POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Matematica

Tesi di Laurea Magistrale

Modelling relocation strategies for shared mobility system management

Relatori
prof. Luca Vassio
prof. Danilo Giordano
prof. Marco Mellia

Giulio Cerruto

Anno Accademico 2021-2022

Summary

A vehicle sharing system is based on a pool of vehicles that are located across the city and can be used by multiple people. Each implementation has its own characteristics, e.g., if it is station -based or free-floating, if it comprises different kinds of vehicles, if it has a per-minute fare or a flat rate.

Often the demand for these services varies significantly in space and time. Some stations/regions may run out of units due to uneven demand, while concurrent end journeys may overload others. To meet user trip demand and enhance user satisfaction and usage, a relocation procedure is a solution to rebalance vehicles in various stations.

This thesis focuses on free-floating car sharing systems employing an operator-based relocation process to mitigate the imbalanced vehicle distribution problem. Many previous works have already confronted the repositioning problem, optimising the relocation process only based on knowledge about the immediate future. The methods developed in this thesis aim at maximising the satisfied mobility demand avoiding a greedy and possibly short-sighted approach and taking into account demand forecasts over a longer time period and over multiple relocation steps. The objective is to identify, at each time frame, how many cars to move from one zone to another in order to maximise a given dynamic forecast demand, with constraints on the maximum number of cars to be moved and the time needed for vehicles to be relocated.

In particular the thesis analyses different optimisation methods for improving the relocation strategy and investigates how far in the future is it appropriate to look. A first method is based on the dynamic programming approach, which is - in principle - able to find the optimal solution but, unfortunately very little scalable and inappropriate to fit the problem dimensionality. A second procedure to find an approximate yet performing solution is based on the formulation of a linear optimisation program in three versions: a linear program, a mixed integer linear program and a combination of the two, gradually relaxing more and more integrality constraints for computational time purposes. Finally, starting from the solutions obtained through such algorithms, a further neighbourhood search step, including the simulated annealing strategy, may be performed to improve the quality of the solution.

Considering the city of Turin as a case study and using mobility data coming from the operating Car2go fleet, the performances of the developed strategies have been evaluated, taking into account satisfied trips, costs, and computational effort needed. A comparison with a greedy approach optimising over only a single time frame and with a no-relocation based strategy is then carried out. The approximated solution is also compared with the optimal one found through the dynamic programming approach only in a toy example. Finally, the impact of the relocation costs on the profits and the maximum relocation are investigated.

Results show that a long-sighted strategy is preferred over a greedy one, with performance improvements of up to 10% more satisfied demand, with a 2 two-hour time frame

look-ahead period already improving performances by around 7%. A relocation strategy enhances the system performance in the first place, although high relocation costs may result in a reduction in profits and user satisfaction, which may be mitigated with an increase in fares. Moreover, a MILP-based approach proves to be the best performing one, although computational times become prohibitive for long look ahead periods, while the LP-based approach gives good performances always in short times. Finally, the neighbourhood search does not always turn out to improve performances while needing great computational effort.

Acknowledgements

Chi mi conosce bene sa quanto io sia introverso ma allo stesso tempo poco introspettivo, e quanto sia difficile per me tirar fuori i sentimenti e le parole giuste per esprimere la mia gratitudine verso le persone a cui voglio bene. Questi ringraziamenti rappresentano, quindi, un enorme sforzo emotivo da parte mia.

Un primo grande ringraziamento va sicuramente ai miei relatori, i professori Luca Vassio, Danilo Giordano e Marco Mellia per avermi accompagnato, guidato e aiutato con passione e pazienza nella realizzazione di questa tesi.

Quest'ultimo anno mi ha sicuramente messo a dura prova sotto molti aspetti, forse come mai mi era successo in precedenza, e reagire non è sempre stato semplice per me. Però - sembra banale dirlo - ho avuto la fortuna di avere al mio fianco persone che in un modo o nell'altro sono riuscite ad aiutarmi e starmi vicino.

Il ringraziamento più grande va ai miei genitori, semplicemente per esserci sempre stati. L'affetto da parte di un genitore è spesso una cosa scontata, ma io sono stato abbastanza fortunato da non aver mai per un secondo nella vita pensato di non poter contare sul vostro appoggio, sulla vostra comprensione e sul vostro sostegno in ogni mia decisione. Grazie per avermi reso la persona che sono oggi, per starmi vicino e accettare quei momenti in cui sono più scontroso e ho meno voglia di parlare, grazie per celebrare ogni mio ritorno a casa come un evento della massima importanza, grazie per avermi cresciuto insegnandomi valori come l'umiltà, l'educazione e l'onestà e come non dimenticarli, anche quando capita di non ritrovarli in chi ti sta di fronte. Semplicemente grazie, per tutta la mia vita.

Se l'affetto da parte della propria famiglia è una cosa che spesso ci viene assicurata, ritrovarsi al proprio fianco qualcuno che quella vicinanza te la garantisca senza alcun obbligo è un privilegio forse più grande. Io ho avuto la fortuna di trovare tutto questo in te, Orazio. Grazie per avermi regalato alcuni dei momenti più belli della mia vita e per essermi sempre accanto quando ne ho bisogno. I ringraziamenti in una tesi sono, come al solito, la fiera della banalità, ma sappiamo benissimo entrambi quanto tu mi sia sempre stato vicino e mi abbia sostenuto, mettendomi spesso al primo posto, e ciò è tutt'altro che banale. Succeda quel che succeda, sai bene che potrai sempre contare su di me.

Oltre alle ben note sofferenze e ansie, l'università mi ha permesso di conoscere, tra un funzionale e l'altro, Simona e Giulia. Con voi si è costruita negli anni un'amicizia che mai mi sarei aspettato di poter vivere e che si è rafforzata nonostante la prova della lontananza durante i vari coprifuochi e lockdown legati al Covid. Grazie per essermi state ad ascoltare in tutte le mie paranoie, apprensioni e tormenti interiori vari, soprattutto in quest'ultimo anno. So di essere pesante, ma voi siete state abbastanza pazienti da aver

retto alla pressione. Grazie Giulia per la tua generosità, il tuo altruismo e soprattutto la tua dolcezza, sia emotiva che culinaria. Grazie Simona per la tua genuinità e allegria, e per aver cambiato idea su di me, nonostante la tua prima impressione non fosse esattamente positiva.

Questi ultimi anni mi hanno poi permesso di conoscere tante altre belle persone, con cui ho condiviso intere giornate durante i periodi di zona rossa e che continuano ad essere vicino a me anche adesso che stiamo uscendo da questa pandemia. Ad una in particolare devo tanto in questi ultimi mesi. Grazie Carlotta per essermi stata vicino nei vari drammi ed avermi ascoltato e consigliato (anche se poi non ti davo retta). Sarà forse perché siamo caratterialmente molto simili, ma con te mi sono sempre sentito compreso e appoggiato, in momenti in cui mi ci voleva davvero.

Infine, volevo ringraziare tutti coloro che sono stati al mio fianco in questi anni, per più o meno tempo, e con cui ho condiviso le mie giornate.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and motivation

This first chapter serves as an introduction to the present thesis. A description of the framework is given in section 1.1, discussing shared mobility systems and why relocation strategies need to be elaborated. In section 1.2, the research questions faced in this work are presented. The contribution of this work is described in section 1.3, together with a summary of the achieved results. Finally, an overview on the structure of the thesis is given in section 1.4.

1.1 Scenario and Framework

Cities are adopting innovative mobility methods to address smart city challenges such as carbon reduction, multimodal urban transportation, and traffic decongestion, with a focus on shared modes such as car and bike-sharing systems. These sharing models are being used by an increasing number of cities worldwide in order to combat the current trends of increased urban mobility, air pollution, and changes in urban mobility patterns and behaviour, issues which have been exacerbated by the recent pandemic crisis. As reported by Albuquerque et al. [2021], more than 1000 bike-sharing programmes have been operating since 2016 in 60 different nations, and these programmes have undergone several changes. The most recent technologies enable real-time data collection via wireless connections and sensors, producing massive amounts of data.

The notion of "Mobility as a Service" (Maas) has perhaps been most widely implemented through bike-sharing programmes (BSS). BSS debuted in Amsterdam as a public service in 1965 with the "White Bike" initiative. However, it was not until the last ten years that they gained international acclaim, mainly thanks to the increased awareness regarding environmental issues and the newly developed technologies, such as smartphones, allowing users to easily rent a vehicle and pay, without any direct contact with the operator. Indeed, as Vallez et al. [2021] reported, the number of BSS implementations greatly expanded from 13 BSS implementations in 2004 to 1956 active local programmes and around 15,254,400 shared bicycles in 2019. Besides, according to Chinese Ministry of Transport, by 2017 the number of dockless bikes and their users have reached 16 million and 130 million, respectively.

To gain access to the system, new users often sign up via a website or mobile application and may need to pay a subscription fee. Following that, users who have already registered with the system can locate, possibly reserve, and unlock the available vehicles using a mobile application. They can then pay by the amount of time they use the vehicle (for example, a per-minute fee), occasionally in addition to a drop-off fee based on the area of the city where the vehicle is returned.

Therefore, a vehicle sharing system is based on a pool of publicly available vehicles (either cars, bikes, motor scooters, scooters) that are located across the city and ready to be rented for a nominal cost. Each implementation has its own characteristics, such as:

- whether it is managed by a private company or by the government;
- whether it is dock-based (if it is based on stations picking areas) or free-floating (if vehicles can be picked up and dropped anywhere in the city);
- whether it has a restricted area of use (a delimited perimeter) or does not;
- whether it uses various payment methods, incentives to use the service, discounts, and pricing plans;
- the type of vehicles employed, which can be electric scooters, scooters, bicycles, cars (with different types of engines).

Each station in a station-based sharing system offers car/bike slots where users can pick up a vehicle or return one they have previously used. Maintaining a high degree of service efficiency despite the system's cyclical dynamic reconfiguration throughout the operation period is one of vehicle sharing systems' key challenges. For instance, during the morning rush hour, most people ride their bicycles to work. Due to the extreme oversupply of shared bikes, residential areas have extremely few bikes, which reduces possible future demand. Subway stations and business districts are thus paralysed. Due to users' parking sites being unrestricted, this issue is even worse with dockless sharing systems. Cities may experience serious issues as a result of this imbalance, as well as customers and service providers.

To guarantee that there are enough units that are distributed geographically among the stations to meet customer needs, regular relocation of vehicles between stations becomes essential. Some stations may run out of units due to uneven demand, while others get completely filled and may be overloaded by concurrent end journeys. More recently, free-floating systems allow to pick up and drop off the vehicles at virtually any parking space within a predetermined operational region. However, free-floating systems face essentially the same issue, with the only difference that no overloading scenarios need to be avoided, since it is unlike that a city zone gets completely filled with cars. The problem has been identified in the literature under numerous names, the most common ones being **imbalance**, **rebalance**, **repositioning**, and **balancing**.

To meet user trip demand and enhance user satisfaction and usage, a support staff taking care of a relocation procedure is commonly employed to rebalance vehicles in various stations. This strategy is distinguished by a fleet of vehicles and a staff devoted to manually moving vehicles between various places. However, this method is frequently

used at specific times of the day, making it impossible to change the number of bikes in real time. Due to the heavy usage of truck routing, this strategy comes at a high cost and has an environmentally unfriendly operation mode. Alternatively to or together with this operator-based strategy, subsidies may be granted to customers who agree to pick up their vehicle from a station with excess units and return it to a station with an inadequate number of cars/bikes, encouraging repositioning actions that are good for the balance of the system.

This thesis focuses on free-floating car sharing systems employing an operator-based relocation process to mitigate the imbalanced vehicle distribution problem. However, the methods developed in the work can be suitable for any kind of vehicle employed in the system and for both free-floating and station based systems, since in the latter no spatial discretization needs to be carried out.

1.2 Research questions

The present work arises from a few research questions:

- Can we optimise relocation, i.e., perform a relocation that would increase satisfied demand or revenues or profit?
- Is it worth to look ahead in the future avoiding a greedy approach? If so, how far should we go?
- Can we optimise the process in real time? What is the best trade off between performances and computational times?

The relocation process is a very complex procedure and can be carried out in many different ways. Finding the most effective one is an open research question and its main difficulty lies in the very big dimensionality of the problem. Indeed, real-world scenarios involve smart mobility implementations in big cities and therefore, a great number of stations and vehicles. All their possible configurations give life to a huge and hardly tractable problem and confronting it usually requires resorting to metaheuristics resulting in approximate solutions. Moreover, how advantageous is it to relocate vehicles, in the first place? Since it is a process that comes with a cost, the relocation operation might neutralise the additional incomes coming from more satisfied mobility demand.

Implementing a relocation strategy is widely considered a good way to improve the shared mobility system performances. However, previous works have often worked out methods optimising the relocation process only based on immediate or short-term information about the demand patterns. The main aim of this work is to go beyond these greedy approaches, gathering information about longer term demand patterns and optimising the process in a long-sighted way, avoiding choices that look best immediately, but turn out to compromise the performances in the long run. Besides, how far in the future is it worth to look? Looking ahead is usually a good strategy, and the further we look, the better it is. Unfortunately, increasing the amount of information taken into consideration often comes at a cost: the computational effort needed for the problem to be solved may

become overwhelming. Additionally, looking too far in the future may not even prove to give advantage, since the effect of the current decision may gradually fade.

Finally, vehicle sharing systems involve dynamic scenarios continually evolving and changing. As a consequence, it is necessary to take new decisions and adjust the old ones on the go. Therefore, fast algorithms are needed in these frameworks, gathering new information and rapidly elaborating new strategies to be carried out to improve the performances. Unfortunately, not all algorithms are able to provide solutions quickly: actually, the most performing ones are usually the slowest. It is therefore an important challenge to work out a method setting an appropriate trade off between performances and computational times.

1.3 My contribution and results

Some algorithms have been developed in the present work to identify which city zones are overcrowded and need fewer cars than their actual availability and can provide cars to city zones which, on the contrary, cannot satisfy the forecast demand and are in need of additional vehicles.

The dynamic programming methodology is a first way to identify the best relocation strategy. Theoretically, using this method, it is possible to examine indefinitely ahead in the future and identify the best possible solution to the issue at all time frames. It is feasible to associate an optimum choice to every potential system state, reacting to the system evolution and making adjustments to the decision as it goes along, constantly deciding what to do to get the best result. Since every conceivable state of the system must be analysed, this technique is unfortunately not particularly scalable and quickly becomes unworkable when dealing with real-world situations.

Therefore, in order to develop a relocation strategy in a fair amount of time with limited resources, approximation methodologies must be used. Consequently, a sequential algorithm based on the formulation of a linear program is resorted to. This method can be declined in one of three ways:

- a Linear Program, with no integrality constraints. Since the resulting relocation strategy is coded by continuous variables, a post-processing step follows to get an integer valued relocation vector;
- a Mixed Integer Linear Program, with integrality constraints on the relocation variables;
- a combination of the two, where the LP is first solved and the relocation variables whose optimal value is found to be 0 are fixed, leaving the remaining ones as the integer constrained variables for the MILP.

Even with a very high dimensionality and a somewhat large number of constraints, a linear program may frequently be solved rather instantly, but a post-processing of the solution is required in order to round its entries. On the other hand, there is no need for rounding steps when employing a mixed integer linear program since keeping the integrality constraints on the relocation variables produces an already feasible solution. However, it is

well known that mixed integer linear programs are exceedingly computationally costly. The third method's goal is to solve the MILP, obtaining an integer solution only after lowering the complexity and dimensionality of the optimisation problem. To further enhance the effectiveness of the method, a neighbourhood search might be started from the solution provided by the linear program. Since the optimisation problem and the neighbourhood search are carried out at the start of each time frame to change the relocation strategy based on the realisation of the trip demand and, therefore, of the system state, the algorithm in issue is described as sequential.

First of all, this research has once again proven that operators lose a great deal of consumer satisfaction if they don't move their cars to suit the shifting demand patterns. The efficiency of the system is increased by around 4.5% by moving just 1 percent of the fleet, while user satisfaction is increased by about 12% when the percentage of relocated cars is increased to 7.5%. However, too many transferred vehicles appear to reduce customer happiness since there are too many cars that are unavailable for booking by users because they are being relocated.

Another question examined is how far it is worthwhile to look ahead in the future, given that the further we look, the larger the solution becomes in terms of dimensionality, necessitating an increasing amount of time for the algorithms to solve the issue. When the look ahead period is set to two two-hour time frames and fewer than three percent of the total cars are moved by the operator, the results indicated that a long-sighted approach did, in fact, raise customer happiness, with an efficiency increase of about 7%. When the look ahead period reaches 10 two-hour time frames, the efficiency improvement can reach up to 10%, but computational times get overwhelming with such look aheads.

The mixed integer linear program optimising across a 2 two-hour look ahead horizon in essentially no time ensures an 82 percent system efficiency, and turns out to be the algorithm that offers the optimum trade-off between performances and computing times. Despite the fact that some of the strategies took way too long to be executed, all of them prove to be effective in tackling the issue. Implementing the neighbourhood search starting from the solution provided by the optimisation program, in particular, can enhance the quality of the solution, but the additional time required to do so prevents it from being an effective substitute for other approaches. Working with a toy example also demonstrates that the created algorithms, which are only able to identify approximate solutions, may often come up with almost optimum solutions, by using a dynamic programming solution as a benchmark. The latter option is the best one, but the algorithm that finds it quickly becomes computationally impractical since the state space grows nearly exponentially as more cars and city zones are added to the system.

A simple analysis of the system profits and relocation costs reveals that greater fees ensure higher profits and that relocation expenses have a roughly linear influence on profits. Aside from situations with extremely high per-minute fees, the system efficiency tends to decline as relocation costs rise. Finally, the higher the relocation costs, the less vehicles are relocated, since the additional cost of moving a vehicle does not get rewarded by a sufficient profit, with higher fee cases less sensitive to an increase in costs.

1.4 Overview of the thesis

Chapter 2 provides a review of the scientific literature consulted to gaze the state of the art of the confronted problem and gather new information to solve it. Chapter 3 gives an overview on the formalisation of the problem and the implemented algorithms to work it out. Following, chapter 4 provides details about the dynamic programming method and how it has been implemented in the work. Chapter 5 gives insights on the implementation and resolution of the different versions of the linear programs and chapter 6 details the neighbourhood search step with the simulated annealing strategy. Appendix A shows the tuning procedure followed to set the parameters for the simulated annealing strategy implementation.

Chapter 7 shows the data extraction procedure. Real-world data coming from the operating Car2go fleet in the city of Turin were used and fed to the algorithms. Both the spatial and temporal discretisation procedures are presented and a little data exploration is carried out. Finally, chapter 8 displays and comments the obtained results from running the algorithms. Chapter 9 closes the thesis and wraps up the whole work.

The code realised for this thesis can be found at `https://github.com/giuliocerruto/shared-mobility-relocation`.

Chapter 2

Literature Review

This chapter focuses on the review of the existing literature concerning the fleet management problem in shared mobility systems and some optimisation methods employed in this work. A quick and broad overview is initially provided, followed by a review of a very widespread approach to the relocation problem: the Vehicle Routing Problem. Then a review of articles confronting the repositioning problem in a fleet operator-based fashion and an user-incentive-based fashion is given. Heuristics for MILP problems and the Simulated Annealing method are later examined. Finally, an overview of the work done in the SmartData@Polito[1] research group is presented.

2.1 Overview

Smart urban mobility solutions have the potential to become a key part of the urban policy agenda for addressing the negative effects of transportation. For this reason they have recently become subject of numerous studies, tackling all aspects of this topic.

Golbabaei et al. [2021] analysed and reviewed 81 articles concerning many different aspects of shared mobility such as the optimal fleet size, the effects on traffic volume and congestion, the travel cost and fares of both the shared mobility system and their effect on traditional taxi service fares. They also reviewed articles on the impact on household locations, on parking demand and spaces, on the number and position of pick-up and drop-off areas and charging stations, as well as on social and travel behaviours. In particular, they pointed out that demand levels and arrival patterns are key considerations in determining the right fleet size. Furthermore, Winter et al. [2016] found that the initial vehicle position has a significant influence on system performance.

Hu et al. [2016] provided an overview and classification of the optimisation and control methodologies employed for smart charging of electric vehicle fleets. They identified three control strategies used by fleet operators including centralised control, transactive control, and price control and discuss their related mathematical modelling methods.

[1] https://smartdata.polito.it

Hu et al. [2016] further discussed the charging cost minimisation through methodologies like dynamic programming in order to obtain the optimal charging schedule. Moreover, they analysed several mathematical modelling frameworks such as linear programming, quadratic programming, dynamic programming, mixed-integer linear and non-linear programming, stochastic programming, robust optimisation, heuristic optimisation and model predictive control in order to model the state of charge of the batteries.

Albuquerque et al. [2021] focused on machine learning techniques' contributions applied to bike-sharing systems. They mainly reviewed articles concerning the prediction of bike spatial and temporal demand and their optimal repositioning. Methods applied in those articles span from random forest, weighted correlation network and Monte Carlo simulation, artificial neural networks, graph convolutional neural networks, long-short term memory neural networks, linear and logistic regression to hierarchical clustering, k-means, k-medoids, among others. They observed that some research used clustering methods like hierarchical clustering to predict the behaviour configurations on bike station spatiotemporal redistribution. A random forest and a graph convolutional neural network with data-driven graph filter has been employed to predict station and city bike demand.

Vallez et al. [2021] reviewed papers on the repositioning problem in dock-based bike-sharing services, observing that, since their inception, the balancing problem has been highlighted as one of the most significant ones that such systems have to face. In particular they pointed out at Ban and Hyun [2019]'s finding that in docking-station based bike sharing systems, the operator-based approach appears to be more effective; on the contrary in dockless bike sharing systems, the user-based strategy seems to outperform the former. Vallez et al. [2021] analysis revealed that such problem has been tackled through methods like stochastic programming, the travelling salesman problem, integer programming, Petri nets to create modular dynamic models, chemical reaction optimisation, game theory, heuristic methods such as tabu search, genetic algorithms or large neighbourhood search. In Pfrommer et al. [2014], the authors coupled the traditional redistribution mechanism with a pricing incentive approach to the consumers. Tang et al. [2020a] focused on reducing three expenses associated with the rebalancing procedure: transportation costs, penalty charges for all stations for not meeting the demand, and holding costs.

2.2 Vehicle Routing Problem (VRP)

The vehicle routing problem (VRP) is one of the most common solutions employed when facing the system rebalancing problem. The VRP is a combinatorial optimisation and integer programming problem, which generalises the travelling salesman problem (TSP), and aims at finding the best set of routes for a vehicle fleet whose task is to deliver to a given set of customers. The classical formulation of this problem assumes the best set of routes to be the one that minimises the total route cost, which usually is the total travelled distance. Constraints on the vehicle capacity, number of customers and on whether or not a point can be visited more than once are usually added. Determining the VRP's optimal

solution is known to be a NP-hard problem[2], therefore heuristics need to be used, since an exact solution can't in general be found.

The road network may be represented as a graph, with arcs representing roads and vertices representing intersections between them. Due to the possibility of one-way streets or varying costs in each direction, the arcs may be directed or undirected. Each arc has a cost associated to it, which is usually the length or travel time of the street.

In Shi et al. [2019] the VRP is applied to the fleet rebalancing problem and solved using an upgraded particle swarm optimisation (PSO) algorithm. A (non-realistic) case study is then conducted to verify the model's and algorithm's validity. The authors formulated the problem as to minimise the operation's overall cost, i.e, the fixed maintenance truck cost and the variable distribution costs. Constraints on the number and capacity of the relocating trucks are included, as well as the balancing of incoming and outgoing trucks in a node and the possibility of every node to be visited from at most one truck. The problem is then solved through a modified version of the PSO, where inertia weights and learning factors had been adjusted.

In Lin and Chou [2012], the authors implemented the VRP to solve the unbalancing problem and then analyzed three algorithms to solve such problem: the nearest neighbour algorithm, the savings method and the farthest insertion algorithm, finding the last one to be the best performing approach. They also faced the metric choice problem for measuring the distance between nodes, using the Google Directions API instead of the euclidean distance to have more reliable estimates.

In Bräysy and Gendreau [2005a] the authors analysed a variation of the classical VRP: the vehicle routing problem with time windows (VRPTW). This formulation adds a further constraint to the problem, requiring a vehicle to deliver to a certain point within a given time interval. The authors, in particular, allowed the vehicles to get to the point of interest before the time window opens and wait without penalty until the service is available, but they did not allow anyone to arrive after the time window closed. In Bräysy and Gendreau [2005a], three heuristics were then studied and their performances analysed. The first one is based on the construction of a new route combining pre-existing ones. The second heuristic is a time-oriented nearest neighbour, which begins each route by locating an unrouted client who is closest to the depot and then subsequently adding the closest unrouted customer. The most effective of the three suggested sequential insertion heuristics creates a route by starting with a "seed" customer and then adding the other unrouted customers until the route is filled depending on specified criteria (like its distance). In addition to route construction heuristics, the performance of several local search methods had also been inspected, which differ on the way new candidate optimal solutions are identified. In Bräysy and Gendreau [2005b], many different meta-heuristic algorithms to solve the VRPTW are explored: tabu search algorithms, as well as genetic

[2]A problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H; that is, assuming a solution for H takes 1 unit time, H's solution can be used to solve L in polynomial time. As a consequence, finding a polynomial time algorithm to solve any NP-hard problem would give polynomial time algorithms for all the problems in NP. Informally, a NP-hard problem is at least as hard as the hardest problems in NP. A NP problem, instead, is a decision problem that can be solved in polynomial time by a nondeterministic Turing machine.

algorithms, algorithms employing neural networks and many more. The authors found meta-heuristic algorithms to be performing generally better than the previously examined construction heuristics and local search algorithms, although being harder to handle in the calibration and implementation.

The vehicle routing problem is not a tool used in the present work since no optimal path needs to be found through the donor and receiver zones. It is, however, the most convenient method to be implemented at the end of the presented procedure: once how many cars need to be taken from or brought to a given city zone has been found (i.e. the relocation strategy), the optimal path through them needs to be elaborated. Based on the features of the system (free-floating/station-based, type of vehicles employed) different kinds of constraints need to be formulated in order to reflect the operational procedure of the support fleet.

2.3 Fleet Operator based relocation

Hellem et al. [2021] confronted the relocation and recharging problem in a rolling-horizon and dynamic fashion, that is, fresh information regarding the distribution of automobiles and their current state of charge is constantly obtained and used to adjust the decisions. A mixed integer linear program is then formulated and solved through an adaptive large neighbourhood search heuristic algorithm, combining a tabu search and a large neighbourhood search. The objective was to maximise profits by supplying a suitable number of cars, charged following an expected ideal distribution of rental vehicles. The authors formulated a solution by assigning vehicles with low battery levels to charging stations, cars in need of relocation to under-supplied zones, and car-moves, routes and schedules to fleet workers. The algorithm was then tested on a simulation environment based on real traffic data collected in the city of Oslo, finding the model to perform better than a greedy heuristic.

In Caggiani et al. [2018], the authors suggested a dynamic bike redistribution approach that starts with a bike number and position forecast over a system operating region and finishes with a relocation Decision Support System, activated at regular gap times during the day in order to adjust the choices made by the operator. The relocation procedure is engaged at regular intervals in order to perform dynamic bike redistribution, with the primary goal of maximising the user satisfaction while minimising vehicle repositioning expenses. A wavelet transform for denoising and signal compression was initially employed in order to subsequently identify first temporal and then spatio-temporal bike demand clusters. Temporal clusters were identified through a hierarchical clustering algorithm, while spatio-temporal clusters were obtained starting from the former ones through the usage of a k-means method. Subsequently, a non-linear autoregressive neural network was employed to predict the bike demand in each spatio-temporal cluster. Two optimisation steps were then carried out: the first one minimises lost users and unfavourable times (i.e. times in which the number of bikes goes below a given threshold in a cluster) and generates a relocation matrix, giving information on how many bikes have to be moved from one cluster to another. The second step involves the resolution of the VRP, in order to find the best path through the donor clusters first (to collect the exceeding bikes) and

then the best path through the receiver clusters. The usefulness of the proposed technique for the real-time management of free-floating bike-sharing systems was tested using a test case study and a full sensitivity analysis was also performed.

Brinkmann et al. [2019] confronted the routing problem in a station-based bike sharing system in a stochastic and dynamic formulation by means of dynamic lookahead policies. Such policies, starting with the current state of the system, simulated future user demand, based on historical data. A Markov decision process was employed in order to determine how many bikes the stuff truck has to pick-up or leave in a station and what is the next station to be visited. The horizons are time-dependent and autonomously parameterised using value function approximation since the demand pattern varies throughout the day. The benefits of both the lookahead and time-dependent horizons of the dynamic lookahead policies were tested in comparisons with conventional policies from the literature and lookaheads with static horizons through simulations based on real-world data of the bike-sharing system in Minneapolis.

Chiariotti et al. [2018] suggested a dynamic rebalancing technique that uses historical data to forecast network dynamics. The authors decided to employ birth-death processes to model the station occupancy and determine when to redistribute bikes, as well as graph theory to choose the optimal path through the stations. As to the birth-death processes, birth and death rates that change throughout time were employed so that how long a station will be self-sufficient before it runs out of bikes or available stalls could be anticipated. Through this problem formulation, a desired state for each station can be computed, which maximises its survival time. After doing so, stations that are either under- or overcrowded are identified and the optimal route through them is obtained by means of the VRP. The suggested paradigm was validated using data from New York City's bike-sharing system. The numerical simulations revealed that this dynamic technique, which can adapt to the network's changing nature, exceeds static rebalancing schemes performances.

Finally, Tang et al. [2020b] developed a dynamic optimisation model and used approximated dynamic programming to make dynamic repositioning decisions, while coping with the problem dimensionality. The idea was to prevent myopic judgements by using a neural network as an approximation function that examines the influence of present actions on future system states.

2.4 User incentive based systems

As an alternative to a centralised control of the relocation process by the fleet manager, some authors have investigated what effects a user incentive based strategy brings about.

In Wang et al. [2022], the authors developed an adaptive relocation model based on user incentives that mixed car and bike sharing. To forecast relocation demand, multi-time-period dynamic thresholds are initially implemented. The authors then introduced a joint relocation mixed integer programming optimisation model aiming at minimising a cost function that considers incentive expenditures and user satisfaction. Multiple dynamic constraints such as state of charge, journey distance, station status, and user orders were added. Subsequently, this joint relocation model was solved using a genetic algorithm.

The relocation model was used to determine the ideal incentive price for users, the initial vehicle configuration at each station, the multi-time-period dynamic thresholds in each station, and the vehicle relocation scheme across all stations. The model was the tested on a real case study of an electric car sharing firm in Shanghai leading to the conclusion that the cost for the operator can be reduced by up to 70%. Wang et al. [2022] finally carried out a sensitivity analysis on the incentive prices.

Fanti et al. [2019] mixed a (incentivised) user-based and an operator based relocation approach. An integer linear program problem for the company staff which aimed at minimising the relocation costs was first developed. The model was then enriched to take user incentives into account, which are given in order to avoid the intervention from the operator. The models were subsequently tested on a case study in order to assess the effectiveness of the user incentives, finding them useful to improve the overall quality of service.

Wu et al. [2019] faced the relocation problem in a free-floating bike sharing system with a user-incentive plan obtained through a ranking and selection method, a data-driven approach in the simulation optimisation field, in order to cope with the complicated structure of uncertainty sets in this context. Such method was applied to solve a profit maximisation model with constraints on the customer service level. Wu et al. [2019] finally concluded that the ideal customer incentive-based rebalancing strategy may not only greatly raise the average daily profit, but also achieve a higher service level, using numerical simulations.

Pan et al. [2018] modelled the problem as a Markov decision process that takes into account both geographical and temporal characteristics of the user demand, constructing a deep reinforcement learning algorithm based on the deep deterministic policy gradient method. After a discretization into pricing areas and time slots, user incentives to conveniently move bikes were obtained through a hierarchical reinforcement pricing algorithm, based on the idea of decomposing the Q-value of an entire area into more easily manageable sub-Q-values of smaller regions, which makes an efficient search for policies possible. A real-case study was finally carried out with data from the city of Shangai, achieving a 40% to 80% un-service ratio decrement over 80% of tested areas.

2.5 Optimisation algorithms

The following section deals with some optimisation solutions explored for the solution of the confronted problem.

An optimisation program is formulated in order to find the optimal relocation strategy, where variables are supposed to take up only integer values. Due to the problem dimensionality, an exclusively integer formulation is not practicable and heuristics need to be resorted to. Some suggested heuristics are therefore explored.

A neighbourhood search is also implemented at the end of the optimisation program, including the simulated annealing strategy, whose features and performances are investigated below.

2.5.1 Heuristics in mixed integer programming

Fischetti and Lodi [2011] confronted the resolution of mixed integer linear programs through heuristics, in order to cope with excessive computational times, deriving from such problems being NP-hard. They examined LP-based heuristics such as the *rounding* of the continuous solution obtained from the LP-relaxation of the initial problem: variables that are supposed to be integer are usually rounded to the nearest integer value. The *diving* heuristic fixes some integer valued variables that take a fractional value in the LP-relaxation solution in a sequential manner. *Pivoting* methods make use of different types of pivots, aiming at maintaining primal feasibility while increasing or decreasing the number of nonbasic 0-1 variables. Many *line search* methods were also presented, whose main construction is based on starting a line search from the LP-relaxation solution of the integer problem; the line search then moves through discretised solutions making use, as an example, of hyperplanes, like in the *OCTANE* method. The *Feasibility Pump* was the last LP-based heuristic examined: each iteration consists in the resolution of the LP-relaxation of the problem and then the rounding of the solution; the following step consists in solving an auxiliary LP which aims at finding a feasible solution to the problem which minimises the L^1 distance to the previously found rounding. They also cited some further improvements to this heuristic.

Fischetti and Lodi [2011] also presented some MILP-based heuristics. The *Local branching* method looks for a new solution in a large neighbourhood search method fashion, exploring the neighbourhood of a starting reference solution, defined through a local branching constraint. Other neighbourhood search methods are the basis of more heuristics: the *relaxation induced neighbourhood search* (RINS) compares the incumbent solution and the LP-relaxation and fixes the integer-constrained variables that agree in value and projects them out of the MILP, so to reduce the size of the problem. The *distance induced neighbourhood search* (DINS) improves the latest method starting from the idea that good solutions are those 'close' to the LP-relaxation one: the algorithm then fixes those variables whose distance between the LP-relaxation and the incumbent solution does not exceed 0.5. Another variation to the RINS is the *relaxation enforced neighbourhood search* (RENS), which fixes already integer variables and rebounds the other ones by using the ceiling and the floor values of the variable's incumbent solution as upper and lower bounds, respectively. Finally, the *polishing* algorithm was presented, which is called on chosen nodes during the exploration of a branch-and-bound tree and is based on the ideas underlying genetic algorithms. The combination phase recalls the RINS algorithm: variables whose values are equal in the parents are fixed and then the resulting MILP is heuristically solved. In the mutation phase, some solutions in the population are chosen and some of their variables are fixed. Finally, the selection phase is performed choosing at random one solution and then a second one only among those ones whose objective value is better than the former.

2.5.2 Simulated Annealing (SA)

Simulated annealing is a probabilistic approach for determining the global minimum of a function with possibly several local minima. It works by simulating the physical process of

a solid being steadily cooled until its structure is frozen at the lowest energy configuration possible. Bertsimas and Tsitsiklis [1993] detailed the approach, its convergence, and its behaviour in applications, limiting themselves to the situation of a cost function specified on a finite set. After defining a proper neighbourhood structure, given a current state of the system, the SA algorithm consists in choosing one of its neighbours at random; the state of the solution, therefore, can be described as a Markov chain. If the cost function computed with the given neighbour improves, then the latter is chosen like in other neighbourhood search algorithms, while, if it does not improve, it may still be chosen as the new incumbent solution with a given probability. The acceptance probability of a new sub-optimal solution is function of both the cost function increase and a temperature cooling schedule, which has to be properly set. As a result, the SA algorithm may be considered as a local search algorithm with periodic upward advances that result in a cost rise. Such moves will hopefully permit the method to lead the incumbent solution away from local minima. Bertsimas and Tsitsiklis [1993] also investigated the convergence of the algorithm, finding a necessary and sufficient condition for the convergence in probability (i.e. for $\lim_{t\to\infty} \mathbb{P}[x(t) \in S^*] = 1$, t being the iterations, x the state of the system and S^* the optimal set): $\sum_{t=1}^{\infty} exp[-d^*/T(t)] = \infty$ and $\lim_{t\to\infty} T(t) = 0$, being T the cooling schedule and d^* the height of S^*. They further observed that d^* is a measure of how difficult it is for the solution to escape from a local minimum and to move towards the global one. Finally, considerations on the convergence speed of the algorithm were made, concluding that results about it involve too large constants (and, therefore, too many iterations to be performed) for the guarantee that the algorithm performs better than an exhaustive search. The authors noticed, however, that, researchers employed SA extensively despite the lack of a solid theoretical foundation for its rate of convergence, discussing several problems and contexts in which it is successfully used, such as image processing, as well as the graph partitioning problem and the graph colouring problem.

As already pointed out by Bertsimas and Tsitsiklis [1993], the cooling schedule impacts the quality of the obtained solution. Nourani and Andresen [1998] compared different temperature cooling schedules: constant thermodynamic speed ($\frac{dT(t)}{dt} = \frac{-v_s T}{\epsilon\sqrt{C}}$, v_s, ϵ and C being parameters to be tuned), exponential ($T(t) = T_0\alpha^t$, T_0 being the initial temperature and α being a parameter to be tuned), logarithmic ($T(t) = \frac{c}{log(t+d)}$, c and d being parameters to be tuned) and linear ($T(t) = T_0 - \eta t$, η being a parameter to be tuned) cooling schedules. For given initial and final states and number of iterations, the authors tested on a very simple and small system and on an NP-hard problem and found the constant thermodynamic speed to be the best performing one, having the least entropy production among the examined schedules. The logarithmic schedule, on the other hand, results in the highest entropy production, while the linear and exponential schedules present similar behaviours, with anyway higher entropy than the constant schedule.

Salamon et al. [1988] further inspected the effects of the choice of the constant thermodynamic speed schedule on the SA algorithm. They hypothesised that the optimal setting for the method requires the distance between the system state and the target (the state in which the system would be in equilibrium) to be constant along the iterations, that is the constant thermodynamic schedule. They finally proved such hypothesis to be consistent applying the schedule to the graph partitioning problem.

2.6 Research group works

The study for this thesis project falls into a wider work on mobility and shared transportation carried out by the research group SmartData@Polito; this center focuses on Big Data technologies, Data Science (from data management, to data modeling, analytics, and engineering), and Machine Learning methodologies applied to several domains of knowledge, finding solutions for both theoretical problems and helping companies toward applications.

In Cocca et al. [2020], making use of real car-sharing trip data from the city of Vancouver and social-demographic characteristics of the city related to urban mobility, spatial and temporal patterns of the user demand for cars were predicted. Long-term and short-term prediction of the temporal demand was performed using historical data through several machine learning algorithms such as auto-regressive models, support vector regression, linear regression, long-short term memory neural networks and random forest regression with the last method (and multivariate models in general) reaching the best performance both in the long and in the short term prediction. The authors also observed that, assuming perfect weather forecast, the mean absolute percentage error in the prediction can be improved by around 3%. Using social-demographic data of the city, spatial demand patterns were predicted using support vector machine regression, obtaining the best performance with the polynomial kernel, and random forest regression, performing feature ranking and selection in order to disclose what socio-demographic characteristics influence demand patterns the most.

In Cocca et al. [2019, 2018] the problem of determining the optimal position of charging stations and designing the optimal return policy in a FFCS system was faced. Two different approaches to the optimal charging station positioning problem were discussed: the first one is a heuristic placement based on three possible definition of likelihood. The second approach uses two different data-driven simulation-based meta-heuristics: a local-search algorithm based on a hill-climb method and a genetic algorithm. These algorithms performance and the impact of different return policies were assessed through simulations based on real data coming from the cities of Turin and Milan, revealing that equipping only 5% of the city zones with charging stations allows the system to be self-sustainable and all trip requests to be satisfied with limited user discomfort.

Ciociola et al. [2020a] proposed a way to design sharing systems for electric scooters and heuristic approaches to solve the relocation and recharge problems. Real world data from the cities of Minneapolis and Louisville were first processed with spatio-temporal disaggregation techniques in order to increase resolution and obtain a realistic demand model. First, for each one of the identified time bins, the demand in time was modeled through modulated Poisson processes, then the Kernel Density Estimation (KDE) technique was used to model the demand in space within each time bin. A mobility and charging simulator is also developed and used based on the demand model to gauge the impact of design choices like the fleet size, the battery threshold that triggers a charging operation, the provider workflow size, the average time to reach the e-scooter and the volunteers' willingness to handle charging on some system's performance metrics. The study revealed that a high number of e-scooters and battery swap operations are required to meet the trip demand and prevent customers from running out of battery. Moreover,

reducing the time it takes for employees to get to their e-scooters and replace their batteries has a significant impact for reducing costs. Finally, including users in the recharging process can further reduce costs.

Fassio et al. [2021] compared the economical and environmental impact of FFCS with different source of power (Electric, internal combustion engine with petrol, LPG and diesel). A demand model was first developed using real-world data from a non-electric FFCS like already discussed in Ciociola et al. [2020a] and then fed to an event-based simulator. Results revealed that quicker chargers provide just a minor benefit; consequently, a slower and less expensive charger may be used. Furthermore, the widespread availability of EV infrastructure aids in minimising the time it takes to move automobiles to a charging station. When compared to a gasoline-powered fleet, EV fleets proved to have the minimum environmental effect, with a reduction of more than 50%. Unfortunatelu, in terms of economics, EV FFCS proved to be the least competitive system, owing to the high vehicle expenses.

In Tolomei et al. [2021] the benefits of a relocation strategy were assessed and how important it is to accurately satisfy the mobility demand was investigated. In particular, a baseline relocation model based on the average expected demand and a deep-learning based predictive model were compared. The latter model was used to predict future routing matrices and was obtained assembling a convolutional neural network able to learn the spatio-temporal demand patterns, a long-short term memory neural network to reinforce the temporal correlation and finally a fully connected network able to consider external factors, too. Simulating the demand in the cities of Austin and Louisville, the utility of a relocation strategy is confirmed (especially for smaller fleets, while its advantage decreases with more available vehicles) and the utility of a more precise prediction of the demand through the deep learning model is proven, especially in bigger cities like Austin.

In Ciociola et al. [2020b] the impact of heuristic policies for charging batteries of EV vehicles in FFCS was investigated. One scenario implies charging operations to be performed only by the operator, while the other one involves charging poles distributed into the operational area, where both the operator and the users can plug the cars. The study was then extended to the cities of Milan, Vancouver and New York, showing that a distributed charging infrastructure allows for better system performances than a centralised hub according to all metrics employed in the work. Moreover, the authors found that involving the users in the charging operation can further reduce the relocation costs.

Barulli et al. [2020] analyzed through simulation based on data coming from the city of Turin the scalability of sharing system and different charging infrastructure designs. Their results showed that the optimal charging station placement involves more poles to be placed in high demand areas, resulting in a decrease in the operational costs for the management fleet. They also found that the charging infrastructure has to grow proportionally to the trip demand, while the fleet size can grow much slower (sublinearly) with respect to the demand.

Chapter 3

Methodology

This chapter provides a high level presentation of all the algorithms employed to confront the optimal relocation problem, underlining their main features, strengths and weaknesses. A mathematical formalisation of the problem is first discussed and the notation employed throughout the thesis is presented. An overview and summary of the developed algorithms is then given and, finally, the method to evaluate the performances of the different procedures is explained.

3.1 Problem formulation and objective

The aim of the devised procedure is to find the optimal relocation schedule in order to satisfy as many trip requests in a shared mobility system as possible. With this in mind, the problem has been formalised and some notation has been introduced, which is presented in this section. Let $\mathcal{Z} = \{0, \dots, Z-1\}$ be the set of city zones ($\#\mathcal{Z} = Z$). Let P be the **optimisation horizon**, i.e. the number of time-frames the optimal relocation decision is to be found, and T the **look-ahead horizon**, i.e. the time horizon up to which the optimisation procedure is to be carried out each time. Denoting $\mathcal{P} = \{0, \dots, P-1\}$ the set of time-frames at the beginning of which a new optimisation step is to be performed, the set of time-frames involved in the p-th step is $\mathcal{T}_p = \{p, p+1, \dots, p+T-1\}$, $p \in \mathcal{P}$.

At the generic optimisation round starting at $p \in \mathcal{P}$, the relocation strategy may be codified as

$$\begin{aligned} \mathbf{r} &= (r_{ti})_{t\in\mathcal{T}_p,\ i\in\mathcal{Z}},\ r_{ti} \in \mathbb{Z} \\ \text{where } r_{ti} > 0 &\quad \text{if zone } i \text{ is a receiver during time-frame } t \\ r_{ti} < 0 &\quad \text{if zone } i \text{ is a donor during time-frame } t \end{aligned}$$

Moreover, let $\mathcal{S}$ be the space of possible relocation strategies, i.e. $\mathbf{r} \in \mathcal{S}$.

Other variables have been of use for the mathematical formulation of the problem:

- $\mathbf{s} = (s_{ti})_{t\in\mathcal{T}_p, i\in\mathcal{Z}},\ s_{ti} \in \mathbb{N}$ is the number of vehicles in zone i at the beginning of time-frame t;

- $\mathbf{EO} = (EO_{ti})_{t\in\mathcal{T}_p,i\in\mathcal{Z}}$, $EO_{ti} \in \mathbb{N}$ is the number of vehicles leaving zone i during time-frame t;
- $\mathbf{EI} = (EI_{ti})_{t\in\mathcal{T}_p,i\in\mathcal{Z}}$, $EI_{ti} \in \mathbb{N}$ is the number of vehicles arriving in zone i during time-frame t.

Some external data were extracted and provided, like described in chapter 7:

- $N \in \mathbb{N}$ is the total number of vehicles in the system, so $\sum_{i\in\mathcal{Z}} s_{ti} = N \ \forall t$;
- $R_{max} \in \mathbb{N}$ is the maximum number of vehicles which can be relocated by the operator within a time-frame;
- $\mathbf{V} = (V_{ij}^t)_{t\in\mathcal{T}_p,i,j\in\mathcal{Z}}$, $V_{ij}^t \in \mathbb{N}$ is the number of vehicles moving from zone i to zone j during time-frame t;
- $\mathbf{O} = (O_{ti})_{t\in\mathcal{T}_p,i\in\mathcal{Z}}$, $O_{it} \in \mathbb{N}$ is the number of outgoing trip requests from zone i during time-frame t;
- $\mathbf{I} = (I_{ti})_{t\in\mathcal{T}_p,i\in\mathcal{Z}}$, $I_{it} \in \mathbb{N}$ is the number of incoming trip requests to zone i during time-frame t;
- $\mathbf{p} = (p_{ij}^t)_{t\in\mathcal{T}_p,i,j\in\mathcal{Z}}$, $p_{ij}^t \in \mathbb{R}^+$ is the probability of a vehicle to be moving from zone i to zone j during time-frame t.

Let us notice that a total number of $P + T$ time frames will be involved in the whole optimisation procedure. At the generic optimisation round for time-frame $p \in \mathcal{P}$, the aim of the problem is to maximise the total satisfied outgoing trip demand, that is:

$$\max \sum_{t\in\mathcal{T}_p} \sum_{i\in\mathcal{Z}} EO_{ti}, \tag{3.1}$$

Observing that $\mathbf{O}$ can be regarded in this contex as some constant given data and not a problem variable and recalling the observation in (5.1), (3.1) can be restated equivalently as

$$\min \sum_{t\in\mathcal{T}_p} \sum_{i\in\mathcal{Z}} O_{ti} - \sum_{t\in\mathcal{T}_p} \sum_{i\in\mathcal{Z}} EO_{ti}.$$

which can be rewritten as

$$\min \sum_{t\in\mathcal{T}_p} \sum_{i\in\mathcal{Z}} O_{ti} - EO_{ti}, \tag{3.2}$$

i.e., the objective of the problem is equivalent to the minimisation of the total unsatisfied outgoing trip demand.

3.2 Overview on implemented algorithms

A first method to find the optimal relocation strategy, presented in chapter 4, is the dynamic programming approach. This technique allows - in principle - to find the optimal solution to the problem throughout all time frames, with possibly an infinite look ahead.

That is, starting at time frame 0 and solving the problem employing the dynamic programming method, we are able to identify the relocation strategy leading to the optimal outcome through all time frames up to the last one. To any possible state of the system, it is possible to associate an optimal decision, therefore adapting to the evolution of the system and adjusting the decision on the go, always choosing what is best to do to get the best possible outcome. Therefore, this algorithm can be thought to be involving only one optimisation round starting at time frame 0 with look ahead P, the optimisation horizon. Unfortunately, this solution is very little scalable, since any possible state of the system needs to be evaluated. Producing all possible system states becomes soon infeasible when increasing the number of vehicles and city zones involved in the optimisation process and, therefore, this approach cannot guarantee a solution for real-world cases.

Hence, approximate methods need to be resorted to for a relocation strategy to be worked out in reasonable times and with limited resources. As stated in section 3.1, all variables are supposed to take up integer values. Unfortunately, formulating an integer linear program with hundreds of integer variables still results in a problem which would take too much time to be solved. Therefore, some or all integrality constraints were relaxed, to speed up the resolution algorithm. A sequential algorithm based on the formulation of a linear program, presented in chapter 5, is therefore employed. The optimisation program can be formulated in three different ways:

1. a Linear Program, where all variables' integrality constraints have been dropped. Since the resulting relocation strategy is coded by continuous variables, a post-processing step follows to get an integer valued relocation vector;

2. a Mixed Integer Linear Program, where the integrality constraints have been kept only on the relocation variables r_{ti}, while relaxing them on all other variables;

3. a combination of 1 and 2, where the pure LP is first solved and the r_{ti} whose optimal value is found to be 0 are fixed, leaving the remaining ones as the integer constrained variables for the MILP in 2.

Relaxing the integrality constraints on r_{ti} returns unrealistic solutions, as it is impossible to move half a vehicle, therefore a rounding step needs to be performed. Nevertheless, solving a linear program is often virtually immediate, even though the problem dimensionality is very high and the number of constraints is rather elevated. Keeping the integrality constraints only on the relocation variables returns an integer strategy, therefore no rounding steps are needed. However, mixed integer linear programs are known to be very computationally expensive. As will be shown in chapter 8, short look ahead periods (which involve a smaller number of variables) can be solved fairly fast anyway, while, working with longer look ahead periods (and, therefore, more variables) requires very long computational times. Finally, the idea behind the method in 3 is to first solve the LP, identify which relocation variables take up an optimal value near to 0 and fix their value in the MILP, in order to reduce the dimensionality of the optimisation problem.

A neighbourhood search, described in chapter 6, can be performed starting from the solution provided by the linear program to further improve the quality of the strategy. Observe that a neighbourhood search might be used alone, starting from a random solution, but the very high dimensionality of the problem would make the convergence to a

sufficiently good solution too slow to be used in practice. On the contrary, starting from an already good solution can give hope to find a more performing one. Finally, the neighbourhood search can be - and it has been, indeed - implemented to keep the relocation solution integer.

The algorithm in question is defined to be sequential since the optimisation problem, and possibly the neighbourhood search, is performed at the beginning of each time frame in order to adjust the relocation strategy based on the realisation of the trip demand and, therefore, of the system state.

The features of the presented algorithms are summarised in Table 3.1.

3.3 Implementation

This section deals with the practical implementation of the aforementioned tools for the resolution of the problem. For each optimisation round starting with time-frame $p \in \mathcal{P}$, an optimisation procedure is carried out, with any of the methods described in section 3.2. Finally, a way to assess the **quality of the solution** has been implemented and is presented in section 3.4. After obtaining the optimal relocation decision for time-frame p, the **system reaction is simulated**, employing data on the trip demand, and a new state of the system is obtained, which will be used as the initial state for the optimisation round involving time-frame $p + 1$. The backbone of the whole procedure can be summarised in the following general and high level steps:

$s \leftarrow s_0$
$total\_satisfied\_demand \leftarrow 0$
for $p = 0, \dots, P - 1$ **do**
  $r \leftarrow$ solve_optimisation_problem$(s, \mathcal{T}_p)$
  $s_{new},\ satisfied\_demand \leftarrow$ simulate_system(r, s, p)
  $total\_satisfied\_demand \leftarrow total\_satisfied\_demand + satisfied\_demand$
  $s \leftarrow s_{new}$
end for

The look-ahead horizon T is fixed and an initial state s_0 needs to be provided. At each loop a relocation strategy is determined for time frame p using information about the look ahead period time frames $\mathcal{T}_p$. According to the implemented strategy, the system is simulated and the satisfied mobility demand and its next state are obtained, the latter being used in the following round as the initial state of the optimisation problem. The procedure is exemplified in Figure 3.1.

3.4 System simulation and objective evaluation

The algorithm in section 3.3 involves a system simulation step, whose aim is to reproduce the reaction of the system when a given relocation strategy is carried out by the operator. During the simulation, the satisfied trip demand obtained implementing the chosen strategy is computed, in order to gauge the problem objective as formulated in (3.1) and, therefore, assess the quality of the given method.

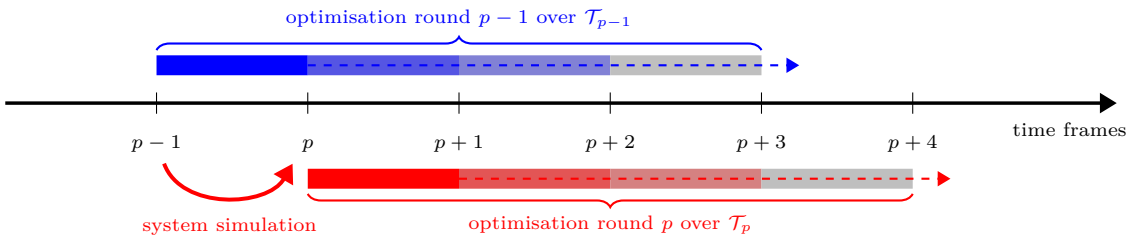

Figure 3.1. Optimisation procedure representation.

The system simulation requires a traffic flow scenario, to estimate how the system performances change and, therefore, how good the strategy obtained from the method is. The traffic flow scenario consists of an ordered list of trips, described by a couple of indexes specifying the departure and arrival city zones. It is very important to notice that only one traffic flow scenario is used here and is common to all the carried out attempts, in order not to facilitate any algorithm choice. A few Monte Carlo runs employing different scenarios would have been more appropriate to get a more accurate estimate of the algorithm performances but, unfortunately, that would have made the computational effort of the estimation unbearable, since the optimisation rounds should have been executed not only P times, but the whole procedure should have been carried out for every Monte Carlo run.

For implementation purposes, a collection of data structures $\{trips_t\}_{t\in\mathcal{T}_p}$ has been used, obtained from the array $\mathbf{V}$ described in section 3.1. $trips_t$, $t \in \mathcal{T}_p$, gives an ordered list of trip requests during time frame t, containing each couple (i, j) V_{ij}^t times, accounting for the number of trips starting in zone i and ending in zone j during time frame t. Therefore, looping through $trips_t$ gives a mobility demand scenario, which is used in the evaluation of a relocation strategy.

Finally, the system simulation procedure during the generic time frame p can be described. Given the current state of the system at time frame p, *current_state*, and the relocation strategy $\mathbf{r} = (r_{pi})_{i\in\mathcal{Z}}$, the algorithm to compute the objective value for time frame p consists in the simulation of the system and in the subsequent computation of the satisfied trip requests, i.e. the number of times a starting zone i has at least one car.

The simulation coincides with the function called *simulate_system* in section 3.3 and is articulated in the following steps:

1. set the current state
   $s \leftarrow current\_state$,
   and initialise the satisfied trip counter to 0
   $satisfied\_trips \leftarrow 0$;
2. compute s^r, the state of the system after the beginning of the relocation process, i.e. after all vehicles have been collected from the operator to be transferred to another zone:
   $s_i^r \leftarrow s_i + min\{r_{pi}, 0\},\ \forall i \in \mathcal{Z}$;

3. copy s^r to keep track of the zone availability:
   $availability \leftarrow copy(s^r)$;

4. looping over the first half of $trips_p$, for each couple (i, j) identifying a trip request, compute the new state of the system and update the zone availability only if the request can be carried out:
   if $availability_i > 0$ **then**
   $s_i^r \leftarrow s_i^r + 1$
   $s_j^r \leftarrow s_j^r - 1$
   $availability_i \leftarrow availability_i - 1$
   $satisfied\_trips \leftarrow satisfied\_trips + 1$
   end if

5. update s^r simulating the redistribution of the vehicles after the ending of the relocation process:
   $s_i^r \leftarrow s_i^r + max\{r_{pi}, 0\}$;

6. looping over the second half of $trips_p$, for each couple (i, j) identifying a trip request, compute the new state of the system and update the zone availability only if the request can be carried out, like in steps 3 and 4;

7. update the current system state
   $s \leftarrow s^r$

   and return it, together with the number of satisfied trip requests:
   return s, *satisfied_trips*

To take the time for the relocation procedure to be carried out into account, the state of the system is first updated in 3 removing from each zone the number of cars to be transferred according to the given strategy. The first half of trip requests is then simulated, modifying the system. Successively, the system state is updated in 6 emulating the redistribution of the relocated cars, making them available for users to rent them. Finally, the second half of trips is simulated.

Method	Solution optimality	Look ahead period	Efficiency	Scalability	Relocation values	System state values
Dynamic programming	Optimal	Possibly infinite	Slow	Little scalable	Integer	Integer
Linear program	Approximated	Finite	Fast	Easily scalable	Continuous	Continuous
Mixed integer linear program	Approximated	Finite	Fast with short look aheads	Little scalable	Integer	Continuous
LP + MILP	Approximated	Finite	Fast with short look aheads	Little scalable	Integer	Continuous
Neighbourhood search	Approximated	Finite	Slow	Little scalable	Integer	Integer

Table 3.1. Summary of implemented algorithms, condensing their specificities and performances, as worked out in section 8.3.

Chapter 4

Dynamic programming approach

The first method undertaken to solve the optimal relocation problem in a non-myopic forward-looking way was the dynamic programming approach.

The idea behind this method (or rather, principle) is first briefly described, then its implementation to fit the problem's characteristics is illustrated.

4.1 The dynamic programming principle

Dynamic programming (DP) is a useful technique for tackling difficult dynamic optimisation problems. It is not, however, a technique like linear programming. DP, on the other hand, may need a significant amount of customisation and implementation effort.

The basic concept is that DP is a decomposition principle that may be used to break down a complex multistage problem into a series of simpler single-stage problems.

The idea of system state is crucial to dynamic programming, and a mathematical model is needed to capture its evolution through time as a result of decisions and external inputs. External inputs relate to the influence of the outside world on the system of interest and therefore cannot be controlled, whereas decisions are under the agent's control.

The building blocks of this approach are the following:

- the time instants with indexes $t = 0, 1, 2, \ldots, T-1$, at which the system status is observed and decisions are taken;
- s_t, $t = 0, 1, 2, \ldots, T-1$, which summarises all of the information on the system;
- the vector of decision variables, also called control variables, is denoted by x_t, $t = 0, 1, 2, \ldots, T-2$;
- a sequence of exogenous factors denoted by ξ_t, $t = 0, 1, 2, \ldots, T-1$, considered as random perturbations or risk factors;

- a state transition function g_t, $t = 0, 1, 2, \ldots, T-1$, such that $s_{t+1} = g_{t+1}(s_t, x_t, \xi_{t+1})$, which represents the system dynamics.

If a performance measure H over a time-horizon is to be optimised, the multiperiod problem can be stated in a general form as

$$\text{opt } \mathbb{E}[H(x_0, x_1, \ldots, x_{T-2}; s_0, s_1, \ldots, s_{T-1})],$$

where the expected value is taken with respect to the exogenous factors $\xi_1, \xi_2, \ldots, \xi_{T-1}$.

Let's suppose the problem takes up a Markov structure[1] and the objective function assumes an additive form like

$$\mathbb{E}\left[\sum_{t=0}^{T-2} \gamma^t f_t(s_t, x_t) + \gamma^T F_{T-1}(s_{T-1})\right],$$

where f_t and F_{T-1} are performance measures at the intermediate and final times and $\gamma \in (0,\ 1]$ is a discount factor which expresses a trade-off between immediate and future contributions of the performance measure. Then the so-called *value function* can be defined, which maps a state of the system at a certain time to its value. The value $V_t(s)$ of state s at time t should represent the predicted reward/cost attained when the optimum strategy from time t on is implemented, i.e.

$$V_t(s_t) = \underset{x_t \in \mathcal{X}_t(s_t)}{\text{opt}} \{f_t(s_t, x_t) + \gamma \mathbb{E}[V_{t+1}(g_{t+1}(s_t, x_t, \xi_{t+1}))|s_t, x_t]\}, \tag{4.1}$$

where $\mathcal{X}_t(s)$ is the set of feasible decisions at time t if the system is in state s.

(4.1) is usually referred to as **Bellman's equation** and encapsulates the core idea of dynamic programming: when in state s_t at time t, decision $x_t \in \mathcal{X}_t(s_t)$ should be taken, which optimises the sum of the immediate cost/reward and the expected value of the next state, depending on the state and the decision themselves and on the random realisation of the exogenous risk factors.

With this approach, the optimal decision can be obtained by solving a single-stage optimisation problem using the next value function, therefore in a feedback fashion, i.e. starting from the last time period and proceeding back to the starting one. The optimal decision policy μ can finally be determined, where

$$\mu = (\mu_0, \mu_1, \ldots, \mu_{T-2}),$$
$$x_t = \mu_t(s),$$

i.e. a decision is associated to each state at each time.

Furthermore, just as other functional equations like differential equations, boundary conditions are needed to solve the problem. Going the procedure backwards in time, the most natural boundary condition is the terminal condition

$$V_{T-1}(s_{T-1}) = F_{T-1}(s_{T-1}) \quad \forall s_{T-1}.$$

[1] In probability theory and statistics, the term Markov property refers to the memory-less property of a stochastic process, i.e. a process whose future states are assumed to depend only on the current state.

4.2 Implementation

This next section deals with how the dynamic programming principle has been adapted to the problem in order to elaborate the optimal relocation strategy. As discussed in section 3.2, this approach can be thought of as one single optimisation round over the whole optimisation period, whose aim is to minimise the total cost function. Therefore, $T = P$ and the set of time frames over which the procedure is carried out can simply be identified with $\mathcal{T} = \{0, \dots, P-1\}$.

Since the dynamic programming - as observed in section 4.1 - is more a principle than an actual algorithm, some work has to be done to suit the problem peculiarities. Specifically, since future contributions to the total cost do not need to be weighted differently, the discount factor can be set to 1, i.e. $\gamma = 1$.

A system state at time t can be described with the set of variables $\mathbf{s}_t = (s_{ti})_{i \in \mathcal{Z}},\ t \in \mathcal{T},\ s_{ti} \in \mathbb{N}$, is the number of vehicles in zone i at time frame t. A system state $\mathbf{s}_t$ immediate cost function at time t can be defined as

$$f_t(\mathbf{s}_t) = \sum_{i \in \mathcal{Z}} (O_{ti} - s_{ti})^+,$$

where $(O_{ti} - s_{ti})^+$ represents the positive part[2] of $(O_{ti} - s_{ti})$. Defined like this, the immediate cost function has the meaning of total unsatisfied outgoing mobility demand during time frame t, since $O_{ti} - s_{ti}$ represents the unsatisfied demand in the city zone i during time frame t. Being a cost function, the procedure has to be developed to minimise the sum of the cost function over all time frames.

Since a limit R_{max} to the relocation within a time frame has to be respected, not all system states are reachable through a relocation step. Given a state $\mathbf{s}_t$ of the system, we can associate to it a set of reachable states $\mathcal{R}(\mathbf{s}) \subseteq \mathcal{S}$ and

$$\hat{\mathbf{s}}_t \in \mathcal{R}(\mathbf{s}_t) \iff \sum_{i \in \mathcal{Z}} (s_{ti} - \hat{s}_{ti})^+ \leq R_{max}.$$

That is, state $\hat{\mathbf{s}}_t$ is reachable from state $\mathbf{s}_t$ if they differ for the location of at most R_{max} vehicles. Notice that, if the positive part was not used, the summation would always be equal to zero, since

$$\sum_{i \in \mathcal{Z}} (s_{ti} - \hat{s}_{ti}) = \sum_{i \in \mathcal{Z}} s_{ti} - \sum_{i \in \mathcal{Z}} \hat{s}_{ti} = N - N = 0,$$

where N is the total number of vehicles in the system.

[2] The positive part of a real-valued function $f : \mathcal{X} \to \mathbb{R}$ is defined by the formula

$$f^+(x) = max(f(x),\ 0) = \begin{cases} f(x) & if\ f(x) \geq 0 \\ 0 & if\ f(x) < 0 \end{cases} \quad \forall x \in \mathcal{X}$$

Also notice that,

$$\hat{\mathbf{s}_t} \in \mathcal{R}(\mathbf{s}_t) \iff \mathbf{s} \in \mathcal{R}(\hat{\mathbf{s}}),$$

i.e., $\hat{\mathbf{s}_t}$ is reachable from $\mathbf{s}_t$ if and only if $\mathbf{s}_t$ is reachable from $\hat{\mathbf{s}}_t$.

Finally, the demand is assumed to be perfectly known, and (4.1) can be stated as

$$V_t(\mathbf{s}_t) = f_t(\mathbf{s}_t) + \min_{\mathbf{r}_t} V_{t+1}(g_{t+1}(\mathbf{s}_t, \mathbf{r}_t, \xi_t)),$$

where V_t is the value function at time frame t, g_t is the state transition function, $\mathbf{r}_t$ is the relocation decision at time t and ξ_t are the exogenous factors, which, in this context, are the incoming and outgoing mobility demand from each city zone.

The algorithm to compute the value function of every state at each time frame is the following:

1. Compute the immediate cost function for every state at time $T-1$, which will be its value at the final time frame:

   for $s \in \mathcal{S}$ **do**
   $V_{T-1}(\mathbf{s}) \leftarrow \sum_{i \in \mathcal{Z}} (O_{T-1,i} - s_{T-1,i})^+$
   end for

2. Compute the value for every state at every intermediate time frame. Its value depends on the state having the lowest value in the set of reachable states, which has to be found. The value of each state at the current time frame will be the sum of its immediate cost and the value of the most convenient state, which is the reachable state having the lowest value. The relocation strategy can be found as the difference vector between the arrival and the initial state:

   for $t = T-2, \ldots, 1$ **do**
   for $\mathbf{s}_t \in \mathcal{S}$ **do**
   $V_{min} \leftarrow +\infty$
   $\hat{\mathbf{s}}_{min} \leftarrow$ **None**
   for $\hat{\mathbf{s}_t} \in \mathcal{R}(\mathbf{s}_t)$ **do**
   $\hat{\mathbf{s}}_t^d \leftarrow compute\_after\_demand(\hat{\mathbf{s}}_t)$
   if $V_{min} > V_{t+1}(\hat{\mathbf{s}}_t^d)$ **then**
   $V_{min} \leftarrow V_{t+1}(\hat{\mathbf{s}}_t^d)$
   $\hat{\mathbf{s}}_{min} \leftarrow \hat{\mathbf{s}_t}$
   end if
   end for
   $\mu_t(\mathbf{s}) \leftarrow \hat{\mathbf{s}}_{min} - \mathbf{s}_t$
   $immediate\_cost \leftarrow \sum_{i \in \mathcal{Z}} (O_{ti} - s_{ti})^+$
   $V_t(\mathbf{s}_t) \leftarrow immediate\_cost + V_{min}$
   end for
   end for

3. Compute the value for the initial state $\mathbf{s}_0$ at the initial time frame. Its value depends on the state having the lowest value in its set of reachable states, which has to be found:

$V_{min} \leftarrow +\infty$
$\hat{\mathbf{s}}_{min} \leftarrow$ **None**
for $\hat{\mathbf{s}_0} \in \mathcal{R}(\mathbf{s}_0)$ **do**
  $\hat{\mathbf{s}}_0^d \leftarrow$ *compute_after_demand*$(\hat{\mathbf{s}}_0)$
  if $V_{min} > V_1(\hat{\mathbf{s}}_0^d)$ **then**
    $V_{min} \leftarrow V_1(\hat{\mathbf{s}}_0^d)$
    $\hat{\mathbf{s}}_{min} \leftarrow \hat{\mathbf{s}}_0$
  end if
end for
$\mu_0(\mathbf{s}) \leftarrow \hat{\mathbf{s}}_{min} - \mathbf{s}_0$
$immediate\_cost \leftarrow \sum_{i \in \mathcal{Z}} (O_{0i} - s_{0i})^+$
$V_0(\mathbf{s}_0) \leftarrow immediate\_cost + V_{min}$

In the previous algorithm a function called *compute_after_demand* is used, which determines the resulting state from a starting one, considering the effect of the demand patterns, contained in the data collected in $\mathbf{O} = (O_{ti})_{t \in \mathcal{P}, i \in \mathcal{Z}}$ and $\mathbf{I} = (I_{ti})_{t \in \mathcal{P}, i \in \mathcal{Z}}$, like described in section 3.1. The function is articulated in the following steps:

1. compute the overall mobility demand at time frame t and the resulting system state:
   $demand_i \leftarrow I_{ti} - O_{ti}, \forall i \in \mathcal{Z}$
   $\hat{\mathbf{s}}_t^d \leftarrow \hat{\mathbf{s}}_t \ + \ demand$

2. set a variable to account for the unbalance in the state:
   $unbalance \leftarrow 0$

3. reset to zero the zone indexes whose state is negative and add the reset value to the unbalanced amount counter:
   for $i \in \mathcal{Z}$ **do**
     $unbalance \leftarrow unbalance - \hat{\mathbf{s}}_i^d$
     $\hat{\mathbf{s}}_{ti}^d \leftarrow 0$
   end for

4. if $unbalance \neq 0$, then the state has to be rebalanced. To do so, compute the list of indexes whose value can be decreased to rebalance the vector, which are the ones with some incoming demand, assumed to not have been carried out:
   $indexes \leftarrow list()$
   for $i \in \mathcal{Z}$ **do**
     if $I_{ti} > 0$ **then**
       $indexes.append(i)$[3]
     end if
   end for

[3]Given a data-structure implemented as a list, the *append* operation adds the given element as the last one in the list.

5. while the unbalance is a positive value, decrease by one unit the entries given by the *indexes* vector, starting over if necessary:

$i \leftarrow 0$
while $unbalance > 0$ **do**
  if $\hat{\mathbf{s}}^d_{ti} > 0$ **then**
    $\hat{\mathbf{s}}^d_{ti} \leftarrow \hat{\mathbf{s}}^d_{ti} - 1$
    $unbalance \leftarrow unbalance - 1$
  end if
  if $i \neq len(indexes) - 1$ **then**
    $i \leftarrow i + 1$
  else
    $i \leftarrow 0$
  end if
end while

The research into the set $\mathcal{R}(\mathbf{s}_t)$ of reachable states of $\mathbf{s}_t$, together with the above function gives the state transition function g_t, depending on the current state of the system, the relocation strategy and the exogenous factors.

After having associated a decision to every single system state through the policy function μ_t, the optimal strategy found through this algorithm can be implemented, with the precious advantage of dynamically adapting to the system evolution without any need to solve the problem again. Starting from an initial state $\mathbf{s}_0$, for each time frame, the optimal relocation decision associated to the current system state is implemented, the system reaction is then simulated, like described in section 3.4, and a new state of the system is obtained, which will be used as the current state in the following time frame. The procedure can be described by the following algorithm:

$\mathbf{s} \leftarrow \mathbf{s}_0$
for $t = 0, \ldots, P - 2$ **do**
  $r \leftarrow \mu_t(\mathbf{s})$
  $\mathbf{s}_{new}$, _ $\leftarrow$ simulate_system(r, $\mathbf{s}$, t)
  $\mathbf{s} \leftarrow \mathbf{s}_{new}$
end for

Observe that *simulate_system* is the same function described in section 3.4.

Chapter 5

Sequential approximated linear programming and mixed integer linear programming solution

This chapter focuses on the formulation and resolution of the linear program employed to confront the relocation problem. The optimisation program can be solved in three different ways, based on the number of integrality constraints forced on the variables. A further optimisation step can be added at the end and is based on a neighbourhood search, described in chapter 6. A theoretical background is first provided, giving a brief insight on linear and mixed integer linear programs, as well as on numerical methods to obtain their solution. The mathematical formulation of the problem and the implementation of these tools for its resolution is successively described.

5.1 Theoretical background

5.1.1 Linear programs (LP)

Linear programming is a technique for optimising a linear objective function subject to linear equality and inequality constraints. The feasible region of a LP, i.e. the set of points defined by the problem constraints which are eligible to be the optimum point, is a polyhedron, coming from the intersection of a finite number of half spaces, each one defined by a constraint. The objective function is a linear combination of the problem's variables and is defined on such feasible region.

Assuming $\mathbf{x} \in \mathbb{R}^n$, i.e. a problem with n optimisation variables, an LP can be stated

in more than one way, the most general one being:

$$\begin{aligned} \min_{\mathbf{x}} \quad & \mathbf{c}^T\mathbf{x} + d \\ s.t. \quad & \mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq} \\ & \mathbf{A}_{in}\mathbf{x} \leq \mathbf{b}_{in}, \end{aligned}$$

where $\mathbf{c} \in \mathbb{R}^n$ gives the linear combination of the problem's variable and, together with $d \in \mathbb{R}$, defines the objective function. $\mathbf{A}_{eq} \in \mathbb{R}^{k,n}$ and $\mathbf{b}_{eq} \in \mathbb{R}^k$ provide the linear equality constraints, while $\mathbf{A}_{in} \in \mathbb{R}^{l,n}$ and $\mathbf{b}_{in} \in \mathbb{R}^l$ express the linear inequality constraints. k and l are, respectively, the number of equality and inequality constraints.

LPs are pretty convenient, being a linear function both convex[1] and concave[2], which implies that any local optimum point is also a global one.

Let's also observe that LPs are usually stated as minimization problems, but also a maximization problem is treatable by switching the objective function sign, i.e.

$$\max_{\mathbf{x}} \mathbf{c}^T\mathbf{x} + d \quad \Longleftrightarrow \quad \min_{\mathbf{x}} -(\mathbf{c}^T\mathbf{x} + d). \tag{5.1}$$

Geometrically - it can easily be proven - the optimal solution lies on one of the vertices of the polyhedron corresponding to the feasible set.

Many methods have been developed in order to solve LPs. One of the best know and most employed algorithms is the **simplex method**. It works by starting at a basic vertex in the feasible set and proceeding towards adjacent vertices repeatedly, as long as the objective gets improved and until the best solution is obtained. Pivoting techniques are employed to avoid cycling events or that the problem gets stuck with no further improvements of the objective function without the optimum being reached.

In real-life applications, the simplex method is rather efficient and can be relied upon to obtain the global optimum. The simplex algorithm has been shown to effectively solve "random" problems, i.e. in a cubic number of steps, as found by Borgwardt [1987].

5.1.2 Mixed integer linear programs (MILP)

Mixed integer linear programming is a very similar optimization technique, in principle, to linear programming. The only difference between MILPs and LPs is that some optimization variables are also constrained to take up only integer values. Like in LPs, the objective function is a linear combination of the problem's variables, and equality and inequality constraints may be added in order to define a feasibility region. Therefore, if

[1] A function $f : \mathcal{X} \to \mathbb{R}$ is convex if $\forall t \in [0, 1]$ and $\forall x_1, x_2 \in \mathcal{X}$ then $f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$. Convexity is a very desirable property of functions when facing a minimization problem, being local optima also global ones in this case.

[2] A function $f : \mathcal{X} \to \mathbb{R}$ is concave if $-f$ is convex.

$\mathbf{x} = (x_0, \ldots, x_{n-1})$, a MILP can be stated in a general form as:

$$\begin{aligned} \min_{\mathbf{x}} \ & \mathbf{c}^T\mathbf{x} + d \\ s.t \ & \mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq} \\ & \mathbf{A}_{in}\mathbf{x} \leq \mathbf{b}_{in} \\ & x_i \in \mathbb{Z}, \quad i \in \mathcal{I} \subseteq \{0, \ldots, n-1\}, \end{aligned}$$

where n, $\mathbf{c}$, d, $\mathbf{A}_{eq}$, $\mathbf{b}_{eq}$, $\mathbf{A}_{in}$ and $\mathbf{b}_{in}$ are defined as in subsection 5.2.1 and $\mathcal{I}$ is the set of indexes of variables subject to the integrality constraint (and is, of course, a subset of all problem's variables). If $\mathcal{I} \equiv \{0, \ldots, n-1\}$, i.e. all variables are constrained to take up integer values, then the problem is usually referred to simply as *integer linear program* (ILP).

It is very important to notice that, adding these integrality constraints, the feasible set 'shrinks'. That is to say that the feasible set of the MILP (as well as ILP) problem is contained into the feasible set of its so-called *LP relaxation*, i.e. the corresponding LP problem where integrality constraints have been removed. As a consequence, the optimal solution of the LP may not be attainable any longer from the corresponding MILP, whose solution can only be less optimal (lower for a maximization problem and greater for a minimization one) than the former. The LP relaxation's solution is, therefore, a bound to the solution of a MILP. This fact is used in many algorithms for the resolution of MILPs (and, again, ILPs, in general).

The introduction of integrality constraints also brings about a great deal of troubles, due to the loss of a fundamental property of the feasibility set: its convexity. The convexity property of the feasible set guarantees that, when moving in its inside - as an example, when applying the simplex method - one does not go outside of the region and, therefore, the solution keeps being acceptable. When forcing a variable to be integer, this nice property vanishes due to the fact that taking convex combinations involving integer values results in having real values, too. In principle, as a consequence, an exhaustive research throughout all possible feasible solutions has to be performed. Such an endeavour gets easily out of control when the problem size grows bigger and ILPs can be proven to be NP-complete[3] problems.

Many exact algorithms are today known for the solution of MILP problems, as well as some heuristics. One very common exact algorithm is the **branch and cut method**, which suitably combines the branch and bound method and the cutting-plane method.

Cutting-plane method

Cutting plane methods refer to a variety of optimization methods that iteratively refine a feasible set of an objective function through linear equalities, called *cuts*. Such methods

[3] A problem is NP-complete if it is both in NP and NP-hard. The NP-complete problems represent the hardest problems in NP and the time required to solve the problem using any currently known algorithm increases rapidly as the size of the problem grows. As a consequence, determining whether it is possible to solve these problems quickly, called the P versus NP problem, is one of the fundamental unsolved problems in computer science today.

were introduced with the precise aim of solving ILPs and then coupled with the branch and bound method due to the effectiveness of the combination. Nowadays, *Gomory's cuts* are widely used by commercial solvers and the algorithm that uses them consist in the following steps:

1. $\forall i \in \mathcal{I}$ drop x_i's integrality constraint and solve the given LP relaxation, whose solution will lie on the vertex of the feasible set;

2. if the found solution does not satisfy the integrality constraints, then find a hyperplane having the solution's vertex on one side and all other feasible integer points on the other;

3. add the newly found hyperplane is added as an additional linear constraint to the problem, in order to cut the vertex out;

4. repeat the process until an optimal integer solution is found.

Branch and bound method

The branch and bound method consists - at least in principle - in an exhaustive search of the optimal solution in a state space rooted tree, where all possible solutions are enumerated. The algorithm explores branches of such tree or 'prunes' them based on lower (or upper, for maximization) bound estimates. The fundamental steps for finding the optimal solution $\mathbf{x}^*$ are the following:

1. set the value of the incumbent solution ν^*, i.e. the best integer solution found so far, to $+\infty$ and initialize the set of open subproblems $\mathcal{P}$;

2. if $\mathcal{P}$ is not empty, select a subproblem from the search tree and solve its LP relaxation (relaxing the integrality constraint only on variables whose value has not been fixed yet), being $\overline{\mathbf{x}}$ its solution and β its corresponding objective function value, serving as lower bound for the branch subproblems;

3. if $\overline{\mathbf{x}}$ is feasible (integrality constraints included), prune all the subproblems belonging to its branch; morover, if $\overline{\mathbf{x}}$'s objective function value is lower than ν^*, update the incumbent solution $\mathbf{x}^* \leftarrow \overline{\mathbf{x}}$ and its value $\nu^* \leftarrow \beta$. Go back to 2.

4. if the LP relaxation is infeasible, prune the problem and its whole branch. Go back to 2.

5. if $\beta > \nu^*$ prune the problem and its whole branch of subproblems. Go back to 2.

6. replace in $\mathcal{P}$ the problem with subproblems obtained by further partitioning its feasible set. Go back to 2.

Issues like which variable to branch on, which subproblem choose from $\mathcal{P}$ and how to partition a feasible set are disregarded and there is not a unique answer to them. Solutions are usually a matter of the solvers' implementation.

Branch and cut method

The branch and cut method combines the previous two approaches, using, at each iteration the cutting plane method to disregard non-feasible solutions and the branch and bound algorithm to reduce the search space cutting off unuseful problems. The fundamental steps for finding the optimal solution $\mathbf{x}^*$ are the following:

1. set the value of the incumbent solution ν^*, i.e. the best integer solution found so far, to $+\infty$ and initialize the set of open subproblems $\mathcal{P}$;
2. if $\mathcal{P}$ is not empty, select a subproblem from the search tree;
3. solve its LP relaxation, being $\overline{\mathbf{x}}$ its solution and β its corresponding objective function value, serving as lower bound for the branch subproblems;
4. if $\overline{\mathbf{x}}$ is feasible (integrality constraints included), prune all the subproblems belonging to its branch; morover, if $\overline{\mathbf{x}}$'s objective function value is lower than ν^*, update the incumbent solution $\mathbf{x}^* \leftarrow \overline{\mathbf{x}}$ and its value $\nu^* \leftarrow \beta$. Go back to 2.
5. if the LP relaxation is infeasible, prune the problem and its whole branch. Go back to 2.
6. if $\beta > \nu^*$ prune the problem and its whole branch of subproblems. Go back to 2.
7. if $\overline{\mathbf{x}}$ has a non-integer component which is supposed to be integer, search for cutting planes and, if any are found, add them to the LP relaxation. Go back to 3
8. replace in $\mathcal{P}$ the problem with subproblems obtained by further partitioning its feasible set. Go back to 2.

Like in the previous method, issues like which variable to branch on, which subproblem choose from $\mathcal{P}$ and how to partition a feasible set are a matter of the solvers' implementation.

5.2 Implementation

The variables introduced in section 3.1 to formalise the problem are supposed to take up only integer values. This would lead to a integer linear program with at least $4 \times Z \times T$ variables, which, having $Z = 276$ means $1104 \times T$ variables. This number may therefore be 1104 if $T = 1$ and range up to 11040 variables if $T = 10$! That is definitely too many variables for an ILP to be solved in reasonable time. To cope with this great numerical effort, three different methods have been developed:

1. a pure LP, where all variables' integrality constraints have been dropped. A post-processing step follows to get an integer valued relocation vector;
2. a MILP, where the integrality constraints have been kept only on the relocation variables r_{ti}, while relaxing them on all other variables;

3. a combination of 1 and 2, where the pure LP is first solved and the r_{ti} whose optimal value is found to be 0 are fixed, leaving the remaining ones as the integer constrained variables for the MILP in 2.

The rationale behind these choices is that increasing the number of integer constrained variables adds a great deal of numerical effort to the resolution algorithm of an optimization program. LPs are usually fairly fast to be solved, even with a large number of variables, while MILPs are definitely not. Integer constraints on all variables other than relocation ones have been dropped in 2 in order to ease the computational effort. Finally, the number of integer constrained variables has further been reduced in 3 to speed up the algorithm all the more.

The optimisation program is the first step in the resolution algorithm and it is solved at the beginning of each new round.

5.2.1 Linear program and solution rounding

The first of the three methods involves the resolution of a linear program, with no integrality constraint on any of the variables introduced in section 3.1. One more set of variables has been introduced in the problem formulation:

- $\mathbf{w} = (w_{ti})_{t\in\mathcal{T}_p, i\in\mathcal{Z}}$, $w_{ti} \in \mathbb{R}^+$ represents the positive part[4] of the corresponding relocation variable: $w_{ti} = r_{ti}^+$.

[4]The positive part of a real-valued function $f : \mathcal{X} \to \mathbb{R}$ is defined by the formula

$$f^+(x) = max(f(x),\ 0) = \begin{cases} f(x) & if\ f(x) \geq 0 \\ 0 & if\ f(x) < 0 \end{cases} \quad \forall x \in \mathcal{X}$$

The linear program has been formulated in the following way:

$$\max \quad \sum_{t\in\mathcal{T}_p}\sum_{i\in\mathcal{Z}} EO_{ti} \tag{5.2}$$

$$s.t. \quad s_{ti} = s_{t-1,i} + EI_{t-1,i} - EO_{t-1,i} + r_{t-1,i} \quad \forall t \in \mathcal{T}_p\backslash\{p\}, \forall i \in \mathcal{Z} \tag{5.3}$$

$$0 \leq EI_{ti} \leq I_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.4}$$

$$0 \leq EO_{ti} \leq O_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.5}$$

$$\sum_{i\in\mathcal{Z}} r_{ti} = 0 \quad \forall t \in \mathcal{T}_p \tag{5.6}$$

$$w_{ti} \geq r_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.7}$$

$$w_{ti} \geq 0 \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.8}$$

$$\sum_{i\in\mathcal{Z}} w_{ti} \leq R_{max} \quad \forall t \in \mathcal{T}_p \tag{5.9}$$

$$\sum_{i\in\mathcal{Z}} s_{ti} = N \quad \forall t \in \mathcal{T}_p \tag{5.10}$$

$$s_{ti} \geq 0 \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.11}$$

$$-r_{ti} \leq s_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.12}$$

$$s_{ti} + \frac{EI_{ti} + r_{ti}}{2} \geq EO_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.13}$$

$$EI_{ti} = \sum_{j\in\mathcal{Z}} p^t_{ji} EO_{tj} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \tag{5.14}$$

(5.2) is the objective function, representing the total number of outgoing trip requests from all city zones during all the time-frames in the look-ahead period $\mathcal{T}_p$, which has to be maximised.

(5.3) is the transition equation of a city zone from the current state at the beginning of time-frame $t-1$ to the future state at the beginning of time-frame t, taking into account the incoming and outgoing trips carried out and the relocated vehicles from the operator during time frame $t-1$.

(5.4) gives an upper and lower bound on the number of vehicles entering each zone during each time-frame, which has to be obviously a positive value but cannot exceed the incoming trip demand. (5.5) does the same with the number of vehicles leaving each zone during each time-frame.

(5.6) forces the relocation during each time-frame to sum up to 0: as a negative value of r_{ti} means that vehicles are taken away from zone i and a positive value of r_{ti} means that they are brought to zone i, this constraint ensures that, during every time-frame, the total number of vehicles taken away from all zones equals the total number of vehicles brought to all zones.

Constraints (5.7) and (5.8) make sure that w_{ti} represents the positive part of r_{ti}: being the solution on the edge of the feasible set, w_{ti} will be either 0 or r_{ti} and greater than or equal to both of them.

(5.9) ensures that the total relocation during each time-frame does not exceed the maximum value R_{max}. $\mathbf{w}$ instead of $\mathbf{r}$ is used in this constraint since only positive (or

negative) values of $\mathbf{r}$ need to be taken into account for the number of vehicles moved from the operator to be calculated (using $\mathbf{r}$ would yield a zero summation, as forced in (5.6)).

Constraint (5.10) forces the total number of vehicles in the system to be equal to N and constraint (5.11) the number of vehicles in each zone to be always a non-negative value.

(5.12) is an availability constraint, ensuring that the number of vehicles to be taken away from a zone does not exceed the zone initial vehicle availability. This constraint only makes sense if $r_{ti} < 0$, i.e. if vehicles need to be taken away, otherwise, if $r_{ti} \geq 0$, it boils down to (5.11), meaning that any number of vehicles can be brought to a zone, no matter its initial availability.

Constraint (5.13) gives a further upper bound to the number of vehicles leaving each zone during each time frame, which cannot be higher than its corresponding initial availability plus the average number of vehicles arriving and being relocated there. This constraint, therefore, takes into account the trip flows within each time-frame, too.

Finally, (5.14) introduces a probabilistic constraint to the problem, providing a balance between the incoming and outgoing vehicles in each city zone. The number of vehicles moving from zone i to zone j has to reflect the estimated probability from the data on that flow to avoid myopic behaviours of the model, which would otherwise find a solution that would not reflect the truth by unrealistically enhancing convenient flows to maximize the number of satisfied trips.

The problem was solved making use of PuLP[5], which employs the simplex algorithm to find the optimal solution.

The resulting relocation solution, i.e. the variables r_{ti} $t \in \mathcal{T}_p$ $i \in \mathcal{Z}$, is afterwards processed to take up only integer values, consistently with the reality, being impossible, of course, for the operator to move half a car. Since the rounded solution still has to satisfy constraints (5.6) and (5.9), the variables r_{ti} have to be processed grouped by time frame. Fixed $\tilde{t} \in \mathcal{T}_p$, the rounding algorithm is therefore structured in the following steps:

1. compute $\hat{r}_{\tilde{t}i}$ by rounding $r_{\tilde{t}i}$ to the nearest integer $\forall i \in \mathcal{Z}$;
2. compute *pos_rel*, the list of indexes of entries which have been rounded up and such that $\hat{r}_{\tilde{t}i} > 0$;
3. compute *neg_rel*, the list of indexes of entries which have been rounded down and such that $\hat{r}_{\tilde{t}i} < 0$;
4. sort *pos_rel* and *neg_rel* in descending order by the magnitude of the corresponding value's rounding, i.e. sort from indexes corresponding to values which have been rounded the most to those ones which have been rounded the least;
5. compute
   $pos\_excess \leftarrow \sum_{i:\hat{r}_{\tilde{t}i}>0} \hat{r}_{\tilde{t}i} - R_{max}$,

[5]PuLP is a free open source software written in Python. It is used to describe optimisation problems as mathematical models. Full documentation is available at https://coin-or.github.io/pulp/technical/index.html.

taking into account only positive values of relocation, the amount of relocated vehicles by which the rounded solution exceeds the limit R_{max} (which is no more guaranteed by constraint (5.9) because of the rounding step);

6. likewise, compute
   $neg\_excess \leftarrow \sum_{i:\hat{r}_{\tilde{t}i}<0} \hat{r}_{\tilde{t}i} - R_{max}$,
   taking into account only negative values of relocation;

7. remove the excess incoming relocated vehicles:
   while $pos\_excess > 0$ **do**
   if len(pos_rel)>0 **then**
   $index \leftarrow pos\_rel.pop(0)$[6]
   else
   $index \leftarrow sample(\{0, \ldots, Z-1\})$
   end if
   $\hat{r}_{\tilde{t},index} \leftarrow \hat{r}_{\tilde{t},index} - 1$
   $pos\_excess \leftarrow pos\_excess - 1$
   end while
   namely, identify a zone index whose relocation value has to be decreased by one unit until no more excess is reached. The index is identified by looking up in the *pos_rel* list, containing indexes of the rounded entries of the original continuous solution; when the list is emptied, the indexes are drawn randomly from the set of all zones;

8. with an analogous procedure, remove the in excess outgoing relocated vehicles:
   while $neg\_excess > 0$ **do**
   if len(neg_rel)>0 **then**
   $index \leftarrow neg\_rel.pop(0)$
   else
   $index \leftarrow sample(\{0, \ldots, Z-1\})$
   end if
   $\hat{r}_{\tilde{t},index} \leftarrow \hat{r}_{\tilde{t},index} + 1$
   $neg\_excess \leftarrow neg\_excess - 1$
   end while

9. compute the potential relocation unbalance between incoming and outgoing vehicles, i.e. between positive and negative $\hat{r}_{\tilde{t}i}$ values:
   $unbalance \leftarrow \sum_{i:\hat{r}_{\tilde{t}i}>0} \hat{r}_{\tilde{t}i} + \sum_{i:\hat{r}_{\tilde{t}i}<0} \hat{r}_{\tilde{t}i}$

10. if $unbalance > 0$, the solution involves more incoming than outgoing vehicles during time frame $\tilde{t}$; so, the former ones have to be rebalanced:

[6] Given a data-structure implemented as a list, the *pop* operation removes and returns the element of the index corresponding to the argument of the function. If no argument is provided, it usually removes the first element.

while *unbalance*>0 **do**
  if len(*pos_rel*)>0 **then**
    $index \leftarrow pos\_rel.pop(0)$
  else
    $index \leftarrow sample(\{0, \dots, Z-1\})$
  end if
  $\hat{r}_{\tilde{t},index} \leftarrow \hat{r}_{\tilde{t},index} - 1$
  $unbalance \leftarrow unbalance - 1$
end while

namely, reduce the unbalance with the same procedure described in 7 until no more unbalance is left;

11. if *unbalance*< 0, the solution involves less incoming than outgoing vehicles during time frame $\tilde{t}$; so, the latter ones have to be rebalanced:

while *unbalance* > 0 **do**
  if len(*neg_rel*)>0 **then**
    $index \leftarrow neg\_rel.pop(0)$
  else
    $index \leftarrow sample(\{0, \dots, Z-1\})$
  end if
  $\hat{r}_{\tilde{t},index} \leftarrow \hat{r}_{\tilde{t},index} + 1$
  $unbalance \leftarrow unbalance + 1$
end while

using an analogous procedure as the one described before.

The rationale behind this procedure is to get an integer solution starting from the optimal solution provided by the LP problem. The original solution is forced to respect, among the others, constraints (5.6) and (5.9), while a naively rounded solution does - in general - lose these features, which are necessary for the solution to remain acceptable and reasonable. As an example, if $\sum_{i:\hat{r}_{\tilde{t}i}>0} \hat{r}_{\tilde{t}i} > \sum_{i:\hat{r}_{\tilde{t}i}<0} \hat{r}_{\tilde{t}i}$, the solution would involve more incoming than outgoing vehicles to be relocated, meaning that a car would magically appear in the next state of the system, altering the mass conservation constraint in (5.10).

The presented algorithm is therefore used, in order to obtain a feasible integer relocation solution as near as possible to the original continuous one in the ℓ^1 norm[7], while still respecting constraints (5.6) and (5.9), and avoiding a very expensive brute force search in the space of all possible relocation solutions.

[7]Given a vector $\mathbf{x} \in \mathbb{R}^n$, its ℓ^1 norm can be computed as

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i|$$

The idea lying behind the algorithm is to identify what indexes have been rounded up the most for positive $\hat{r}_{\tilde{t}i}$ and rounded down the most for negative $\hat{r}_{\tilde{t}i}$ and subsequently taking relocation units away in order to draw the sum below the maximum limit R_{max} and balance the positive and negative components.

5.2.2 Mixed integer linear program

The second method involves the resolution of a mixed integer linear program, keeping the integrality constraint only on the relocation variables r_{ti} among all the variables introduced in section 3.1. The formulation of the problem is totally analogous to that of subsection 5.2.1, with the only difference of the presence of the integrality constraints.

$$
\begin{aligned}
\max \quad & \sum_{t\in\mathcal{T}_p}\sum_{i\in\mathcal{Z}} EO_{ti} \\
s.t. \quad & s_{ti} = s_{t-1,i} + EI_{t-1,i} - EO_{t-1,i} + r_{t-1,i} \quad \forall t \in \mathcal{T}_p\backslash\{p\}, \forall i \in \mathcal{Z} \\
& 0 \le EI_{ti} \le I_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& 0 \le EO_{ti} \le O_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& \sum_{i\in\mathcal{Z}} r_{ti} = 0 \quad \forall t \in \mathcal{T}_p \\
& w_{ti} \ge r_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& w_{ti} \ge 0 \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& \sum_{i\in\mathcal{Z}} w_{ti} \le R_{max} \quad \forall t \in \mathcal{T}_p \\
& \sum_{i\in\mathcal{Z}} s_{ti} = N \quad \forall t \in \mathcal{T}_p \\
& s_{ti} \ge 0 \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& -r_{ti} \le s_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& s_{ti} + \frac{EI_{ti} + r_{ti}}{2} \ge EO_{ti} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& EI_{ti} = \sum_{j\in\mathcal{Z}} p^t_{ji} EO_{tj} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z} \\
& r_{ti} \in \{-R_{max}, \dots, 0, \dots, R_{max}\} \quad \forall t \in \mathcal{T}_p, \forall i \in \mathcal{Z}
\end{aligned} \tag{5.15}
$$

As stated before, (5.15), the only additional constraint, requires the relocation variables to take up only integer values in the range $[-R_{max},\ R_{max}]$.

The problem was solved making use of MIP[8], which employs the branch-and-cut algorithm to find the optimal solution.

[8]Python-MIP is a collection of Python tools for the modeling and solution of Mixed-Integer Linear programs. The package also provides access to advanced solver features like cut generation, lazy constraints, MIP starts and solution pools. Full documentation is available at https://docs.python-mip.com/en/latest/index.html.

No post-processing of the solution is required here, since the relocation variables representing the operator decision already take up integer values.

5.2.3 Linear program + mixed integer linear program

The last of the three methods involves the resolution of a linear program as stated in subsection 5.2.1 and the subsequent resolution of the mixed integer linear program stated in subsection 5.2.2.

After solving the LP problem, the variables r_{ti} whose optimal value is found to be zero or nearly zero (with a tolerance of 10^{-5}), i.e. $|r_{ti}| < 10^{-5}$, have been fixed to 0 in the MILP problem.

The idea of this approach is to reduce the number of integer constrained variables, which are responsible for the tremendous computational effort to solve the problem, by introducing a little extra computational effort solving the very fast LP. The resolution of the LP can be regarded as one pre-processing step of the variables, whose value is fixed a priori, with the actual resolution of the problem done through the MILP.

Also in this case no post-processing of the solution is required here, since the relocation variables representing the operator decision already take up integer values.

Chapter 6

Neighbourhood search and simulated annealing

The following chapter provides an insight on the latter stage of the optimisation procedure to find the optimal relocation strategy. Given the solution found through the optimisation programs described in chapter 5, a neighbourhood search step including the simulated annealing strategy may be performed to further improve the quality of the solution.

A theoretical background is first provided on neighbourhood search algorithms and on the simulated annealing approach. The mathematical implementation of these tools for the resolution of the problem is then described. Finally, a comment on the parallelisation procedure followed to speed up the execution of the search is provided in subsection 6.2.4.

6.1 Theoretical background

6.1.1 Neighbourhood search

Neighbourhood search algorithms are a class of metaheuristic methods for solving global optimisation problems. When a complete search on the solution space is infeasible, this type of algorithms are sometimes resorted to.

The basic idea behind this class of metaheuristics can be summed up in the following steps:

1. define one (or, possibly, more than one) neighbourhood structure $\mathcal{N}(x)$ of any possible solution $x \in \mathcal{S}$, where $\mathcal{S}$ is the solution space, and $\mathcal{N}(x) \subseteq \mathcal{S}$;

2. let $x \in \mathcal{S}$ be the current incumbent solution;

3. choose, according to some criterion, $y \in \mathcal{N}(x)$ and evaluate its performance;

4. accept or reject y, according to the specific algorithm rules; if y is accepted as the new incumbent solution, then $x \leftarrow y$. Go back to 3.

How to pick a new neighbour and how to decide whether or not a new solution should be accepted as the incumbent one is a matter of the specific algorithm chosen to solve the problem.

A stopping criterion is usually based on a maximum number of iterations, which is fixed a priori.

Many neighbourhood search algorithms implement much more articulated procedures than the one described above: the variable neighbourhood search, as an example, employs many neighbourhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_k$ such that, given $x \in \mathcal{S}$, $\mathcal{N}_1(x) \subseteq \mathcal{N}_2(x) \subseteq \cdots \subseteq \mathcal{N}_k(x)$ to escape from local optima.

6.1.2 Simulated annealing (SA)

Simulated annealing is a probabilistic approach for determining the global optimum of a function with possibly several local optima. More specifically, it is a metaheuristic approach for global optimisation when dealing with a very large search space and is often used when such search space is discrete.

The idea behind this algorithm is to emulate the metallurgic physical process of a solid being steadily cooled in a controlled manner until its structure is frozen at the lowest energy configuration possible. As the solution space is visited, the concept of slow cooling employed in the simulated annealing method is implemented as a gradual decrease in the probability of accepting poorer solutions, instead of only improving ones, like other algorithms do. Accepting worsening solutions enables for a more thorough search of the solution space and allows the solution to escape from local minima, unlike other algorithms such as gradient descent. After defining a proper neighbourhood structure and a quality function to measure how good a solution is, given the current state of the system, the SA algorithm consists in choosing one of its neighbours at random. If the quality function computed with the given neighbour improves, then the latter is chosen as the new incumbent solution, like in other neighbourhood search algorithms. On the contrary, if the quality measure does not improve, the neighbour may still be chosen as the new incumbent solution with a given probability, depending on how big the quality decrease is and on the current temperature of the system. Let $\mathcal{S}$ be the state space, $x^* \in \mathcal{S}$ the incumbent solution, $\mathcal{N}(x^*) \subseteq \mathcal{S}$ its neighbourhood, $\overline{x} \in \mathcal{N}(x^*)$ a new randomly chosen candidate solution and $J : \mathcal{S} \to \mathbb{R}$ the quality measure function of a solution. Assuming to be dealing with a maximisation problem (therefore J has to be maximised), the generic t-th iteration of the SA algorithm involves the following steps:

1. if $J(\overline{x}) \geq J(x^*)$, then accept $\overline{x}$ as the new incumbent solution: $x^* \leftarrow \overline{x}$;
2. if $J(\overline{x}) < J(x^*)$, then accept $\overline{x}$ as the new incumbent solution with probability $exp\{-(J(x^*) - J(\overline{x}))/T(t)\}$.

The stopping criterion is usually based on a maximum number of iterations to be performed. $T : \mathbb{N} \to \mathbb{R}^+$ is the cooling schedule function, which maps the current iteration number to the temperature of the system, and Bertsimas and Tsitsiklis [1993] pointed out its choice impacts the quality of the obtained solution. Some popular choices for the temperature cooling function are:

1. the *linear schedule*: $T(t) = T_0 - \eta t$;
2. the *exponential schedule*: $T(t) = T_0 \alpha^t$;
3. the *logarithmic schedule*: $T(t) = \frac{c}{log(t+d)}$ (usually $d = 1$);
4. the *constant thermodynamic schedule*: T such that $\frac{dT(t)}{dt} = \frac{-v_s T}{\epsilon\sqrt{C}}$, where C is the heat capacity and ϵ is the relaxation time of the system.

T_0 is the initial temperature of the system and the parameter c in the logarithmic schedule can be thought of as initial temperature, as well. The initial temperature, along with all the other parameters need to be properly tuned.

6.2 Implementation

This section deals with the practical implementation of the aforementioned tools for the practical resolution of the problem.

Given a starting solution provided by the resolution of the linear program, like described in chapter 5, a neighbourhood search step implementing the simulated annealing strategy may be included to further improve the quality of the solution.

A neighbourhood structure formal definition is first provided in subsection 6.2.1 and successively employed in the neighbourhood search algorithm described in subsection 6.2.2, together with the cooling schedule applied for the simulated annealing strategy. Two alternative quality measure functions can be used to compare different relocation solutions and they are described in subsection 6.2.3. Finally, a comment on the parallel implementation of this algorithm is made in subsection 6.2.4.

6.2.1 Neighbourhood structure definition

First of all, a neighbourhood structure of a given solution $\mathbf{r} = (r_{ti})_{t\in\mathcal{T}_p, i\in\mathcal{Z}}$ needs to be defined. A neighbour of $\mathbf{r}$ may be obtained - in principle - by altering some entries of $\mathbf{r}$. However, since any solution has to follow some constraints, i.e.,

$$\sum_{i\in\mathcal{Z}} r_{ti} = 0 \quad \forall t \in \mathcal{T}_p \tag{6.1}$$

$$\sum_{\substack{i\in\mathcal{Z} \\ (t,i):r_{ti}>0}} r_{ti} \leq R_{max} \quad \forall t \in \mathcal{T}_p, \tag{6.2}$$

some special care needs to be taken. To keep the solution feasible after some alteration, multiple neighbourhood structures can be defined, one for each time frame. Given $\tilde{t} \in \mathcal{T}_p$, we can define

$$\begin{aligned}\mathcal{N}_{\tilde{t}}(\mathbf{r}) = \{\hat{\mathbf{r}} = (\hat{r}_{ti})_{t\in\mathcal{T}_p, i\in\mathcal{Z}} \mid \hat{r}_{ti} = r_{ti}\ \forall t \neq \tilde{t},\ \forall i \in \mathcal{Z} \text{ and} \\ \exists!\hat{i} \in \mathcal{Z} : \hat{r}_{\tilde{t}\hat{i}} = r_{\tilde{t}\hat{i}} + 1 \text{ and } \exists!\hat{j} \in \mathcal{Z} : \hat{r}_{\tilde{t}\hat{j}} = r_{\tilde{t}\hat{j}} - 1 \\ \text{and } \hat{r}_{\tilde{t}k} = r_{\tilde{t}k}\ \forall k \neq \hat{i}, \hat{j} \text{ and (6.1) and (6.2)}\}.\end{aligned}$$

That is, $\hat{\mathbf{r}} \in \mathcal{N}_{\tilde{t}}(\mathbf{r})$ if and only if all its entries corresponding to time frames different from $\tilde{t}$ are the same as in $\mathbf{r}$; concerning time frame $\tilde{t}$, only two entries $\hat{i}$ and $\hat{j}$ can differ from the original solution: $\hat{r}_{\tilde{t}\hat{i}}$ has one unit more than the original $r_{\tilde{t}\hat{i}}$ and $\hat{r}_{\tilde{t}\hat{j}}$ has one unit less than $r_{\tilde{t}\hat{j}}$.

One neighbour $\hat{\mathbf{r}} \in \mathcal{N}_{\tilde{t}}(\mathbf{r})$ can be - in principle - produced from $\mathbf{r}$ sampling a couple $(\hat{i}, \hat{j})$, $\hat{i}, \hat{j} \in \mathcal{Z}$ and setting $\hat{r}_{ti} = r_{ti}\ \forall t \in \mathcal{T}_p,\ \forall i \in \mathcal{Z}$, except from $\hat{r}_{\tilde{t}\hat{i}} = r_{\tilde{t}\hat{i}} + 1$ and $\hat{r}_{\tilde{t}\hat{j}} = r_{\tilde{t}\hat{j}} - 1$. However, one more caution needs to be taken when computing the neighbourhood of a solution: not all $(\hat{i}, \hat{j})$ couples produce acceptable vectors. In particular, depending on the sign of $r_{\tilde{t}\hat{i}}$ and $r_{\tilde{t}\hat{j}}$, altering their value may result in exceeding the maximum relocation threshold R_{max}. Let us assume, as an example, $R_{max} = 2$, only one time frame and

$$\mathbf{r} = (2, -1, 0, -1).$$

In this case, both constraints (6.1) and (6.2) are respected. If $\hat{i} = 0$ and $\hat{j} = 1$, then

$$\hat{\mathbf{r}} = (3, -2, 0, -1),$$

and, while constraint (6.1) keeps being respected, (6.2) does not hold any longer.

When constraint (6.2) holds with a strict inequality for the incumbent solution, i.e., when it does not reach the maximum relocation threshold, all (i, j) couples produce acceptable neighbours when chosen to alter the original solution. On the contrary, when constraint (6.2) holds with an equality, i.e., when the maximum relocation threshold is reached, not all couples produce feasible relocation solutions. In particular, given $i \in \mathcal{Z}$, :

- if $r_{\tilde{t}i} \geq 0$, only couples (i, j), $i, j \in \mathcal{Z}$ such that $r_{\tilde{t}j} > 0$ are acceptable;
- if $r_{\tilde{t}i} < 0$, all couples (i, j), $i, j \in \mathcal{Z}$ are acceptable.

6.2.2 Neighbourhood search

Having defined a proper neighbourhood structure, a local search can then be performed. The procedure is articulated in the following steps:

1. start from an incumbent solution $\mathbf{r} \in \mathcal{S}$, where $\mathcal{S}$ is the space of all possible solutions and encodes the relocation strategy provided by the optimisation program, like described in chapter 5; set the iteration number $t \leftarrow 0$;
2. randomly select a time frame $\tilde{t} \in \mathcal{T}_p$;
3. randomly select $\hat{\mathbf{r}} \in \mathcal{N}_{\tilde{t}}(\mathbf{r})$, a neighbour of the incumbent solution;
4. evaluate the quality of $\hat{\mathbf{r}}$ and let $J(\hat{\mathbf{r}})$ be its corresponding value;
5. if $J(\hat{\mathbf{r}}) > J(\mathbf{r})$, accept $\hat{\mathbf{r}}$ as the new incumbent solution: $\mathbf{r} \leftarrow \hat{\mathbf{r}}$. Increase the iteration number: $t \leftarrow t + 1$. Go back to 2;
6. otherwise, if $J(\hat{\mathbf{r}}) \leq J(\mathbf{r})$, accept $\hat{\mathbf{r}}$ as the new incumbent solution with probability $exp\{-(J(\mathbf{r}) - J(\hat{\mathbf{r}}))/T(t)\}$, where T is the chosen cooling schedule. If $\hat{\mathbf{r}}$ is accepted: $\mathbf{r} \leftarrow \hat{\mathbf{r}}$, $t \leftarrow t + 1$. Go back to 2. Otherwise go back to 3.

Two stopping criteria are implemented:

- a first one is based, as usual, on a maximum number of iterations;
- a second one stops the search after a given number of iterations which produced no change in the incumbent solution.

$J : \mathcal{S} \to \mathbb{R}^+$ is the quality measure function of the relocation solution and is described in subsection 6.2.3, while $T : \mathbb{N} \to \mathbb{R}$ is the cooling schedule of the simulated annealing algorithm.

Step 6 of the neighbourhood search algorithm implements the simulated annealing strategy. As stated in subsection 6.1.2, a cooling schedule needs to be chosen when implementing such procedure. After a numerical study, described in Appendix A, the exponential schedule

$$T(t) = 1 \times 0.9^t$$

has been selected, setting $T_0 = 1$ and $\alpha = 0.9$, after investigating the performances of the functions described in subsection 6.1.2.

6.2.3 Solution evaluation

The neighbourhood search algorithm needs a way to compare two solutions and assess which one is better. To do so, a quality measure of a solution $J : \mathcal{S} \to \mathbb{R}$ needs to be properly defined, mapping any relocation strategy to a real value, making a comparison between solutions possible.

Two different quality measures have been implemented:

- the first one is based on the resolution of a linear program giving the best case scenario for the number of satisfied trips, having fixed the initial state of the system and the relocation strategy for each time frame;
- the second one is based on a Monte Carlo simulation of the traffic flows, again, having fixed the initial state of the system and the relocation strategy for each time frame.

Best case scenario quality measure

The first possible quality measure function employs a linear program to get the best case scenario number of satisfied trips. Given the initial state of the system, *initial_state*, and the relocation strategy $\mathbf{r} = (r_{ti})_{t \in \mathcal{T}_p, i \in \mathcal{Z}}$, the algorithm to compute the function value over the time period $\mathcal{T}_p = \{p, p+1, \ldots, p+T-1\}$, $p \in \mathcal{P}$, is articulated in the following steps:

1. set the current state
   $s \leftarrow$ *initial_state*,
   initialise the total satisfied trip counter to 0
   total_trips $\leftarrow 0$
   and set the current time frame to p

$\bar{t} \leftarrow p$

2. while $\bar{t} < p + T$, solve the following LP to get the satisfied demand in the best case scenario

$$\max \quad \sum_{i \in \mathcal{Z}} EO_i \tag{6.3}$$

$$s.t. \quad 0 \leq EI_i \leq I_{\bar{t}i} \quad \forall i \in \mathcal{Z} \tag{6.4}$$

$$0 \leq EO_i \leq O_{\bar{t}i} \quad \forall i \in \mathcal{Z} \tag{6.5}$$

$$s_i + \frac{EI_i + r_{\bar{t}i}}{2} \geq EO_i \quad \forall i \in \mathcal{Z} \tag{6.6}$$

$$EI_i = \sum_{j \in \mathcal{Z}} p_{ij}^{\bar{t}} EO_j \quad \forall i \in \mathcal{Z} \tag{6.7}$$

$$s_i + EI_i - EO_i + r_{\bar{t}i} \geq 0 \quad \forall i \in \mathcal{Z} \tag{6.8}$$

3. round the solution variables $(EO_i)_{i \in \mathcal{Z}}$ and $(EI_i)_{i \in \mathcal{Z}}$ (according to the steps described below):
   $\widehat{EI}, \widehat{EO} \leftarrow round\_solution(EI, EO)$

4. add the number of satisfied trips to the total count
   $total\_trips \leftarrow \sum_{i \in \mathcal{Z}} \widehat{EO}_i$

5. compute the following system state
   $s_i \leftarrow s_i + \widehat{EI}_i - \widehat{EO}_i + r_{\bar{t}i} \quad \forall i \in \mathcal{Z},$

   increase the time frame number
   $\bar{t} \leftarrow \bar{t} + 1$

   and go back to 2.

Concerning the linear program used to obtain the best case traffic flows, (6.3) is the objective function, representing the total number of outgoing satisfied trip requests from all city zones during the current time frame $\bar{t}$, which has to be maximised.

(5.3) is the transition equation of a city zone from the current state at the beginning of time-frame $t-1$ to the future state at the beginning of time-frame t, taking into account the incoming and outgoing trips carried out and the relocated vehicles from the operator during time frame $t-1$.

(6.4) gives an upper and lower bound on the number of vehicles entering each zone during the current time frame $\bar{t}$, which has to be obviously a positive value but cannot exceed the incoming trip demand. (6.5) does the same with the number of vehicles leaving each zone during the current time frame $\bar{t}$.

Constraint (6.6) gives a further upper bound to the number of vehicles leaving each zone during the current time frame $\bar{t}$, which cannot be higher than its corresponding initial availability plus the average number of vehicles arriving and being relocated there. This constraint, therefore, takes into account the trip flows within the time frame, too.

(6.7) introduces a probabilistic constraint to the problem, providing a balance between the incoming and outgoing vehicles in each city zone. The number of vehicles moving from zone i to zone j has to reflect the estimated probability from the data on that flow to avoid myopic behaviours of the model, which would otherwise find a solution that would not reflect the truth by unrealistically enhancing convenient flows to maximise the number of satisfied trips.

Finally, constraint (6.8) makes sure that the following system state has a non-negative value, as it represents the number of vehicles in each zone at the beginning of the following time frame. The problem was again solved making use of PuLP.

Besides, in step 3 of the previous algorithm, a solution rounding step is involved, which is articulated in the following steps:

1. compute $\widehat{EO}_i$ and $\widehat{EI}_i$ by rounding respectively EO_i and EI_i to the nearest integer $\forall i \in \mathcal{Z}$;

2. compute the potential unbalance between incoming and outgoing vehicles, i.e. $unbalance \leftarrow \sum_{i\in\mathcal{Z}} \widehat{EO}_i - \sum_{i\in\mathcal{Z}} \widehat{EI}_i$

3. if $unbalance > 0$, the rounded solution involves more outgoing than incoming vehicles during time frame $\bar{t}$; so, the former ones have to be rebalanced. To do so compute *indexes*, the list of entry indexes which have been rounded up and sort it in descending order by the magnitude of the corresponding value's rounding, i.e. sort from indexes corresponding to values which have been rounded the most to those ones which have been rounded the least; then rebalance $(EO_i)_{i\in\mathcal{Z}}$:

   while $unbalance > 0$ **do**
   if len($indexes$)>0 **then**
   $index \leftarrow indexes.pop(0)$[1]
   else
   $index \leftarrow sample(\{0, \dots, Z-1\})$
   end if
   if $s_{index} - (\widehat{EO}_{index} - 1) + \widehat{EI}_{index} + r_{\bar{t},index} \geq 0$ **then**
   $\widehat{EO}_{index} \leftarrow \widehat{EO}_{index} - 1$
   $unbalance \leftarrow unbalance - 1$
   end if
   end while

   namely, reduce the unbalance until no more is left by identifying a zone index whose outgoing vehicle count has to be decreased by one unit. The index is identified by looking up in the *indexes* list, containing indexes of the rounded entries of the original continuous solution. Observe that the count is decreased only if the following zone state obtained after altering the value of the variable keeps being acceptable, i.e., it

[1] Given a data-structure implemented as a list, the *pop* operation removes and returns the element of the index corresponding to the argument of the function. If no argument is provided, it usually removes the first element.

has a non-negative value. When the list is emptied, the indexes are drawn randomly from the set of all zones;

4. if *unbalance*< 0, the solution involves less outgoing than incoming vehicles during time frame $\bar{t}$; so, the latter ones have to be rebalanced:

   while *unbalance* > 0 **do**
   if len(*indexes*)>0 **then**
   $index \leftarrow indexes.pop(0)$
   else
   $index \leftarrow sample(\{0, \dots, Z-1\})$
   end if
   if $s_{index} - \widehat{EO}_{index} + (\widehat{EI}_{index} - 1) + r_{\bar{t},index} \geq 0$ **then**
   $\widehat{EI}_{index} \leftarrow \widehat{EI}_{index} - 1$
   $unbalance \leftarrow unbalance + 1$
   end if
   end while

   using an analogous procedure as the one described before.

Monte Carlo simulation quality measure

Alternatively to the best scenario measure, the quality of a relocation solution may be computed through a Monte Carlo simulation of the system. To reduce the computational effort needed for the simulations to be performed, every possible solution gets evaluated using the same traffic flow scenarios, whose number is set a priori to MC.

Each traffic flow scenario consists of an ordered list of trips, described by a couple of indexes specifying the departure and arrival city zones. For implementation purposes, a collection of data structures $\{trips_t^k\}_{t\in\mathcal{T}_p}^{k=0,\dots,MC-1}$ has been used, obtained from the array $\mathbf{V}$ described in section 3.1.

$trips_t^k$, $t \in \mathcal{T}_p$, $k = 0, \dots, MC-1$ gives an ordered list of trips carried out during time frame t in scenario k, containing each couple (i, j) V_{ij}^t times, accounting for the number of trips starting in zone i and ending in zone j during time frame t. Each traffic flow scenario is obtained from the previous one by shuffling the list, i.e. $trips_t^k = shuffle(trips_t^{k-1})$ $\forall k$, $\forall t \in \mathcal{T}_p$. Therefore, looping through $trips_t^k$ gives a mobility demand scenario, which is used in the evaluation of a relocation strategy.

Finally, the whole Monte Carlo solution evaluation procedure can be described. Given the initial state of the system, *initial_state*, and the relocation strategy $\mathbf{r} = (r_{ti})_{t\in\mathcal{T}_p, i\in\mathcal{Z}}$, the algorithm to compute the function value over the time period $\mathcal{T}_p = \{p, p+1, \dots, p+T-1\}$, $p \in \mathcal{P}$, consists in the simulation of the system many times and in the subsequent computation of the performance measure for each Monte Carlo run. Finally, the performance measures obtained by all the Monte Carlo runs are averaged to get a more accurate and less uncertain estimate of the actual performance measure value, which is - in general - a random unknown value.

The k-th Monte Carlo run is articulated in the following steps:

1. set the current state

$s \leftarrow initial\_state$,

initialise the total satisfied trip counter to 0
$total\_trips \leftarrow 0$

and set the current time frame to p
$\bar{t} \leftarrow p$;

2. while $\bar{t} < p + T$, initialise the satisfied trip counter to 0:
$satisfied\_trips \leftarrow 0$;

3. compute s^r, the state of the system after the beginning of the relocation process, i.e. after all vehicles have been collected from the operator to be transferred to another zone:
$s_i^r \leftarrow s_i + min\{r_{\bar{t}i}, 0\},\ \forall i \in \mathcal{Z}$;

4. copy s^r to keep track of the zone availability:
$availability \leftarrow copy(s^r)$;

5. looping over the first half of $trips_{\bar{t}}^k$, for each couple (i, j) identifying a trip request, compute the new state of the system and update the zone availability only if the request can be carried out:
if $availability_i > 0$ **then**
$s_i^r \leftarrow s_i^r + 1$
$s_j^r \leftarrow s_j^r - 1$
$availability_i \leftarrow availability_i - 1$
$satisfied\_trips \leftarrow satisfied\_trips + 1$
end if

6. update s^r simulating the redistribution of the vehicles after the ending of the relocation process:
$s_i^r \leftarrow s_i^r + max\{r_{\bar{t}i}, 0\}$;

7. looping over the second half of $trips_{\bar{t}}^k$, for each couple (i, j) identifying a trip request, compute the new state of the system and update the zone availability only if the request can be carried out, like in steps 4 and 5;

8. add the number of satisfied trips during the current time frame $\bar{t}$ to the total count
$total\_trips \leftarrow total\_trips + satisfied\_trips$,

increase the time frame counter
$\bar{t} \leftarrow \bar{t} + 1$,

9. update the current system state
$s \leftarrow s^r$

and go back to 2.

To take the time for the relocation procedure to be carried out into account, the state of the system is first updated in 3 removing from each zone the number of cars to be transferred according to the given strategy. The first half of trip requests is then simulated, modifying the system. Successively, the system state is updated in 6 emulating the redistribution of the relocated cars, making them available for users to rent them. Finally, the second half of trips is simulated.

The variable *total_trips* represents the performance measure of the k-th simulation having undertaken the relocation strategy coded by $\mathbf{r} = (r_{ti})_{t\in\mathcal{T}_p, i\in\mathcal{Z}}$. Running MC different simulations and averaging their *total_trips* values gives the relocation strategy performance measure, making a comparison between solutions possible.

6.2.4 Parallelisation

The described algorithm is highly expensive since a great number of iterations are usually needed to attain an improvement of the solution and one iteration itself is very costly. Specifically, the most computationally expensive procedure is the evaluation of every single strategy, described in subsection 6.2.3. However, given a single strategy, its performance evaluation does not involve any other data except from the initial system state and the mobility data (for what concerns the best case scenario quality measure) or the mobility simulations (for the Monte Carlo measure). This means that a parallel implementation can be worked out. The tool used to do so is the *multiprocessing* library[2]. In step 3 of the neighbourhood search, described in subsection 6.2.2, a random neighbour of the incumbent solution is chosen and then evaluated. Since every evaluation is independent on all the others, it can be carried out in parallel with them. Therefore, a number of neighbours equal to the number of CPUs of the machine running the algorithm can be simultaneously evaluated, speeding up the exploration of the neighbourhood. The search is therefore carried out in simultaneous chunks, whose size is equal to the number of CPUs. When any of the neighbours gets accepted as the new incumbent solution, according to the rules of the simulated annealing, the algorithm simply moves on that new relocation strategy, starting looking in its neighbourhood simultaneously on new candidates. In case more than one neighbour gets accepted during one parallelised chunk, the first one in time which was accepted becomes the new incumbent solution.

[2] *multiprocessing* is a package that supports spawning processes using an API similar to the *threading* module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows. Full documentation is available on https://docs.python.org/3/library/multiprocessing.html.

Chapter 7

Data extraction and analysis

This chapter concerns and details the extraction and processing procedure and the exploration of the data used to confront the relocation problem.

7.1 Data extraction

The data extraction and processing procedure is described in the following section.

7.1.1 The dataset

A dataset of trips carried out in the city of Turin by Car2go, one very well-known car-sharing company operating in many cities all over the world, was initially provided. The data about the trips, carried out by a fleet of 396 vehicles, was collected through a system called Urban Mobility Analisys Platform (UMAP), developed by the SmartData@Polito research group. UMAP queries the Car2go API every minute to get the currently available cars in the operating area and returns each car's number plate, current position (i.e. its latitude and longitude coordinates), current energy level and internal status. Tracking when the cars appear and disappear, the system is able to identify the start and the ending of a booking generating booking events characterised by a number plate (i.e. a car), the initial and final position, time and energy level. Parkings are also tracked, i.e. the time period in which a car is available for users to be taken. The booking distance is approximated by means of the haversine distance[1] and then corrected through a spatial scale factor obtained with the Google Direction API. The energy consumption is the difference between the initial and final fuel level and the duration is the difference between the final and initial time (adjusted with a temporal scale factor obtained again through the Google Directions API) and takes the reservation time into account, as well.

Each record then consists of the following information about a trip:

[1]The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes.

- *plate*: the number plate number of the car employed in the trip;
- *init_time*: the timestamp of the moment the trip began;
- *final_time*: the timestamp of the moment the trip ended;
- *init_lon*: the longitude of the point in which the trip began;
- *init_lat*: the latitude of the point in which the trip began;
- *final_lon*: the longitude of the point in which the trip ended;
- *final_lat*: the latitude of the point in which the trip ended;
- *distance*: the distance between the starting and ending point of the trip in meters;
- *duration*: the duration of the trip in seconds;
- *weekday*: the day of the week the trip was carried out;
- *init_fuel*: the initial value of the vehicle's fuel state;
- *final_fuel*: the final value of the vehicle's fuel state;
- *engine_type*: the type of engine the car is equipped with;
- *init_date*: the date (year and month) the trip began;
- *init_day*: the day, month and year the trip began.

7.1.2 Dataset cleaning

After importing the data as a *pandas*[2] dataset, trips whose duration is below two minutes or over two hours were filtered out, as well as trips whose travelled distance is lower than 100 metres or higher than 40 kilometres. Finally, trips where the initial fuel value is lower than the final fuel value were filtered out, obtaining a dataset with 612,532 records, with trips spanning around one year and one month, being the first one recorded in 2016/12/13 at 17:38 and the last one in 2018/01/31 at 13:08.

7.1.3 Data processing

The data were then processed in order to discretise them both spatially and temporally.

[2] *pandas* is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. See https://pandas.pydata.org/docs/ for full documentation.

Spatial discretization

The spatial variables were processed in order to obtain a grid discretization of the city into zones. Both the longitude and the latitude coordinates were discretised to obtain 285 500-metres-by-500-metres square zones. The zones obtained through this procedure were, then, identified with two discrete coordinates (coming from the discretization of the longitude and latitude coordinates, respectively).

For the sake of simplicity, a 'flattening' procedure of the coordinates was successively performed, in order to associate a single index to each zone. The index, starting from the zone's x and y discrete coordinates, was obtained in the following way:

1. compute min_x, the lowest x coordinate among both departure and arrival zones;
2. compute max_x, the highest x coordinate among both departure and arrival zones;
3. compute min_y, the lowest y coordinate among both departure and arrival zones;
4. compute max_y, the lowest y coordinate among both departure and arrival zones;
5. compute $x\_zones = max\_x - min\_x + 1$, the total number of discrete x-axis values;
6. compute $y\_zones = max\_y - min\_y + 1$, the total number of discrete y-axis values;
7. compute the zone index (for both departure and arrival) as $(y - min\_y) \times x\_zones + (x - min\_x)$, where x and y are the initial coordinates of the zone.

A further cleaning of the dataset was then carried out, filtering out zones whose departure or arrival count was lower or equal than five trips. Nine zones were ruled out with this procedure, reducing their total number to 276.

Finally, a zone re-numbering procedure was performed, in order to have indexes in the range 0-275. In a real-world contex, the initial coordinates of the zones can easily be computed back, but such procedure was not implemented here as there was no need during the work.

Temporal discretization

Successively, a temporal discretization procedure was carried out, in order to obtain time-frames at the beginning of which a new relocation step is to be performed.

Only few data were extracted in this step, covering two working days, as much as was needed for the developed algorithms to be tested.

20 time-frames were extracted, starting from 2016/12/14 at 06:00 and ending with 2016/12/16 at 06:00. Each time frame covers a two-hour period, being a reasonable time for a relocation procedure step to be undertaken and making the discretization not too rough. Indeed, if longer time frames had been chosen, the mobility demand would have been grouped approximately, not reflecting the actual patterns and variations through the day. On the other hand, too short time frames (as an example, 10 minutes) would not be enough to even contain a whole trip. Night hours from 00:00 to 06:00

were actually merged into one six-hour long time-frame to better reflect the operational procedures of the operator. The resulting time-frames' break-points are the following (in the YYYY/MM/DD, HH:MM:SS format) - the bold number in parentheses giving their corresponding position in the data structures employed:

- 2016/12/14, 06:00:00 (**0**)
- 2016/12/14, 08:00:00 (**1**)
- 2016/12/14, 10:00:00 (**2**)
- 2016/12/14, 12:00:00 (**3**)
- 2016/12/14, 14:00:00 (**4**)
- 2016/12/14, 16:00:00 (**5**)
- 2016/12/14, 18:00:00 (**6**)
- 2016/12/14, 20:00:00 (**7**)
- 2016/12/14, 22:00:00 (**8**)
- 2016/12/15, 00:00:00 (**9**)
- 2016/12/15, 06:00:00 (**10**)
- 2016/12/15, 08:00:00 (**11**)
- 2016/12/15, 10:00:00 (**12**)
- 2016/12/15, 12:00:00 (**13**)
- 2016/12/15, 14:00:00 (**14**)
- 2016/12/15, 16:00:00 (**15**)
- 2016/12/15, 18:00:00 (**16**)
- 2016/12/15, 20:00:00 (**17**)
- 2016/12/15, 22:00:00 (**18**)
- 2016/12/16, 00:00:00 (**19**)
- 2016/12/16, 06:00:00

Clearly, the first break-point is just the starting time of the first time-frame and the last break-point is the ending time of the last time-frames, while all other break-points are the ending time of the corresponding previous time-frame and the starting time of the following one. Each trip record was assigned to a time frame based on its *init_time* value, i.e. the time the trip began. That is, one trip beginning during one time frame and ending in the following one gets assigned to the former for time discretization purposes.

7.2 Data aggregation

A data aggregation procedure followed, with the aim of getting the vehicle demand in each zone for each time-frame. Making use of *pandas*, a *group-by* step was then performed, gathering the trips by time-frame, departure zone and arrival zone. A *size* operations was finally executed, returning the number of rows in each group, that is the number of trips operated during each time-frame from a zone to another.

Origin-destination matrix

The aggregated data obtained after this step were then collected in a *numpy*[3] array, whose first axis corresponds to the time-frame, the second axis to the departure zone and the third axis to the arrival zone. Therefore, entry (t, i, j) of the resulting array gives the number of trips from zone i to zone j during time-frame t. The array can be seen as an ordered collection of origin-destination matrices, one for each time-frame.

Origin-destination probability matrix

Starting from each origin-destination matrix, a normalisation by rows of the entries was successively performed: for each row, the sum of the entries was computed, then each entry was divided by such value.

The resulting data were, once again, collected in a *numpy* array, whose structure reflects that of the origin-destination matrix. Therefore, entry (t, i, j) of the resulting array gives the probability of going to zone j starting from zone i during time-frame t. The array can be seen as an ordered collection of origin-destination probability matrices, one for each time-frame.

Incoming and outgoing demand arrays

Starting from each origin-destination matrix, incoming and outgoing demand arrays were also obtained by performing a sum over columns and rows, respectively. In both cases the resulting data were, once again, collected in a *numpy* array: entry (t, i) gives the number of trips starting or ending in zone i during time-frame t.

Figure 7.1 shows the incoming and outgoing demand pattern for each time-frame and for each zone. Blue bars represent the total number of incoming trips into a zone, while red bars display the total number of outgoing trips from a zone. We can usually observe higher peaks on 'central' zones, i.e. zones whose index is in the centre of the range, which correspond to city centre areas.

Figure 7.2 and Table 7.1 portray instead the total (incoming or outgoing) demand for each time-frame. Let's observe that, of course, in each time-frame, the total incoming demand equals the total outgoing demand. Reasonably enough, a lower demand can be observed during evening and night hours (time-frames 7, 8, 17 and 18 corresponding to the hour range 20-24 and time-frames 9 and 19 to the hour range 24-06). The highest peaks in demand can instead be observed in afternoon hours (time-frames 5, 6, 15 and 16 corresponding to the hour range 16-20).

[3] *numpy* is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. See https://numpy.org/doc/stable/ for full documentation.

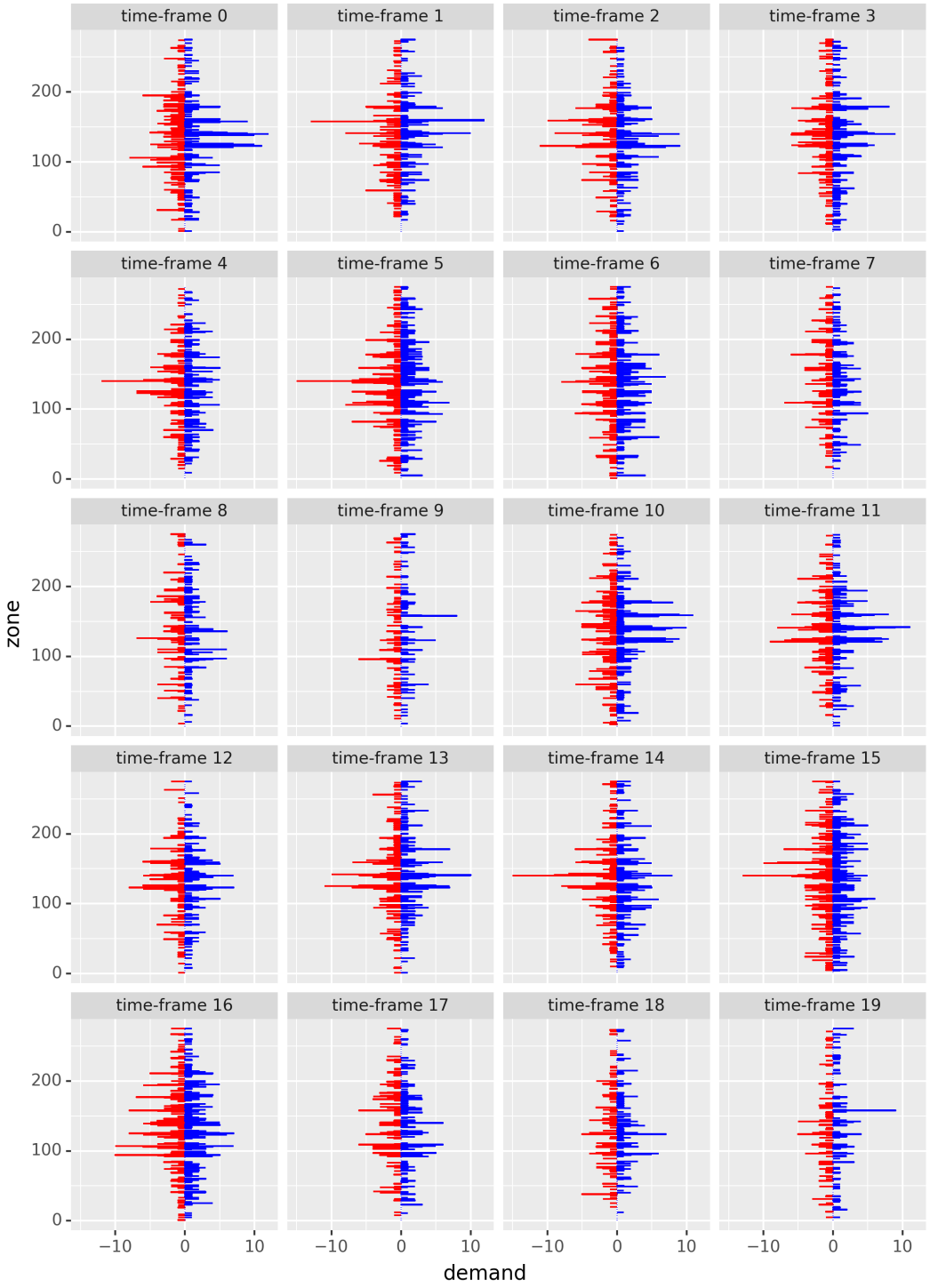

Figure 7.1. Incoming (blue) and outgoing (red) demand by time-frame and zone.

time-frame	0	1	2	3
demand	261	221	227	224
time-frame	4	5	6	7
demand	214	293	275	171
time-frame	8	9	10	11
demand	153	113	257	232
time-frame	12	13	14	15
demand	189	236	234	307
time-frame	16	17	18	19
demand	297	170	142	116

Table 7.1. Total (incoming or outgoing) demand by time-frame.

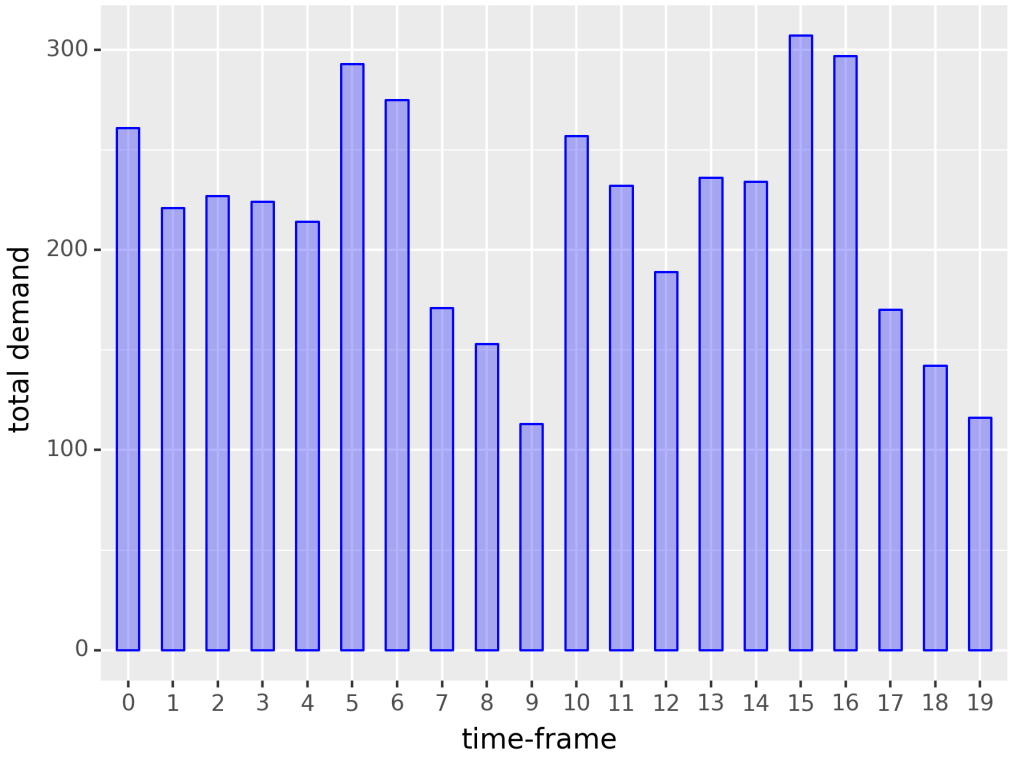

Figure 7.2. Total (incoming or outgoing) demand by time-frame.

Chapter 8

Numerical results

This chapter is devoted to the presentation of the numerical results obtained from the optimisation procedures carried out by means of the algorithms discussed in the previous chapters. The metrics employed to compare the performances of the algorithms are first presented. Then, section 8.2 explores the computational difficulties encountered with the dynamic programming approach and compares its performances with those of the approximated algorithms in a toy case problem. In section 8.3, a comparison of the approximated algorithms is carried out in a real-world situation, using mobility data of the Car2go fleet operating in Turin. Successively, section 8.4 analyses the effect of varying the maximum number of cars which can be relocated within a time frame, with some further comment on systems not implementing a relocation strategy. Finally, section 8.5 provides a simple analysis on the effect of the relocation cost on the profits and the efficiency of the system.

All the experiments have been carried out on a virtual machine belonging to the BigData@Polito cluster[1], maintained through the Hadoop framework and having reserved 24 CPU Threads and 120 GB memory, with maximum 70 CPU Threads and 320 GB memory.

Some parameters need to be fixed before running the experiments: as described in Appendix A, the exponential schedule with $\alpha = 0.9$ and $T_0 = 1$ was set; when solving the mixed integer linear program, the parameter *max_seconds_same_incumbent*[2] was set to 1800 seconds, corresponding to half an hour, to avoid too long runs. Concerning the neighbourhood search, a maximum number of 300 iterations per round was set, together with a maximum number of 15 iterations without any change in the incumbent solution.

Finally, all approximated methods were performed with a look ahead horizon in the set $\{1,\ 2,\ 4,\ 6,\ 8,\ 10\}$, corresponding to, respectively, 2, 4, 8, 12, 16 and 24 hours, with the aim of assessing how long-sighted should be the optimisation strategy in order to attain

[1] https://smartdata.polito.it/computing-facilities/

[2] The parameter *max_seconds_same_incumbent* is the maximum time in seconds that the search for the resolution of the mixed integer linear program, as implemented in the *mip* package, can go on if a feasible solution is available and it is not being improved

the best performances. The optimisation period $\mathcal{P}$ was set to 10 time frames in all runs, corresponding to one operation day. A starting system state needs to be provided to any algorithm to run and that has been set to an uniformly distributed number of vehicles on every city zone.

8.1 The metrics

To compare the algorithms employed to confront the problem and investigate the research questions proposed in section 1.2, a performance measure of the procedures used to obtain the relocation strategies needs to be defined. The main metric employed is the *efficiency* of the system throughout the whole optimisation period, starting with time frame 0 and ending with time frame P, according to the notation introduced in section 3.1. The efficiency of the system can be described by the fraction of trips which could be satisfied out of the total mobility demand on the whole period, that is

$$efficiency = \frac{\sum_{t\in\mathcal{P}}\sum_{i\in\mathcal{Z}} EO_{ti}}{\sum_{t\in\mathcal{P}}\sum_{i\in\mathcal{Z}} O_{ti}}.$$

A higher efficiency clearly means better algorithm performances since the relocation strategy found through the procedure allows more trips to be accomplished.

Together with the system efficiency, the *elapsed time* for the algorithm to be completed is an important performance measure, since, especially in this contex, being able to take fast decisions is very important.

In section 8.5 one more metric is used: the total profits obtained by the operator. Since the relocation process is a costly procedure, carrying it out may actually turn out to be decreasing the overall profits, although boosting the system efficiency. The total profits are computed as

$$total\ profits = f \times \sum_{t\in\mathcal{P}}\sum_{i\in\mathcal{Z}} EO_{ti} - c \times \sum_{t\in\mathcal{P}}\sum_{i\in\mathcal{Z}} r^{+}_{ti}, \tag{8.1}$$

where f is the average earning from a trip and c is the average cost of relocating a vehicle. The first summation is again the total number of trips which could be carried out, while the second summation represents the total number of vehicles moved by the support staff during the relocation procedure. Once again, the positive part of the relocation variables is used since each moved vehicle would otherwise be counted both when it is relocated out of a zone and when it is moved to another one.

8.2 Toy case and optimal solution

In chapter 4, the dynamic programming approach is presented and its implementation is described. This method guarantees to find the optimal solution and is even able to adapt to the system evolution without any need to solve the problem again. Unfortunately, as already observed, it is very little scalable: since any possible state of the system needs to be evaluated and its optimal decision needs to be stored, the problem dimensionality and computational difficulty become soon overwhelming.

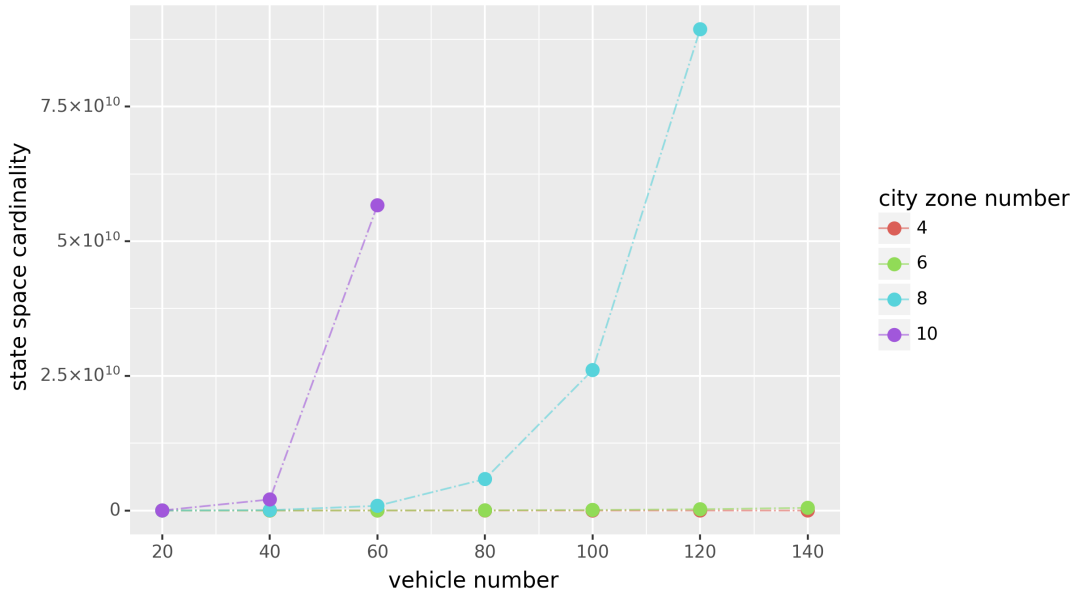

Figure 8.1. State space cardinality by vehicle and city zone number of the system.

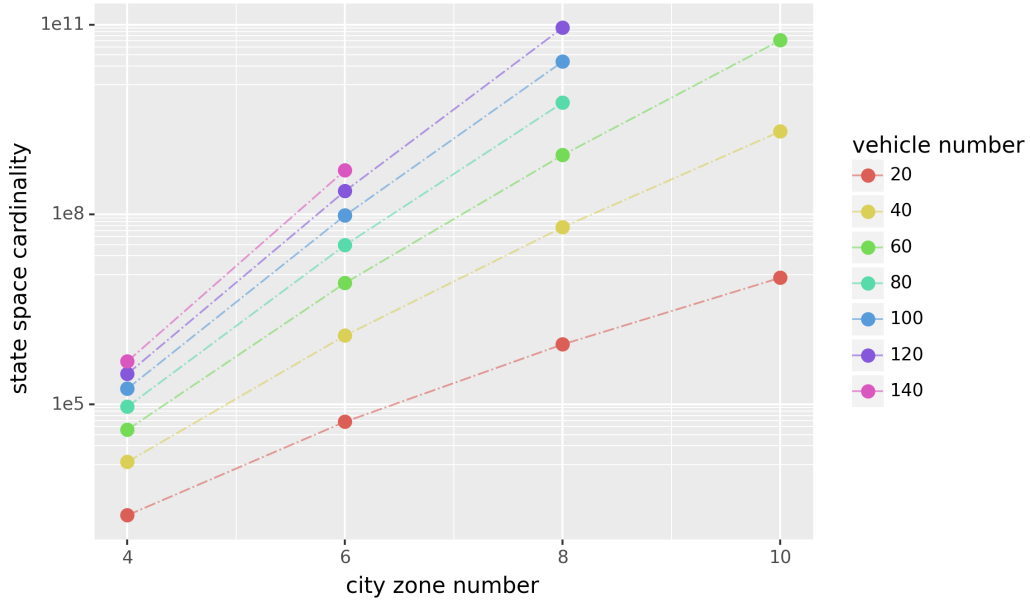

Figure 8.2. State space cardinality by city zone and vehicle number of the system in logarithmic scale.

Figure 8.1 displays the cardinality of the state space against the number of vehicles

efficiency	1	2	4	6	8	10
Dynamic programming	0.961					
LP+MILP & no search	0.865	0.913	0.959	0.942	0.957	0.959
LP+MILP & search & best case	0.913	0.933	0.947	0.945	0.947	0.938
LP+MILP & search & Monte Carlo	0.918	0.953	0.933	0.938	0.949	0.938
LP+rounding & no search	0.894	0.909	0.928	0.933	0.928	0.933
LP+rounding & search & best case	0.900	0.913	0.942	0.945	0.938	0.947
LP+rounding & search & Monte Carlo	0.913	0.952	0.942	0.952	0.938	0.933
MILP & no search	0.865	0.928	0.875	0.933	0.952	0.938
MILP & search & best case	0.894	0.938	0.938	0.946	0.904	0.928
MILP & search & Monte Carlo	0.889	0.956	0.959	0.954	0.954	0.957

Table 8.1. Efficiency results with toy case problem, showing the optimal solution and the approximated solutions performances.

elapsed time **(s)**	1	2	4	6	8	10
Dynamic programming	104.1					
LP+MILP & no search	1.1	1.5	2.7	4.2	7.1	7.9
LP+MILP & search & best case	18.7	27.5	41.5	54.0	73.6	88.5
LP+MILP & search & Monte Carlo	25.2	29.3	62.2	117.3	185.5	244.2
LP+rounding & no search	0.7	1.2	1.6	3.3	3.3	4.4
LP+rounding & search & best case	17.5	24.6	39.6	55.4	71.4	82.2
LP+rounding & search & Monte Carlo	21.2	29.5	56.6	102.0	180.1	266.8
MILP & no search	2.4	1.6	2.9	4.8	5.5	9.9
MILP & search & best case	16.2	25.5	38.2	47.2	53.3	68.9
MILP & search & Monte Carlo	22.1	27.3	60.1	108.7	209.6	309.8

Table 8.2. Elapsed time to run all the algorithms with the toy case problem.

in the system and the number of city zones, showing a computational effort almost exponentially increasing with the number of vehicles, fixing the number of zones. Although the number of vehicles reaches realistic numbers, the number of zones is definitely too small for real-world scenarios and increasing it to around 270, like is needed for the city of Turin, would make solving the problem impossible. Figure 8.2 displays the cardinality of the state space against the number of city zones and the number of vehicles in the system in logarithmic scale, showing the exponential growth of the state space with the number of zones. Besides, the method requires the computation of the reachability set $\mathcal{R}(\mathbf{s})$ of every system state $\mathbf{s}$, a problem whose complexity is $O\left(\frac{n^2}{2}\right)$, where n is the number of system states.

However, the algorithm can still be applied to small dimensionality cases and some comparisons with the approximated algorithms can be made in order to grasp an idea of how good the approximations are, with respect to the optimal solution provided by the

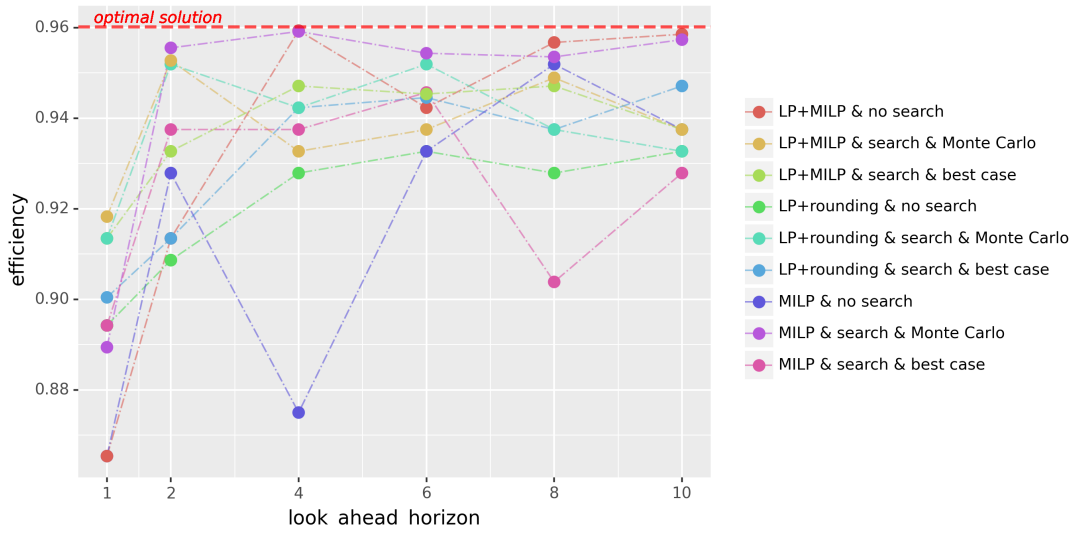

Figure 8.3. Efficiency results with toy case problem, showing the optimal solution and the approximated solutions performances.

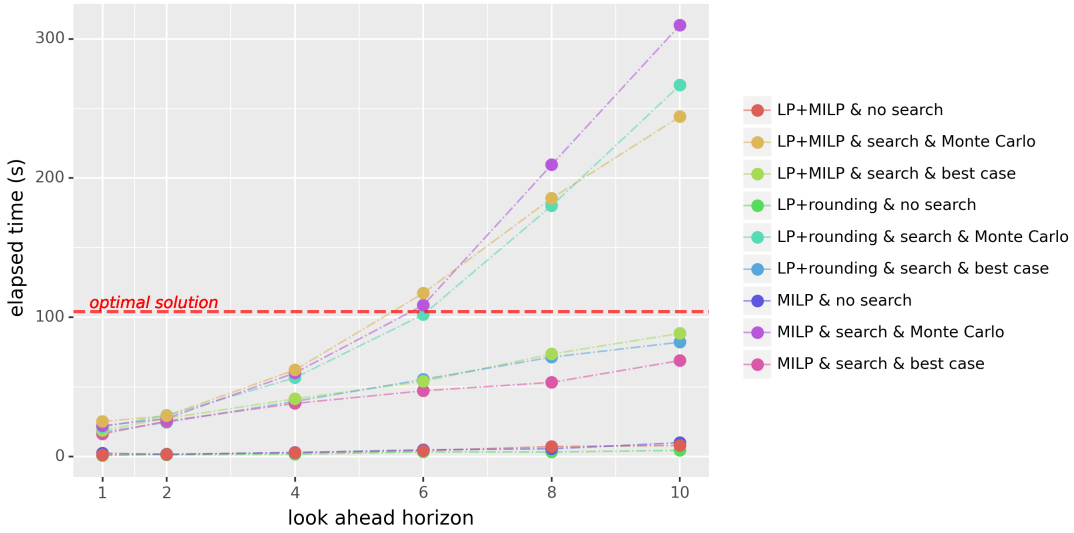

Figure 8.4. Elapsed time to run all the algorithms with the toy case problem.

dynamic programming method. An analysis on a toy case was therefore carried out, fixing the number of vehicles in the system to 20 and working on 4 city zones. Figure 8.3 displays the efficiency results obtained by all the approximated methods run with different look ahead horizons and, in the red dashed line, the efficiency of the optimal result obtained thorough the dynamic programming algorithm. Figure 8.4 shows the required time in seconds to execute the algorithms.

First of all, the advantage of a long-sighted optimisation is already clear in this small dimensionality scenario: increasing the look ahead horizon to just two time frames greatly boosts the efficiency, which, for the greedy approach, is far from the optimal solution provided by the dynamic programming method. The best approximated method turns out to be the mixed integer linear program followed by the neighbourhood search using the Monte Carlo solution quality measure, which is able to attain almost optimal efficiency results with all look ahead horizons, except from the greedy one. However, it is, at the same time, the most computationally expensive method, exceeding the dynamic programming algorithm for look ahead horizons longer than 6.

Figure 8.4 also shows that for this toy problem, the execution time strongly depends on the neighbourhood search step: when it is not performed, the resolution is almost immediate; when it is performed with the best case scenario quality measure, more time is required, but the algorithms keep being faster than the dynamic programming method; instead, comparing the relocation strategies through the Monte Carlo simulation method requires the most time. It is, however, worth noticing that the resolution of the linear program does not weigh on the total running time because, even MILPs, are solvable immediately in such small dimensionality scenarios.

8.3 Comparing approximated methods' performances

Working on real-world problems makes it impossible to use the dynamic programming approach, as already discussed in the previous section. The approximated methods described in chapter 5 and chapter 6 are therefore applied to the case study of the Car2go fleet operating in the city of Turin. The mobility data used to run the algorithms are those described in chapter 7, with 396 vehicles, 276 city zones and a relocation budget $R_{max} = 10$.

Figure 8.5 displays the efficiency results of all the approximated algorithms by the look ahead horizon employed in the optimisation process. There is a crystal clear result: avoiding a greedy strategy favouring a long-sighted approach is able to increase the satisfied demand by anticipating future demand patterns. Increasing the look ahead period boosts the performances, but a plateau is soon reached: indeed, increasing the look ahead from 1 to 2 improves the efficiency by around 7%, with a further improvement of another 3% when the look ahead gets to 10. Observe that a 7 to 10% improvement in efficiency can be obtained using a non-greedy approach just by smartly relocating 10 vehicles per time frame, making up less than 3% of the fleet! The MILP approach seems to be the most performing one, giving good results with all look aheads. The LP+rounding method performs fairly well, showing that the solution post-processing is able to provide quality solutions in very short times, however its performances are lower with respect to the other algorithms. The neighbourhood search seems to be able to do only little improvements to the solution, with the LP+rounding method being the one that benefits the most from using such procedure.

Figure 8.6 shows the time needed for the optimisation algorithms to be run. Short look ahead periods, up to 2 two-hour time frames, do not require great computational

times with any of the algorithms implemented. Unfortunately, starting from a 4 two-hour time frame look ahead horizon the elapsed time starts to become unviable for many algorithms. From a 8 time frame look ahead horizon on, the only algorithm guaranteeing a short execution time is the linear program with no neighbourhood search. The run time of some algorithms is not shown since it so elevated that it exceeded a two-day threshold. The neighbourhood search greatly slows down the optimisation procedure, especially when the Monte Carlo quality measure to compare candidate solutions is used. With the shorter look ahead horizons (up to 4 time frames), the neighbourhood search is actually the main responsible for the total computational time. With the size of the problem increasing (look ahead horizons longer than 6), also solving the mixed integer linear program becomes a computational issue. The pre-processing of the solution carried out through the LP, whose aim is to reduce the problem dimensionality, does indeed speed up the algorithm, but that is not enough to make the resolution of the problem fast enough to be suitable to the shared mobility context.

No sharp superiority in efficiency of any algorithm emerges from the results. The MILP method always guarantees good performances with respect to the others, and its benefits can be further enhanced by the neighbourhood search, which, however, turned out to be too expensive to be carried out in this context, since fast decisions need to be taken at the beginning of a new relocation step. The best overall performances are provided by the LP+rounding method together with the neighbourhood search, but again, the neighbourhood search makes the resolution process too slow to be used in practice by the operator. Since a trade off between efficiency and computational times needs to be reached, the most convenient algorithm might be the MILP problem without the use of the neighbourhood search with only a 2 two-hour time frame look ahead horizon, which guarantees high efficiency in virtually no computational time.

efficiency	*look ahead horizon*					
	1	2	4	6	8	10
LP+MILP & no search	0.730	0.803	0.814	0.822	0.828	0.828
LP+MILP & search & best case	0.736	0.794	0.809	0.820	0.822	-
LP+MILP & search & Monte Carlo	0.766	0.816	0.814	0.832	-	-
LP+rounding & no search	0.718	0.806	0.815	0.820	0.820	0.828
LP+rounding & search & best case	0.747	0.801	0.809	0.828	0.821	0.825
LP+rounding & search & Monte Carlo	0.767	0.814	0.822	0.823	0.835	0.838
MILP & no search	0.755	0.818	0.818	0.832	0.834	0.836
MILP & search & best case	0.760	0.810	0.812	0.825	-	-
MILP & search & Monte Carlo	0.760	0.822	0.816	0.831	-	-

Table 8.3. Efficiency results of approximated algorithms in the real-world case.

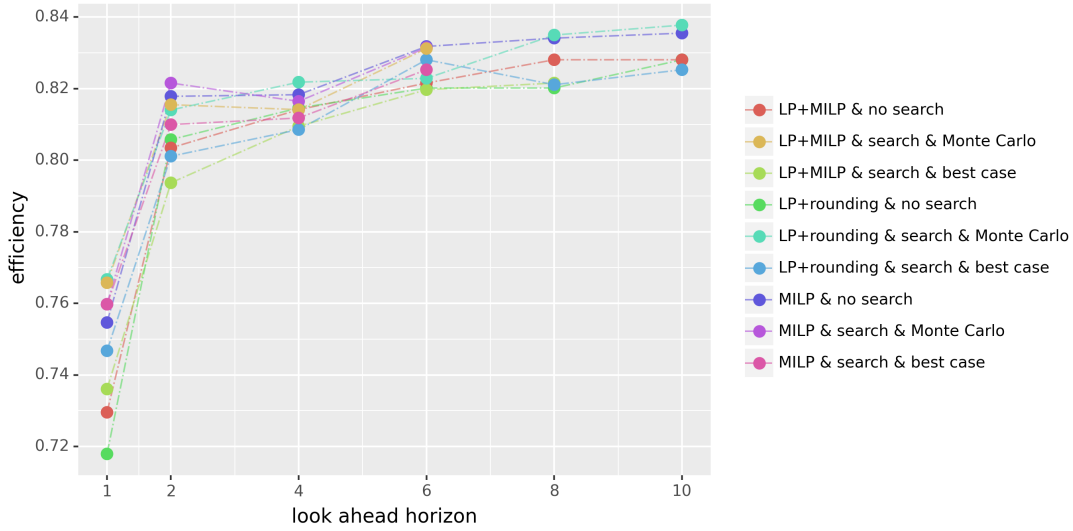

Figure 8.5. Efficiency results of approximated algorithms in the real-world case.

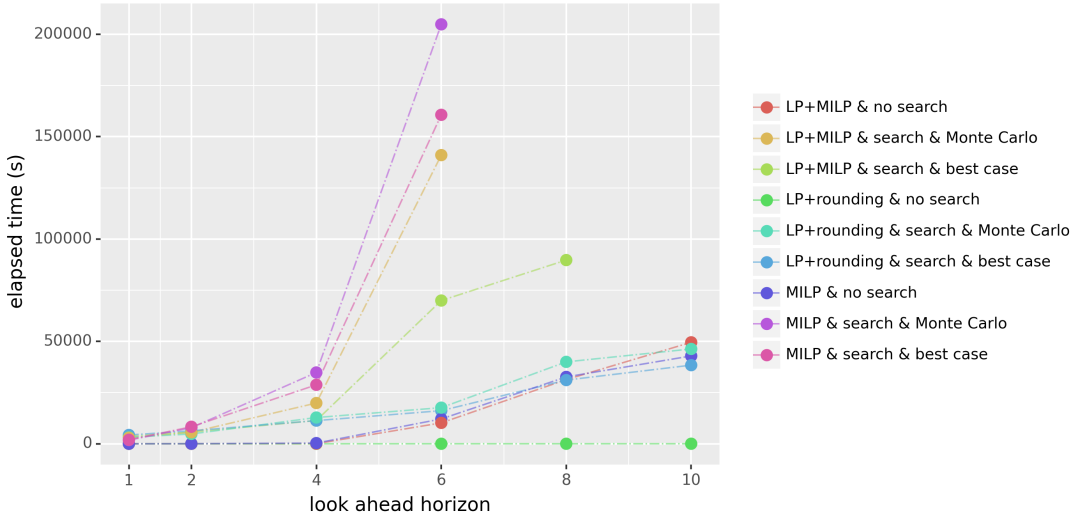

Figure 8.6. Elapsed time to run the approximated algorithms in the real-world case.

8.4 Analysis on the maximum relocation budget

As explained in section 3.1, the parameter R_{max} is the relocation budget set by the operator for each time frame, i.e., each relocation step. This means that, within a single time frame, a maximum number of R_{max} cars can be relocated by the support fleet. Of course, assuming that the operator is able to relocate all vehicles in the system is unreasonable, therefore such technical limit may be introduced. Besides, if all cars could

***elapsed time* (s)**	*look ahead horizon*					
	1	2	4	6	8	10
LP+MILP & no search	3.8	5.4	70.0	10207	31341	49553
LP+MILP & search & best case	3711	5734	11364	69952	89716	-
LP+MILP & search & Monte Carlo	2940	5602	19898	140977	-	-
LP+rounding & no search	1.3	2.8	6.5	10.8	16.8	24.2
LP+rounding & search & best case	4290	6330	11321	16173	31111	38399
LP+rounding& search& Monte Carlo	3163	4799	12823	17639	40000	46310
MILP & no search	5.3	40.6	340	12127	32653	42901
MILP & search & best case	1947	8311	28791	160644	-	-
MILP & search & Monte Carlo	1792	7845	34831	204831	-	-

Table 8.4. Elapsed time to run the approximated algorithms in the real-world case.

be moved within a single time frame to optimise the amount of satisfied trip requests, an optimisation procedure over multiple time frames would not even make sense, since it would be enough to find the system state guaranteeing the most satisfied trips during every time frame and attain that state at every step.

Using the optimisation method based on the resolution of the mixed integer linear program described in subsection 5.2.2, an analysis on the effect of the relocation budget on the efficiency was carried out. Figure 8.7 displays the system efficiency as a function of different values of R_{max}, ranging from 0 - the no relocation scenario - to 60, being around 15% of the total number of vehicles in the system. The plot shows that setting a higher relocation budget clearly boosts the efficiency and definitely proves that implementing relocation strategies is useful to attain a greater user satisfaction. Just moving 5 cars per time frame, making up only 1% of the fleet, increases the efficiency of the system by around 4.5%. Besides, further increasing R_{max} to 30 boosts the efficiency by around 12%. When $R_{max} > 30$ a decrease in the curve can be observed, probably due to the unavailability of too many cars, undergoing the relocation process and therefore not bookable by the users.

R_{max}	0	5	10	15	20
efficiency	0.747	0.789	0.818	0.831	0.848
R_{max}	25	30	35	40	45
efficiency	0.858	0.865	0.863	0.858	0.854
R_{max}	50	55	60		
efficiency	0.852	0.845	0.842		

Table 8.5. System efficiency by relocation budget.

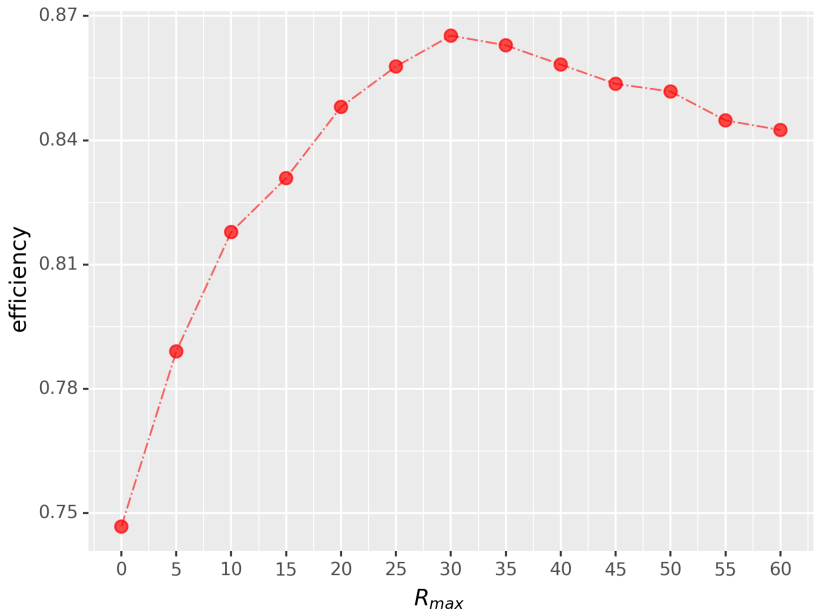

Figure 8.7. System efficiency by relocation budget.

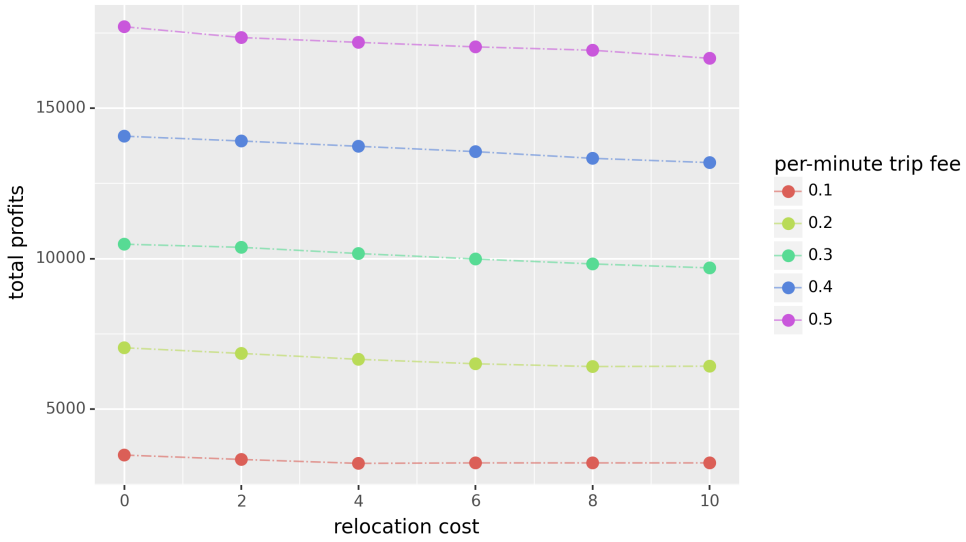

Figure 8.8. Total profits by trip minute fee and cost of each relocation.

8.5 Analysis on profits and relocation costs

This last section is devoted to carrying out a further analysis on the profits of the shared mobility system operator. The objective of the problem to be solved at any optimisation

round p is (3.1), which we repeat here:

$$\max \sum_{t \in \mathcal{T}_p} \sum_{i \in \mathcal{Z}} EO_{ti}.$$

Since only the number of satisfied trip requests is taken into account, all the relocation budget R_{max} tends to be used. Though, relocating a vehicle requires a support fleet and therefore comes with a cost, meaning that the efficiency gain - and, therefore, additional earnings - observed in the previous sections might be offset by the relocation cost.

This section analyses the effect of the relocation costs on the profits of the operator. To do so, the objective of the problem (3.1) was modified to

$$\max f \times \sum_{t \in \mathcal{T}_p} \sum_{i \in \mathcal{Z}} EO_{ti} - c \times \sum_{t \in \mathcal{T}_p} \sum_{i \in \mathcal{Z}} r_{ti}^{+},$$

where, like in (8.1), f is the average earnings from a trip and c is the average cost of relocating a vehicle. The first summation yields the number of satisfied trips over the whole look ahead horizon $\mathcal{T}_p$, while the second summation represents the total number of vehicles moved by the support staff during the relocation procedure. The previous equation therefore represents the total profits obtained over the look ahead period and it has to be maximised. Assuming the average trip duration to be 20 minutes and considering that mobility shared systems charge users based on a per-minute fee f_m, the average earnings from a trip can be written as

$$f = 20 \times f_m.$$

Using again the optimisation method based on the resolution of the mixed integer linear program described in subsection 5.2.2 and fixing again $R_{max} = 10$, an analysis on the effect of the relocation costs on the total profits was then carried out. Figure 8.8 displays the total profits as a function of the unit relocation cost r, which ranges from 0 to 10, and of the per-minute trip fee f_m, ranging from 0.1 to 0.5 (and, therefore, the average trip profit ranges from 2 to 10). The plot shows that, as one would expect, higher fees guarantee higher profits. The relocation costs impact on the profits, again, as expected, decreasing the latter approximately linearly. It is very important, however, to notice that a very basic analysis was carried out here, keeping the demand patterns independent on the rent fees, which is clearly unrealistic, since, usually demand models assume an inverse dependency of the demand and the prices.

Figure 8.9 shows the efficiency of the system as a function of the unit relocation cost r and of the per-minute trip fee f_m. The efficiency has a tendency to decrease with the increasing relocation costs, except from the 0.5 per-minute fee, the most expensive case taken into consideration, in which case no clear dependency emerges. Lower trip fee scenarios are the most sensitive to an increase in the relocation costs.

Finally, Figure 8.10 plots the total relocated vehicles during the whole optimisation period $\mathcal{P}$ as a function of the unit relocation cost r and of the per-minute trip fee f_m. Since both P and R_{max}, the optimisation horizon and the relocation budget per time frame, were set to 10, the maximum possible number of vehicles which can be transferred

from the operator is 100. The figure shows that, the higher the relocation costs, the less vehicles are relocated, since the additional cost of moving a vehicles does not get rewarded by a sufficient profit. Again, higher fee cases are less sensitive to an increase in relocation costs: with a 0.1 per-minute fee the relocation procedure becomes disadvantageous when the relocation cost is more than 4 per unit. On the contrary, a 0.5 per minute-fee makes the relocation process always useful, even with high relocation cost. Observing when the total relocation is zero explains why, in Figure 8.8, the total profit curves, after an initial linear dependency, flatten from a cost value on, depending on the rent fees.

total profits		*relocation cost*					
		0	2	4	6	8	10
per-minute trip fee	0.1	3474	3328	3198	3214	3214	3214
	0.2	7040	6852	6656	6508	6416	6428
	0.3	10482	10378	10172	9990	9826	9694
	0.4	14072	13912	13736	13558	13336	13194
	0.5	17710	17350	17190	17040	16928	16660

Table 8.6. Total profits by trip minute fee and cost of each relocation.

efficiency		*relocation cost*					
		0	2	4	6	8	10
per-minute trip fee	0.1	0.807	0.817	0.744	0.747	0.747	0.747
	0.2	0.818	0.819	0.816	0.780	0.746	0.747
	0.3	0.812	0.819	0.819	0.817	0.795	0.762
	0.4	0.817	0.820	0.821	0.822	0.817	0.808
	0.5	0.823	0.816	0.817	0.820	0.823	0.816

Table 8.7. System efficiency by trip minute fee and cost of each relocation.

total relocation		*relocation cost*					
		0	2	4	6	8	10
per-minute trip fee	0.1	100	94	1	0	0	0
	0.2	100	100	92	34	1	0
	0.3	100	100	100	93	55	14
	0.4	100	100	100	99	92	71
	0.5	100	100	100	100	99	91

Table 8.8. Total number of relocated vehicles by trip minute fee and cost of each relocation.

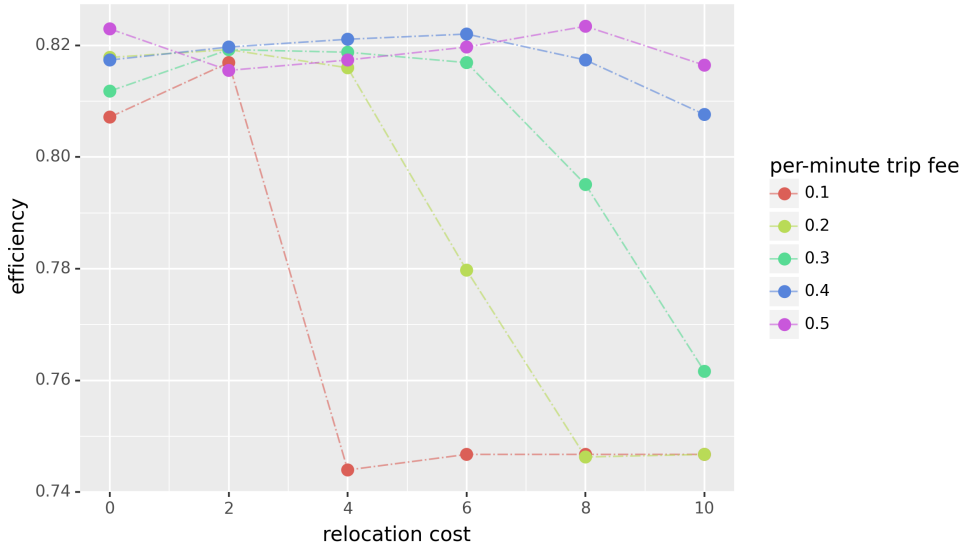

Figure 8.9. System efficiency by per-minute trip fee and cost of each relocation.

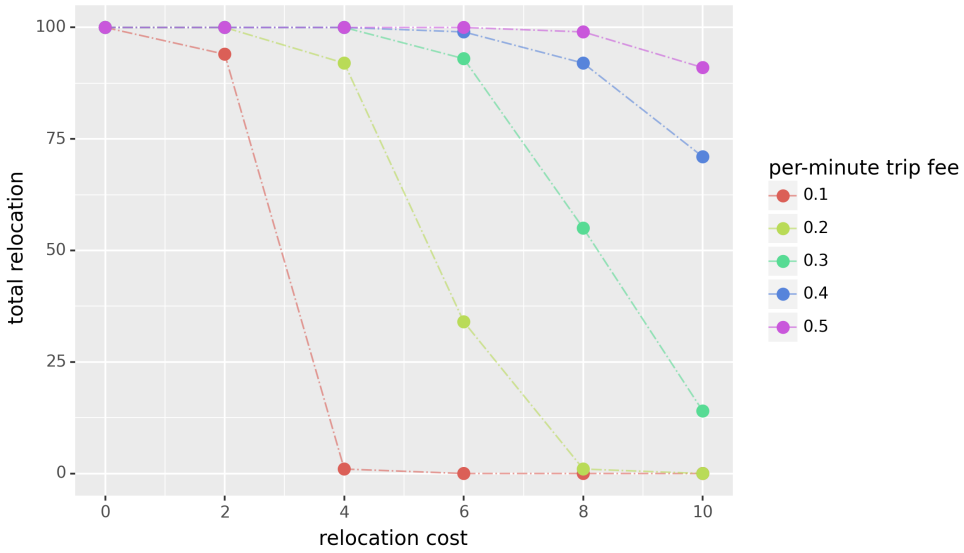

Figure 8.10. Total number of relocated vehicles by per-minute trip fee and cost of each relocation.

Chapter 9

Conclusions

Smart mobility system are widely spreading worldwide to address smart city challenges such as carbon reduction, traffic decongestion, multimodal urban transportation. However, the system's cyclical dynamic reconfiguration and the variability of the demand patterns throughout the operation period may reduce possible future demand, and, as a consequence, user satisfaction and profits. Relocation strategies are broadly implemented to guarantee a more even and appropriate vehicle distribution in the geographical operation area of the fleet, but, since these strategies come with high costs, it is of great interest to study methodologies to optimise such process.

A lot of work has already been done to develop optimal relocation strategies, employing all kinds of methodologies. Nonetheless, many algorithms focus on short term optimisation strategies, disregarding possible future evolutions of the system, which might be penalised from present decisions. This thesis went beyond this greedy approach, looking for the optimal relocation strategy gathering information on longer term demand patterns and using it to work out strategies that avoid myopic behaviours. The real-world case of the Car2go fleet operating in the city of Turin was used to validate the results and discuss the implementation of different strategies.

Although all previous research is already fairly positive on considering the implementation of relocation strategies to boost user satisfaction, this work once again confirmed that operators not relocating vehicles to meet the varying demand patterns suffer a big loss in user satisfaction. Numerical results showed that relocating only 1% of the fleet increases the efficiency of the system by around 4.5%, with a user satisfaction boost reaching 12% when further increasing to 7.5% the share of relocated vehicles. Too many transferred vehicles, however, seem to bring about a decrease in the user satisfaction, due to the unavailability of too many cars undergoing the relocation process and therefore not bookable by the users.

Being settled that implementing a relocation strategy does improve the system performances, the main aim of the present work was to investigate how long is it worth to look ahead in the future, considering that the further we look, the bigger the solution gets in dimensionality, requiring more and more time for the algorithms to find a solution to the problem. Results showed that a long-sighted strategy does indeed boost the user satisfaction, with an efficiency increase of around 7% when the look ahead period is set to

2 two-hour time frames and less than 3% of the total vehicles are moved by the operator. The increase in efficiency can reach up to 10% when the look ahead period gets to 10 two-hour time frames. However, when implementing strategies employing longer look aheads than 4 time frames, the computational times usually become overwhelming, making such algorithms unsuitable for the context of use, where fast decisions need to be taken based on the evolution of the system.

The algorithm providing the best trade off between performances and computational times turned out to be the mixed integer linear program optimising over a 2 two-hour look ahead horizon in virtually no time, guaranteeing a 82% system efficiency. However, all methods proved to be able to provide good solutions to the problem, although some of them needed definitely too much time to be run and fit the problem needs. In particular, implementing the neighbourhood search starting from the solution given by the optimisation program, was able to improve the solution, but the extra time needed to do so was too high for that to be a good alternative to other methods. Working with a toy case problem also proved that the developed algorithms, which are able to find only approximate solutions, can actually work out almost optimal strategies in many cases, taking a solution obtained through the dynamic programming approach as a benchmark. This latter solution is optimal, but the algorithm providing it gets soon computationally infeasible since the state space grows almost exponentially with the number of vehicles and city zones in the system.

In the previous considerations, only the user satisfaction was analysed. Nonetheless, the relocation process comes at a high cost, which might offset the additional earnings coming from trips carried out thanks to the operator intervention on the system. A basic analysis showed that, higher fees guarantee higher profits and the relocation costs impact on the profits decreasing the latter approximately linearly. Besides, the system efficiency has a tendency to decrease with the increasing relocation costs, except from very high per-minute fee scenarios. Finally, the higher the relocation costs, the less vehicles are relocated, since the additional cost of moving a vehicle does not get rewarded by a sufficient profit, with higher fee cases less sensitive to an increase in costs.

Based on the kind of system, the objective function could be modified in order to limit the number of zones involved in the relocation procedure, since, as an example, collecting bikes from many zones in the city would involve higher costs than collecting them from just a few. The present work aimed at devising a general and flexible procedure, therefore, this element was not taken into account. Moreover, the great computational effort required by some algorithms did not make it possible to assess their performances in a more accurate way. However, a Monte Carlo simulation of the system evolution, involving many runs, would be more suitable to inspect the algorithms' performances and assess the resulting efficiency more appropriately.

The total demand over all time frames was assumed to be perfectly known in this work, which is definitely unrealistic. Usually demand models are employed to predict the demand patterns, giving a probability distribution of the number of request during a given time frame. Future works could therefore introduce this further difficulty into the picture, investigating whether longer look ahead periods could actually turn out to be decreasing the methods' performances. The effect of the relocation costs could further be investigated, too, employing appropriate demand models based on the different fees

applied by the operator.

Finally, the present thesis developed methods to identify which city zones should act as donors and receivers of vehicles during every relocation step. The following stage would be to find the optimal path through the zones to redistribute the vehicles in the most convenient way. The vehicle routing problem could be a solution and it has been widely inspected in the existing literature, although some customisation of the constraints needs to be performed to account for, as an example, whether we are working with a car or bike sharing system, for which the practical redistribution procedure is very different.

Appendix A

Tuning simulated annealing

The present appendix describes the procedure employed to choose the cooling schedule for the simulated annealing strategy in the neighbourhood search algorithm.

The following possible schedules have been tested, trying different parameter configurations starting from the possibilities listed in subsection 6.1.2:

- Exponential schedules, whose general form is $T(t) = T_0\alpha^t$:
  - *exp_sched_0*: $T(t) = 1 \times 0.9^t$;
  - *exp_sched_1*: $T(t) = 10 \times 0.9^t$;
  - *exp_sched_2*: $T(t) = 1 \times 0.5^t$;
  - *exp_sched_3*: $T(t) = 10 \times 0.5^t$;
  - *exp_sched_4*: $T(t) = 0.1 \times 0.9^t$;
  - *exp_sched_5*: $T(t) = 0.1 \times 0.5^t$;
- Linear schedules, whose general form is $T(t) = T_0 - \eta t$:
  - *lin_sched_0*: $T(t) = 0.1 - 0.1 \times \frac{t-0.1}{(t_{max}+1)-0.1}$;
  - *lin_sched_1*: $T(t) = 1 - 1 \times \frac{t-1}{(t_{max}+1)-1}$;
  - *lin_sched_2*: $T(t) = 10 - 10 \times \frac{t-10}{(t_{max}+1)-10}$;
- Logarithmic schedules, whose general form is $T(t) = \frac{c}{log(t+d)}$ (usually $d = 1$):
  - *log_sched_0*: $T(t) = \frac{10}{log((t+1)+1)}$;
  - *log_sched_1*: $T(t) = \frac{1}{log((t+1)+1)}$;
  - *log_sched_2*: $T(t) = \frac{0.1}{log((t+1)+1)}$.

Observe that the parameter c in the logarithmic schedule is similar to the initial temperature parameter in the other schedules, and their values are set to either 0.1, 1 or 10. The parameter η in the linear schedules is set to give 0 temperature at the last iteration,

schedule	average percentage efficiency gain	standard deviation
exp_sched_0	3.94%	1.77%
exp_sched_1	3.03%	1.35%
exp_sched_2	3.89%	1.75%
exp_sched_3	3.50%	1.54%
exp_sched_4	3.38%	1.49%
exp_sched_5	3.26%	1.43%
lin_sched_0	3.52%	1.51%
lin_sched_1	3.60%	1.69%
lin_sched_2	-2.76%	2.31%
log_sched_0	-1.70%	2.14%
log_sched_1	-3.09%	3.16%
log_sched_2	3.34%	1.45%

Table A.1. Average percentage efficiency gain and standard deviation by cooling schedule function choice.

in order to decrease the number of tunable parameters. To do so, the maximum number of iterations t_{max} needs to be provided. Finally, $t+1$ instead of t is used in the linear and logarithmic schedules to avoid them to take up exactly zero value, since that would give issues when using the temperature function in the simulated annealing approach (which requires to compute the acceptance probability dividing by the temperature).

The tuning procedure was carried out with mobility data coming from the same dataset described in chapter 7, but different data from those used to test the algorithms were extracted. The maximum number of iterations was set to 50, and the maximum number of iterations which produced no change in the incumbent solution to 15.

The performance metric employed to compare the different cooling schedules is the percentage efficiency gain of the system. It is computed starting from the efficiency of the starting solution e_0, given by solving the linear program described in chapter 5, and the efficiency obtained by the solution after a round of neighbourhood search implementing the simulated annealing approach has been carried out, in the following way:

$$\text{percentage efficiency gain} = \frac{e_1 - e_0}{e_0} \times 100$$

Finally, to gather more data and test the simulated annealing and the neighbourhood search in more scenarios, the procedure was carried out optimising with different look ahead periods in the range $[1,\ 4]$, causing the solution to gradually increase its dimensionality.

Table A.1 shows the results of the trials. The average percentage efficiency gain of the runs involving the same schedule is displayed, together with its standard deviation. The same results are presented in Figure A.1, which exhibits the dispersion of the data

through box plots produces from each sample, grouped by the chosen schedule. Based on these results, the first exponential schedule was used to carry out all the neighbourhood search steps. In general, the exponential schedule is the best performing one, while the logarithmic schedule does not seem to fit the problem needs.

Some further considerations can be made. Figure A.2 plots the average percentage efficiency gain by the initial temperature value, grouped by the schedule type. The plot shows that small values of the initial temperature - namely, 0.1 - always give good results, probably because the algorithm starts from an already performing solution which does not need big adjustments. Higher initial temperature values give mixed results, with the logarithmic schedule always worsening the quality of the solution. The reason behind this behaviour is, probably, that high values of initial temperature make accepting sub-optimal solutions too much likely.

Finally, Figure A.3 plots the average percentage efficiency gain by the look ahead horizon of the solution. The results do not show any striking connection between these two variables, even though a longer look ahead increases the solution dimensionality and, therefore, it should be harder for the algorithm to improve the quality of the solution.

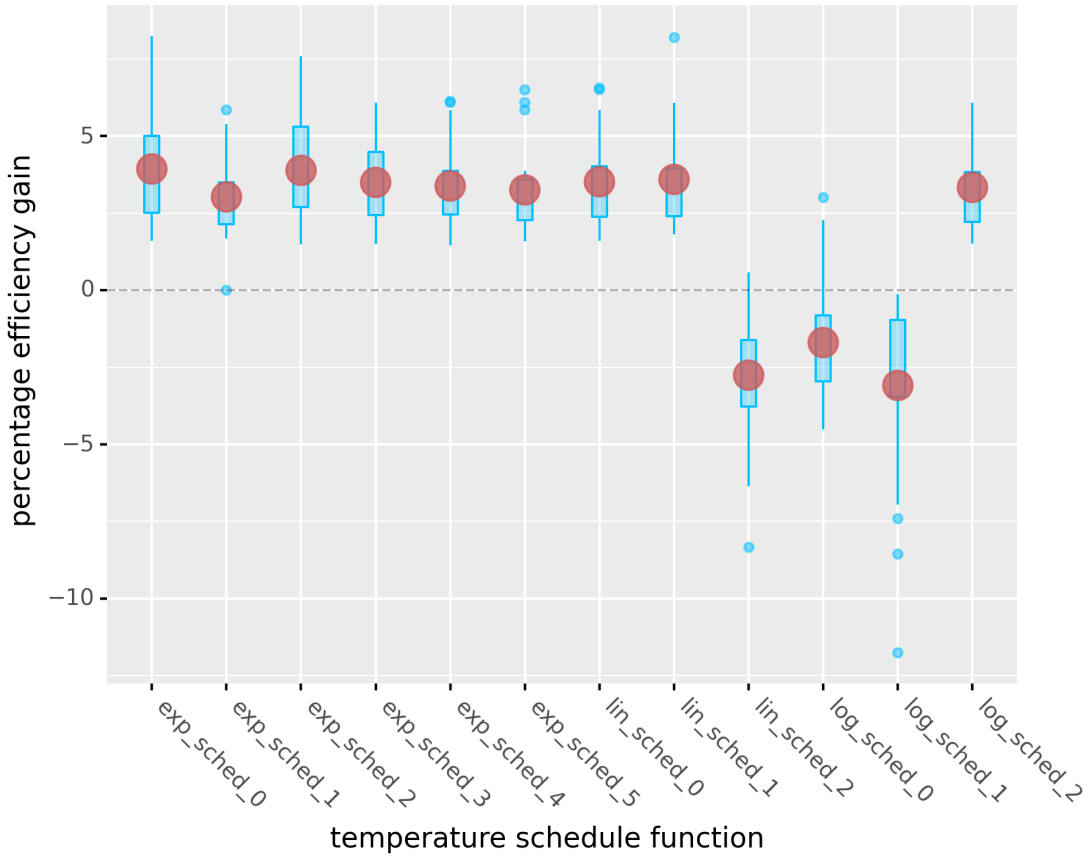

Figure A.1. Average percentage efficiency gain by cooling schedule function choice.

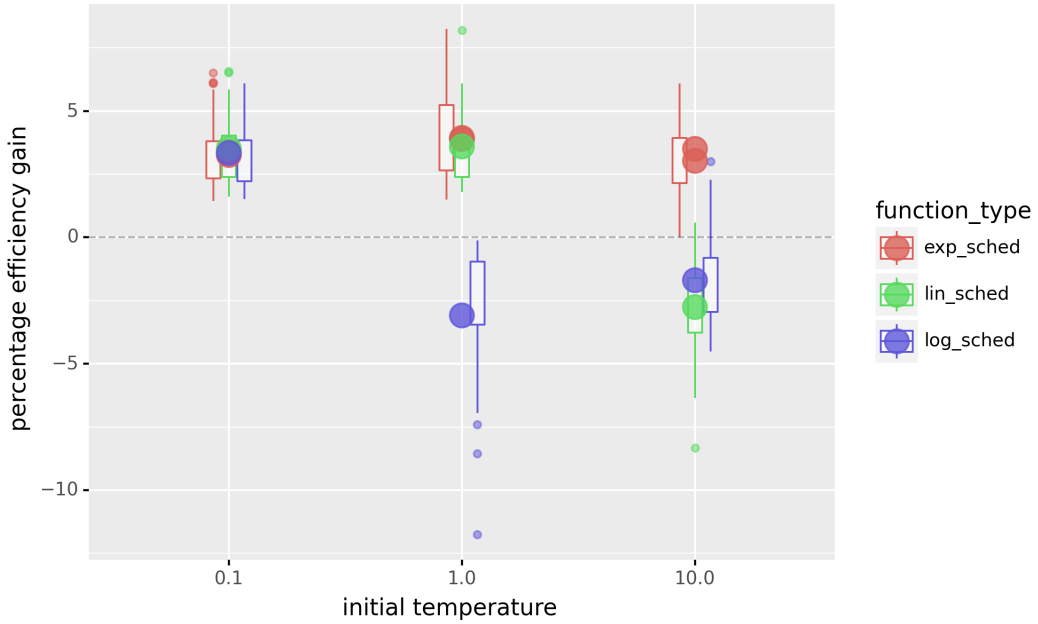

Figure A.2. Average percentage efficiency gain by initial temperature choice.

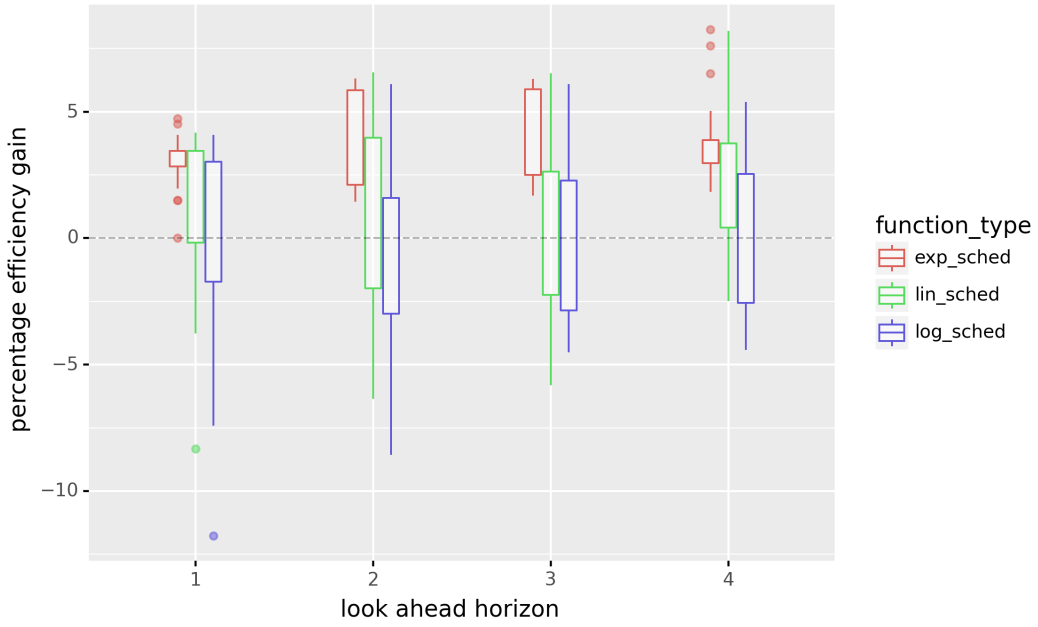

Figure A.3. Average percentage efficiency gain by look ahead horizon.

Bibliography

Vitória Albuquerque, Miguel Sales Dias, and Fernando Bacao. Machine learning approaches to bike-sharing systems: A systematic literature review. *ISPRS International Journal of Geo-Information*, 10(2), 2021. ISSN 2220-9964. doi: 10.3390/ijgi10020062. URL https://www.mdpi.com/2220-9964/10/2/62.

Seonghoon Ban and Kyung Hoon Hyun. Curvature-based distribution algorithm: rebalancing bike sharing system with agent-based simulation. *Journal of Visualization*, 22, 04 2019. doi: 10.1007/s12650-019-00557-6.

Michelangelo Barulli, Alessandro Ciociola, Michele Cocca, Luca Vassio, Danilo Giordano, and Marco Mellia. On scalability of electric car sharing in smart cities. In *2020 IEEE International Smart Cities Conference (ISC2)*, pages 1–8. IEEE, 2020.

Dimitris Bertsimas and John Tsitsiklis. Simulated Annealing. *Statistical Science*, 8(1): 10 – 15, 1993. doi: 10.1214/ss/1177011077. URL https://doi.org/10.1214/ss/1177011077.

Karl-Heinz Borgwardt. The simplex algorithm: A probabilistic analysis. *Algorithms and Combinatorics*, 1, 1987.

Paolo Brandimarte. *From Shortest Paths to Reinforcement Learning*. Springer Cham, 2021. ISBN 978-3-030-61866-7. doi: https://doi.org/10.1007/978-3-030-61867-4. URL https://link.springer.com/book/10.1007/978-3-030-61867-4.

Jan Brinkmann, Marlin W. Ulmer, and Dirk C. Mattfeld. Dynamic lookahead policies for stochastic-dynamic inventory routing in bike sharing systems. *Computers & Operations Research*, 106:260–279, 2019. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2018.06.004. URL https://www.sciencedirect.com/science/article/pii/S0305054818301588.

Olli Bräysy and Michel Gendreau. Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation Science*, 39:104–118, 02 2005a. doi: 10.1287/trsc.1030.0056.

Olli Bräysy and Michel Gendreau. Vehicle routing problem with time windows, part ii: Metaheuristics. *Transportation Science*, 39:119–139, 02 2005b. doi: 10.1287/trsc.1030.0057.

Leonardo Caggiani, Rosalia Camporeale, Michele Ottomanelli, and Wai Yuen Szeto. A modeling framework for the dynamic management of free-floating bike-sharing systems. *Transportation Research Part C: Emerging Technologies*, 87:159–182, 2018. ISSN 0968-090X. doi: https://doi.org/10.1016/j.trc.2018.01.001. URL https://www.sciencedirect.com/science/article/pii/S0968090X18300020.

Federico Chiariotti, Chiara Pielli, Andrea Zanella, and Michele Zorzi. A dynamic approach to rebalancing bike-sharing systems. *Sensors*, 18(2), 2018. ISSN 1424-8220. doi: 10.3390/s18020512. URL https://www.mdpi.com/1424-8220/18/2/512.

Alessandro Ciociola, Michele Cocca, Danilo Giordano, Luca Vassio, and Marco Mellia. E-scooter sharing: Leveraging open data for system design. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–8, 2020a. doi: 10.1109/DS-RT50469.2020.9213514.

Alessandro Ciociola, Dena Markudova, Luca Vassio, Danilo Giordano, Marco Mellia, and Michela Meo. Impact of charging infrastructure and policies on electric car sharing systems. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2020b.

Michele Cocca, Danilo Giordano, Marco Mellia, and Luca Vassio. Data driven optimization of charging station placement for ev free floating car sharing. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2490–2495, 2018. doi: 10.1109/ITSC.2018.8569256.

Michele Cocca, Danilo Giordano, Marco Mellia, and Luca Vassio. Free floating electric car sharing design: Data driven optimisation. *Pervasive Mob. Comput.*, 55(C):59–75, apr 2019. ISSN 1574-1192. doi: 10.1016/j.pmcj.2019.02.007. URL https://doi.org/10.1016/j.pmcj.2019.02.007.

Michele Cocca, Douglas Teixeira, Luca Vassio, Marco Mellia, Jussara M. Almeida, and Ana Paula Couto da Silva. On car-sharing usage prediction with open socio-demographic data. *Electronics*, 9(1), 2020. ISSN 2079-9292. doi: 10.3390/electronics9010072. URL https://www.mdpi.com/2079-9292/9/1/72.

Maria Pia Fanti, Agostino Marcello Mangini, Michele Roccotelli, Bartolomeo Silvestri, and Salvatore Digiesi. Electric vehicle fleet relocation management for sharing systems based on incentive mechanism. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 1048–1053, 2019. doi: 10.1109/COASE.2019.8842852.

Edoardo Fassio, Alessandro Ciociola, Danilo Giordano, Michel Noussan, Luca Vassio, and Marco Mellia. Environmental and economic comparison of icev and ev in car sharing. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 1621–1626, 2021. doi: 10.1109/ITSC48978.2021.9564578.

Matteo Fischetti and Andrea Lodi. *Heuristics in Mixed Integer Programming*. John Wiley & Sons, Ltd, 2011. ISBN 9780470400531. doi: https://doi.org/10.1002/

9780470400531.eorms0376. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0376.

Fahimeh Golbabaei, Tan Yigitcanlar, and Jonathan Bunker. The role of shared autonomous vehicle systems in delivering smart urban mobility: A systematic review of the literature. *International Journal of Sustainable Transportation*, 15(10):731–748, 2021. doi: 10.1080/15568318.2020.1798571. URL https://doi.org/10.1080/15568318.2020.1798571.

Simen Hellem, Carl Andreas Julsvoll, Magnus Moan, Henrik Andersson, Kjetil Fagerholt, and Giovanni Pantuso. The dynamic electric carsharing relocation problem. *EURO Journal on Transportation and Logistics*, 10:100055, 2021. ISSN 2192-4376. doi: https://doi.org/10.1016/j.ejtl.2021.100055. URL https://www.sciencedirect.com/science/article/pii/S2192437621000248.

Junjie Hu, Hugo Morais, Tiago Sousa, and Morten Lind. Electric vehicle fleet management in smart grids: A review of services, optimization and control aspects. *Renewable and Sustainable Energy Reviews*, 56:1207–1226, 2016. ISSN 1364-0321. doi: https://doi.org/10.1016/j.rser.2015.12.014. URL https://www.sciencedirect.com/science/article/pii/S1364032115013970.

J. Lin and T. Chou. A geo-aware and vrp-based public bicycle redistribution system. *International Journal of Vehicular Technology*, 2012, 12 2012. doi: 10.1155/2012/963427.

Yaghout Nourani and Bjarne Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373–8385, oct 1998. doi: 10.1088/0305-4470/31/41/011. URL https://doi.org/10.1088/0305-4470/31/41/011.

Ling Pan, Qingpeng Cai, Zhixuan Fang, Pingzhong Tang, and Longbo Huang. Rebalancing dockless bike sharing systems. *CoRR*, abs/1802.04592, 2018. URL http://arxiv.org/abs/1802.04592.

Julius Pfrommer, Joseph Warrington, Georg Schildbach, and Manfred Morari. Dynamic vehicle redistribution and online price incentives in shared mobility systems. *IEEE Transactions on Intelligent Transportation Systems*, 15(4):1567–1578, 2014. doi: 10.1109/TITS.2014.2303986.

Peter Salamon, James D. Nulton, John R. Harland, Jacob Pedersen, George Ruppeiner, and Luby Liao. Simulated annealing with constant thermodynamic speed. *Computer Physics Communications*, 49(3):423–428, 1988. ISSN 0010-4655. doi: https://doi.org/10.1016/0010-4655(88)90003-3. URL https://www.sciencedirect.com/science/article/pii/0010465588900033.

Lei Shi, Yong Zhang, Weina Rui, and Xinzheng Yang. Study on the bike-sharing inventory rebalancing and vehicle routing for bike-sharing system. *Transportation Research Procedia*, 39:624–633, 2019. ISSN 2352-1465. doi: https://doi.org/10.1016/j.trpro.2019.06.064. URL https://www.sciencedirect.com/science/article/pii/

S2352146519301528. 3rd International Conference "Green Cities – Green Logistics for Greener Cities", Szczecin, 13-14 September 2018.

Qiong Tang, Zhuo Fu, Dezhi Zhang, Hao Guo, and Minyi Li. Addressing the bike repositioning problem in bike sharing system: A two-stage stochastic programming model. *Scientific Programming*, 2020:1–12, 05 2020a. doi: 10.1155/2020/8868892.

Xindi Tang, Congyuan Ji, Fang He, and Meng Li. Online operation of bike sharing system: An approximate dynamic programming approach. pages 2647–2658, 08 2020b. doi: 10.1061/9780784482933.228.

Leonardo Tolomei, Stefano Fiorini, Alessandro Ciociola, Luca Vassio, Danilo Giordano, and Marco Mellia. Benefits of relocation on e-scooter sharing-a data-informed approach. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 3170–3175. IEEE, 2021.

Carlos M. Vallez, Mario Castro, and David Contreras. Challenges and opportunities in dock-based bike-sharing rebalancing: A systematic review. *Sustainability*, 13(4), 2021. ISSN 2071-1050. doi: 10.3390/su13041829. URL https://www.mdpi.com/2071-1050/13/4/1829.

Ning Wang, Qiaoqian Liu, Jiahui Guo, and Tong Fang. A user-based adaptive joint relocation model combining electric car-sharing and bicycle-sharing. *Transportmetrica B: Transport Dynamics*, 10(1):1046–1069, 2022. doi: 10.1080/21680566.2021.2007174. URL https://doi.org/10.1080/21680566.2021.2007174.

Konstanze Winter, Oded Cats, Gonçalo Homem de Almeida Correia, and Bart van Arem. Designing an automated demand-responsive transport system: Fleet size and performance analysis for a campus–train station service. *Transportation Research Record*, 2542 (1):75–83, 2016. doi: 10.3141/2542-09. URL https://doi.org/10.3141/2542-09.

Ruijing Wu, Shaoxuan Liu, and Zhenyang Shi. Customer incentive rebalancing plan in free-float bike-sharing system with limited information. *Sustainability*, 11(11), 2019. ISSN 2071-1050. doi: 10.3390/su11113088. URL https://www.mdpi.com/2071-1050/11/11/3088.