978-3-642-37198-1.

[27] J. S. Shang and C. K. Cuff. Multicriteria pickup and delivery problem with transfer opportunity. *Computers & Industrial Engineering*, 30(4) :631–645, 1996. ISSN 0360-8352.

[28] J. M. Thomson. Great cities and their traffic. 1977.

[29] United Nations. *WMO Provisional State of the Global Climate 2022*. Technical report, World Meteorological Organization, 2022.