BRENO ALVES BEIRIGO

SINGLE-OBJECTIVE AND BI-OBJECTIVE PARALLEL HEURISTICS FOR THE TRAVEL PLANNING PROBLEM

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*

VIÇOSA
MINAS GERAIS - BRASIL
2016

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
B422s
2016

Beirigo, Breno Alves, 1991-
Single-objective and bi-objetive parallel heuristics for the travel planning problem / Breno Alves Beirigo. – Viçosa, MG, 2016.
xii, 92f. : il. (algumas color.) ; 29 cm.

Inclui apêndice.
Orientador: André Gustavo dos Santos.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 83-86.

1. Pesquisa operacional. 2. Otimização combinatória. 3. Solução de problemas. 4. Heurística. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22 ed. 001.424

BRENO ALVES BEIRIGO

SINGLE-OBJECTIVE AND BI-OBJECTIVE PARALLEL HEURISTICS FOR THE TRAVEL PLANNING PROBLEM

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADO: 21 de setembro de 2016.

Mauro Nacif Rocha

Luciana Brugiolo Gonçalves

André Gustavo dos Santos
(Orientador)

We are what we repeatedly do; excellence, then, is not an act but a habit.
Aristotle

Acknowledgments

Agradeço a minha mãe pelo apoio incondicional, ao meu orientador André Gustavo dos Santos por me acompanhar durante mais uma etapa da minha formação, e a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo financiamento da minha bolsa durante o curso.

Contents

List of Tables

List of Figures

Resumo

BEIRIGO, Breno Alves, M.Sc., Universidade Federal de Viçosa, setembro de 2016. **Heurísticas Paralelas para o Problema de Planejamento de Viagens Mono-objetivo e Bi-objetivo**. Orientador: André Gustavo dos Santos.

Nesse trabalho são aplicadas heurísticas paralelas mono-objetivas e bi-objetivas para solucionar formulações abrangentes e realistas do problema de planejamento de viagens. Dado o intervalo de tempo que uma viagem pode ocorrer e um conjunto de destinos com seus respectivos tempos de permanência, a abordagem mono-objetiva procura determinar um itinerário de baixo custo que compreenda voos, hotéis e horários de partida/chegada. Por sua vez, a abordagem bi-objetiva adiciona complexidade à formulação do problema, uma vez que pretende determinar o conjunto Pareto de itinerários de viagem capazes de equilibrar custo e tempo. Quando a sequência de cidades é fixa, a versão mono-objetiva do problema é comumente modelada na literatura como uma rede dependente do tempo e o melhor itinerário é calculado usando algoritmos de caminho mínimo. Contudo, nesse trabalho, determinar a ordem de visitação das cidades também é um objetivo. Portanto, a formulação mono-objetiva proposta representa um TDSPP *(Time Dependent Shortest Path Problem)* incorporado ao TSP *(Travel Salesman Problem)* e a formulação bi-objetiva representa um TDSPP incorporado em um TSP bi-objetivo. Na primeira formulação foi aplicada a heurística ILS *(Iterated Local Search)* e na segunda formulação o framework NSGA-II *(Nondominated Sorting Genetic Algorithm II)*. Os resultados de ambas as heurísticas foram comparados com os resultados produzidos por métodos exatos executados sem restrições temporais. Todos os casos de teste simulam itinerários de viagem realistas e foram executados em um banco de dados de viagens e hospedagens coletadas com antecedência. Além disso, independentemente da abordagem utilizada, estabeleceu-se que o tempo de execução de cada caso deve ser de aproximadamente 1 minuto. A heurística ILS proposta para a versão mono-objetiva do problema foi executada em 285 instâncias e alcançou, em média, soluções no máximo 4.1% divergentes de uma implementação exata, além de atingir a melhor solução em cerca de 30% dos casos de teste. Por sua vez, o framework NSGA-II foi capaz de produzir soluções no máximo 8% divergentes da implementação exata para 180 instâncias.

Abstract

BEIRIGO, Breno Alves, M.Sc., Universidade Federal de Viçosa, September, 2016. **Single-objective and Bi-objective Parallel Heuristics for the Travel Planning Problem**. Advisor: André Gustavo dos Santos.

In this study we apply single-objective and bi-objective parallel heuristics to solve broad and realistic formulations of the travel planning problem. Given a travel time window and a set of destinations with their corresponding dwelling times, the goal of our single-objective approach is to find a route that produces a budget travel's itinerary, involving flights, hotels and departure/arrival times. In turn, our bi-objective approach adds a complexity level in the problem's formulation once we are seeking for a Pareto set of detailed travel itineraries, which are both cost and time efficient. When the sequence of cities is fixed, the single-objective version of the problem is commonly modeled in literature as a time-dependent network and the best itinerary is computed using shortest path algorithms. However, in this study, finding the order of cities that minimizes the total cost, and besides that, a set of good trade-off solutions, are also goals. Therefore, our single-objective formulation stands for a TDSPP (Time Dependent Shortest Path Problem) embedded in the TSP (Travel Salesman Problem) whereas our bi-objective formulation stands for a TDSPP embedded in a bi-objective TSP. On the first formulation we apply an ILS (Iterated Local Search) heuristic and on the second formulation we apply the NSGA-II (Nondominated Sorting Genetic Algorithm II) framework. For performance assessing, the results of both heuristics were compared to the results of corresponding exact methods with no time constraints. All test cases simulate realistic travel itineraries and run upon real-world travel data collected in advance, besides having to comply with an execution threshold of approximately 1 minute. For 285 single-objective test cases, our ILS heuristic was able to reach solutions in average less than 4.1% divergent from an exact implementation, besides reaching the optimal solution in about 30% of the test cases. In turn, for 180 bi-objective test cases, our NSGA-II implementation was able to reach an approximated solution in average up to 8% divergent from an exact implementation.

1 Introduction

The focus of this study is to present optimization strategies that can support the decision making process of a tourist planning a travel. Firstly, in Section 1.1 we present the problem that motivated us to start this study. Then, in Section 1.2 we present the research gap our study intends to address. Next, in Section 1.3 we state all the goals we intend to achieve in this study as well as some specific objectives needed to fulfill these goals. Finally, the contributions and the outline of this study are presented in Section 1.4 and in Section 1.5, respectively.

1.1 Hardships of travel planning

Planning a vacation travel is a process both rewarding and laborious. On the one hand, it increases the tourist satisfaction by reducing uncertainties, budgetary ones inclusive. On the other, it requires patience and dedication once it is necessary to evaluate many travel data to determine a convenient itinerary. Indeed, for most tourists, a trip that lasts more than a week is an important decision that requires previous planning. According to Zalatan (1996) the bigger the distances involved, the bigger the planning horizon: tourists will devote considerable time and energy to the travel decision in order to identify better deals and obtain the best return on their expenditure. In fact, travel planning, as a specific type of consumer information search, can be considered a fundamental component of trip experience in that a traveler often needs to obtain a substantial amount of information in order to develop a travel plan (XIANG et al., 2014). According to Hyde et al. (2011) the Internet is also driving changes to vacation decision-making processes, by providing convenient and ready access for researching and purchasing vacation products. As pointed out by Xiang et al. (2014), the advent of the web empowered travelers adding an additional layer of intermediation to the travel market. This layer is comprised of fare aggregators, meta search engines and social media. While the last provides consumers a platform to not only interact with businesses but also to exchange opinions with other individuals, fare aggregators and meta search engines enable users to compare prices conveniently: they search online travel agencies and supplier sites in order to find the best deals (KRACHT; WANG, 2010).

Although web technology has increasingly given the tourists the capability to comparison shop, this comes at the cost of time and effort in wading through the complex structure of alternative distribution choices (KRACHT; WANG, 2010). As Choudhury et al. (2010) points out, finding a good plan requires a skilled interaction with a multitude of on-line resources: a budget traveler needs to examine several websites in order to compare prices, dates, times and availability. In fact, different online travel agents may have different supplier relationships, and different meta-

search engines might use different search algorithms therefore providing distinct results(KRACHT; WANG, 2010). Moreover, the multi-city option of meta-search engines and travel agencies currently operating requires a basic input schedule of the travel plan: the customer has to provide the sequence of destinations with corresponding departures dates. However, when the tourist intends to travel to many destinations, as longs as he/she spends a certain time at each location, the sequence of visits may become irrelevant. In some cases, even the departure date of the journey does not need to be previously defined: the journey may start at any day within a time window, for instance, a vacation period. In one hand, these features add flexibility to the travel planning, since the traveler must provide only the destinations, the duration of each visit and a travel window. On the other hand, without a support application to automate the process, finding the best deal is an extremely challenging and time consuming activity.

For 2 destinations, for example, there are only 2 possible visiting routes, but verifying the feasibility of each one of them, considering their position on the travel time horizon and the time-dependent travel data, would be an impractical task. If the tourist intends to stay 4 days in each one of these 2 destinations and the travel window span is 30 days, there are 22 possible journey's starting dates, from day 1 to day 22. Hence, the lowest price could only be achieved after checking 44 travel options in order to identify an adequate combination of flights and stays able to decrease the total cost. Nevertheless, even if the tourist were patient enough to check all possible itineraries, at the time he finishes the search, the solution may have become invalid. Travel data are very sensible to demand, so that prices and availability may vary every other minute: the last place in a flight or hotel may be suddenly taken by another customer, invalidating an under construction solution. Figure 1 shows how flight prices may vary within a one month travel window when the order of cities and the departure date from the origin city are flexible. All options correspond to the best deals found on a travel agency website for 3 departure dates and 2 different sequences of visits. Given this scenario, the tourist would be able to save up to $70 dollars if he chose options 2 or 4 over option 5. Besides that, the total travel times of solutions 2, 4 and 5 are 27h10, 27h10 and 28h15 respectively, that is, solution 5 is worse than solutions 2 and 4 regarding two metrics. As the number of destinations gets bigger, the complexity of the problem increases since many more options have to be checked in order to determine the best itinerary.

1.2 Problem statement

The scenario described in Section 1.1 highlights the necessities of a traveler who wants to determine the best travel itinerary considering flexible departure/arrival dates and a set of cities to visit. Besides determining the order of visits, this itinerary

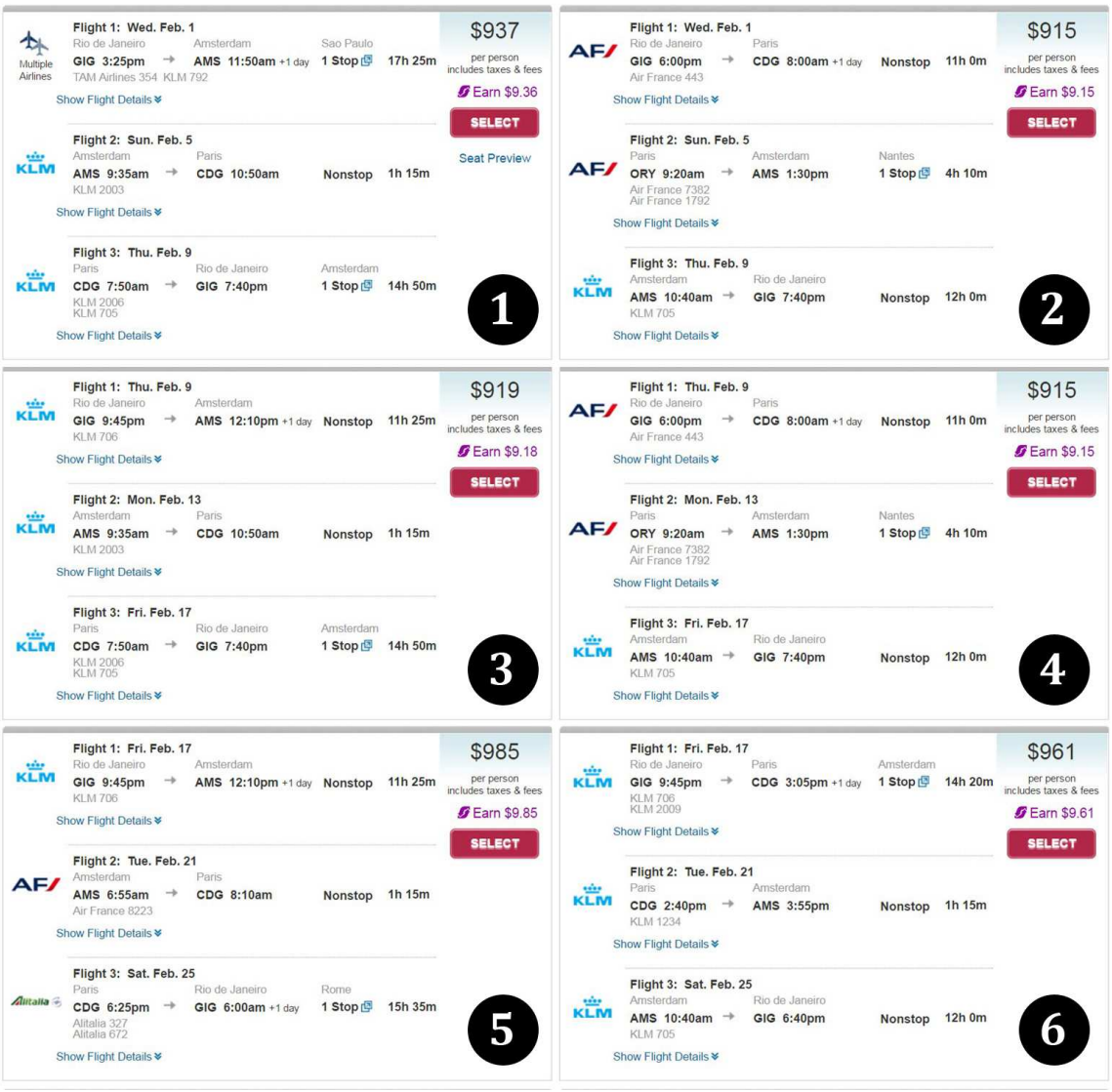

Figure 1 – Differences in flight prices according to the order of visits and departure dates. The first column shows the best deals for the route {Rio de Janeiro, Amsterdam, Paris, Rio de Janeiro}. In turn, the second column show the best deals for route {Rio de Janeiro, Paris, Amsterdam, Rio de Janeiro}. For each route we tested 3 different starting dates (2017/02/01, 2017/02/09, 2017/02/17) and the dwelling time in each destination is 4 days from the date of arrival. Data was collected on 2016/04/17 from Orbitz online travel agency.

must contain the dates and times of departure/arrival in each point of the trip, and also comply with the minimum dwelling times specified by the tourist. Moreover, adequate accommodations must be selected throughout the complete journey. Finding this optimal itinerary in a real world travel environment is what we consider in this study as the travel planning problem. Once this itinerary is achieved, a customer would be able to buy all flight's tickets and book all hotels in advance.

The problem of finding the best sequence of visits stands for the classic NP-Hard Traveling Salesman Problem (TSP), a well-studied and challenging combinatorial problem with many real world applications. In spite of many TSP applications have already been investigated (see literature review in Chapter 2), few authors approach the problem considering the restrictions imposed by a real world travel planning application that must consider transport options, dwelling times and flexible dates. Some authors do consider these travel industry constraints but focus on determining an optimal itinerary for a given route, not the route that produces an optimal itinerary. This is the case of Bérubé, Potvin and Vaucher (2006) who search for an optimal travel plan in a deterministic environment given a predetermined sequence of visits. This plan is created from a Time Dependent Shortest Path Problem (TDSPP) framework, in which shortest path algorithms are used to traverse a space-time network, especially built to deal with the flight/hotel data. To the best of our knowledge only Li, Zhou and Zhao (2016) address the described problem through a combinatorial optimization lens. The authors propose a travel itinerary problem (TIP) which aims to find itineraries with the lowest cost for travelers visiting multiple cities, under the constraints of time horizon, stop times at cities and transport alternatives with fixed departure times, arrival times, and ticket prices. Similarly to the study conduct in this thesis, the scale of TIP is generally small: travelers are usually not inclined to visit dozens of cities at one time. Their solution was able to reach the global optimal itinerary throughout an implicit enumeration algorithm for up to 5 destinations in less than 40s. However, the authors do no test the limitations of the algorithm: the TSP problem has a factorial growth rate what means that finding the global optimal can be extremely time consuming if a few more cities were considered in their analysis.

Moreover, both methods try to optimize only a single objective at a time. In fact, a budget traveler is often interested solely in the total cost of the journey, so that, large waiting times, unusual schedules or uncomfortable transports/accommodations are secondary concerns. However, a more realistic approach to the problem would account for other relevant features when choosing for a travel itinerary. A tourist, for example, could request for a solution that also tries to minimize travels' durations or general waiting times besides minimizing cost. These objectives are often conflicting: the prices tend to increase as travels' durations decreases whereas longer travels, possibly with stops, tend to be cheaper. As pointed out in (ZHOU et al., 2011), in

a multiobjective optimization problem (MOP), a single solution is hardly capable of optimizing all objectives simultaneously. Instead, the best trade-off solutions have to be provided to a decision maker (DM) so that he can choose a suitable option. Hence, the flow of execution of a simple multiobjective travel planning system would be as follows. Firstly, the tourist inputs travel preferences into the system: the time traveling window, the destinations cities and their correspondent dwelling times. Then, the optimization method would return the best set of trade-off solutions given the input settings and the objectives chosen. After the search process, the decision maker can finally choose the best suited solution. Of course, if a single objective approach is concerned, this system would produce a single itinerary, regarding for example, the global cost of the trip. However, regardless the number of objectives, the majority of the studies do not consider realistic data nor have a pressing time execution restriction. In fact, these characteristics impose challenging limitations once a strict time frame may harm the proper convergence of the optimization process.

In summary, regarding the limitations of the online solutions and the gap identified in literature, this study intends to investigate strategies to create practical and realistic travel itineraries for two versions of the travel planning problem: single-objective (cost efficient) and bi-objective (cost and time efficient).

1.3 Objectives

Given a set of destinations, their minimum dwelling times, a travel time horizon and realistic travel data covering transportation and hotels, our goal is to determine:

- The best, or near best, travel itinerary regarding trip's global cost (in less than 1 minute).
- The best, or near best, Pareto set of travel itineraries regarding the trip's global cost and waiting time (in less than 1 minute).

In order to fulfill these goals, the following specific objectives have to be addressed:

- Collect and store real-world travel data regarding flights and accommodations.
- Design a travel network to handle the data collected.
- Design heuristics to rapidly solve the single-objective and bi-objective versions of the travel planning problem.
- Develop exact approaches for assessing the performance of both heuristic methods.

- Calibrate the appropriate parameters of each heuristic approach so that their deviations from the corresponding exact approaches are minimal.
- Implement parallel versions for highly used procedures (when applicable) in order to decrease execution time.

1.4 Contributions of the thesis

The main contributions of this thesis are the following:

- We propose a real world approach to the travel planning problem. In contrast with popular online planners and literature approaches, this study takes in consideration all the details involved in a flexible multi city travel planning to quickly create quality solutions.
- We collect real world flight data concerning 16 destinations and a 41 days travel window. This data can be adapted in other studies to further investigation on finding relevant solutions for transportation related problems.
- We propose a single-objective integer programming formulation for our version of the travel planning problem.
- We adapt a single-objective heuristic approach to find the most cost efficient itinerary given a set of travel preferences.
- We adapt a multi-objective framework to find a set of trade-off travel itineraries which are both cost and time efficient. These itineraries help the customer on the decision making process once they compile all the best or near best travel options according to traveler's preferences.

1.5 Thesis outline

This thesis has been divided into 7 chapters:

Chapter 1, the current one, presents the motivations for this research project and states its problem, objectives and structure.

Chapter 2 presents a literature review on the travel planning field as a whole.

Chapter 3 shows how we have setup the configurations of our travel data base, as well as how he have collected real world travel data to conduct our experiments.

Chapter 4 shows how we formally address the travel data besides setting the base structure applied to model the problem. This chapter also presents the general strategy we used to find a travel itinerary on a fixed sequence of destinations in parallel. Additionally, it describes how the input data can be adapted to a GUI application, and how the solution can be presented to the traveler.

Chapter 5 describes the strategies applied to solve the single objective version of the travel planning problem, i.e., determining what is the travel plan with the lowest price. Besides formalizing the problem through an integer programming formulation, this chapter also shows how the problem was approached both exactly, using a parallel brute force algorithm, and heuristically, using 9 configurations of the Iterated Local Search (ILS) algorithm. At the end, we provide an assessment of the feasibility and quality of these configurations when compared to the exact approach.

Chapter 6 describes the strategies applied to solve the bi-objective version of the travel planning problem, i.e., determining what is the set of trade-off solutions that minimize both price and waiting. After introducing some concepts regarding the multi-objective field, we present a parallel brute force algorithm to solve the problem to its optimality and also show how we have adapted a heuristic framework (non-dominated sort algorithm II or NSGA-II) to rapidly reach approximated results. Finally, we assess 12 configurations of our heuristic algorithm against the optimal values and discuss which of them is suitable for each situation.

Chapter 7 outlines the results obtained in the research project as well as recommendations for further improvements.

Figure 2 portrays an overview of the relations between the chapters of this thesis. Chapters 5 and 6 present two different strategies for the problem considering a single background knowledge showed in previous chapters.

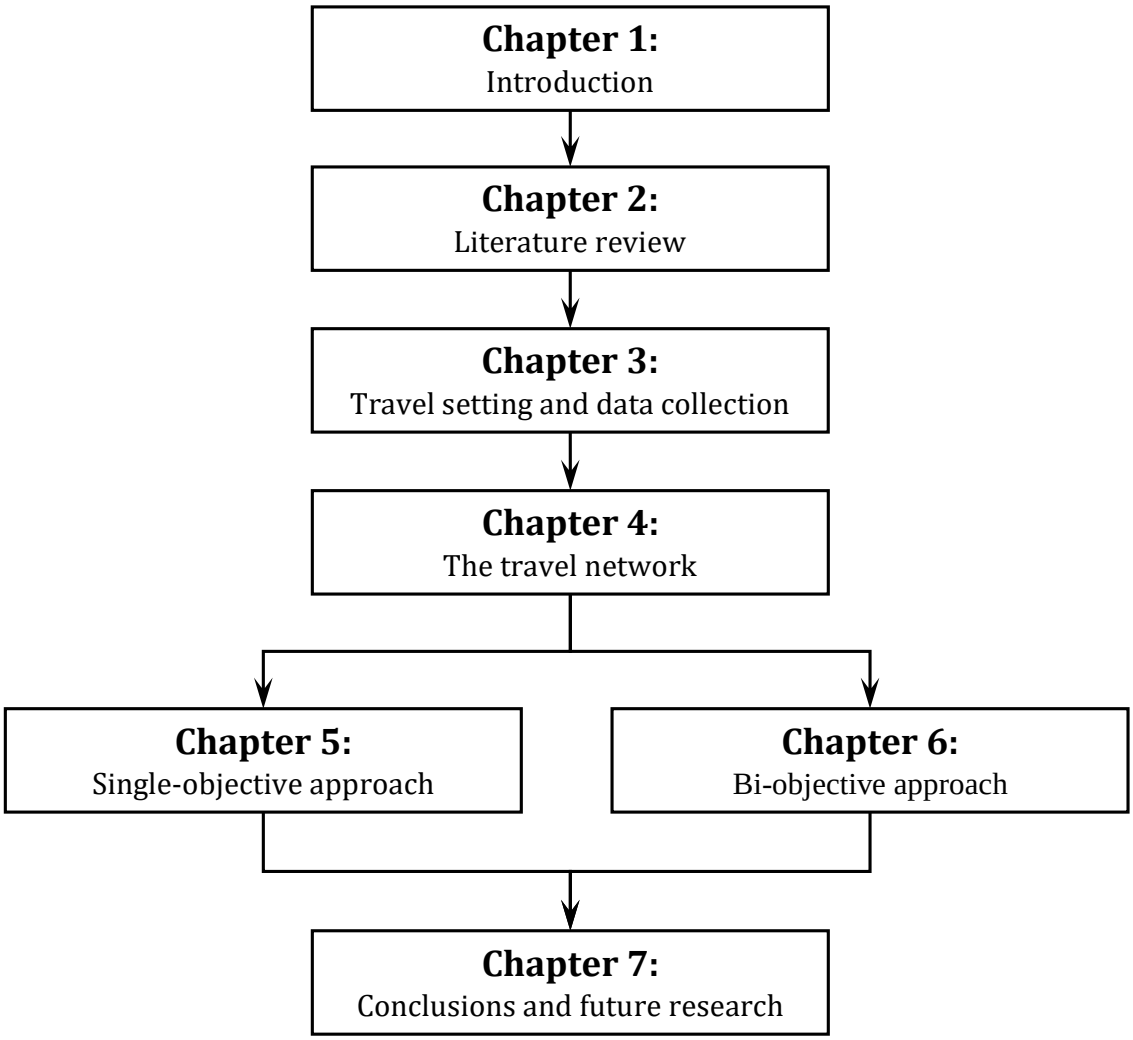

Figure 2 – Outline of the study.

2 Literature review

As mentioned in Section 1.2, to the best of our knowledge, only Li, Zhou and Zhao (2016) investigates a solution for the travel planning problem as we have defined it. Hence, in this chapter, we compile all relevant studies concerning travel planning as a general topic. We have divided the planning activity into two levels, depending on the way the problem is approached: 1) as a list of cities to be visited (Section 2.1) or 2) as a list of points of interest inside a city that should be visited (Section 2.2). Although the second level is not covered in this thesis, its related studies provide some insights about the current optimization strategies regarding the tourism field.

2.1 Intercity travel planning

We consider the first level of travel planning as a broader scheduling activity in which a tourist has to decide: 1) the departure date; 2) the destinations, their visiting order and dwelling times; 3) transportation options (plane, train, bus, etc.) to move between cities. This problem has a prominent resemblance with one of the most studied routing problems of all times: the Traveling Salesman Problem (TSP). In its simpler version, the problem can be formulated as follows: given n cities and a distance matrix $d_{n,n}$, where each element d_{ij} represents the distance between the cities i and j, find a tour that minimizes the total distance (TALBI, 2009). Since its general formulation by Karl Menger in 1932 (as cited by Desrochers, Lenstra and Savelsbergh (1990)), the problem has been very popular over the last decades and several generalizations and variants have been studied. Some of them are described in (GUTIN; PUNNEN, 2006), such as: time-dependent TSP, TSP with time windows, Period TSP, black and white TSP, Angle TSP, resource constrained TSP, Ordered Cluster TSP, minmax TSP, stochastic TSP and k-best TSP. Besides that, TSP is the base framework to formulate other practical problems as for example the deliveryman problem, the vehicle routing problem and the orienteering problem, which is another name for the selective TSP, a variant further explained in Section 2.2. Besides that, multi-objective approaches of the problem were also extensively studied as shown in (LUST; TEGHEM, 2010).

Regarding the solving strategies, besides the $O(n!)$ brute force search that tries all possible permutations, classic exact algorithms were also proposed such as dynamic programming (BELLMAN, 1962), branch-and-bound (LITTLE et al., 1963) and cutting-plane-method (DANTZIG; FULKERSON; JOHNSON, 1954). Approximation techniques are also frequently applied as a way of decreasing the execution time in real world applications. Some popular strategies, for example, are based on constructive heuristics, such as nearest-neighbor and Christofides' algorithm (as cited by Laporte (1992)), and evolutionary algorithms such as simulated annealing

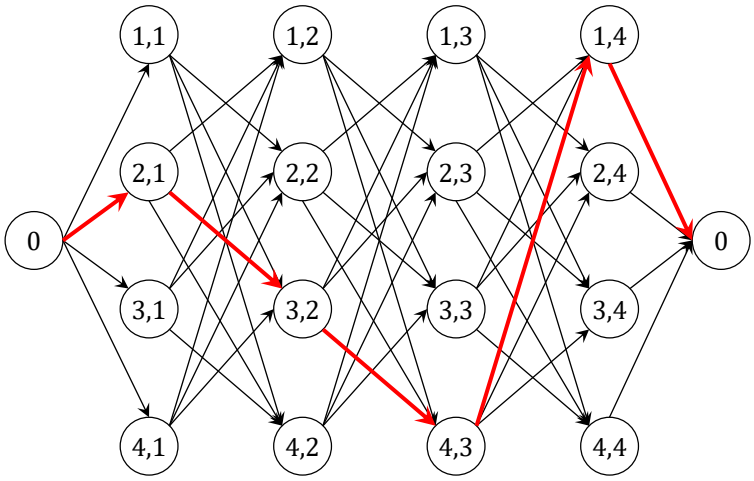

Figure 3 – Example of the TDTSP applied to a one-machine sequencing problem with 4 jobs and 4 positions (PICARD; QUEYRANNE, 1978). The path (2,3,4,1) is highlighted.

(PEPPER; GOLDEN; WASIL, 2002), genetic algorithm (CHATTERJEE; CARRERA; LYNCH, 1996) and ant colony optimization (DORIGO; GAMBARDELLA, 1997). Many of these techniques also have a parallelized versions. Parallel genetic algorithms, for example, have already been used to explore the processing power of a cluster (ER; ERDOGAN, 2014) and also of a GPU chip (KANG et al., 2016).

Many studies however, assume that all the information necessary to formulate the problems are time-invariant what is not usually verified in practical applications (GENDREAU; GHIANI; GUERRIERO, 2015). As explained in Section 1.1, the travel planning problem in fact requires a time-dependent approach since flight and accommodation related data change over time. This characteristic is properly captured by the afore mentioned time-dependent TSP (TDTSP), a generalization in which the cost c_{ij}^t of the travel between cities i and j depends on the position t of the transition (i, j) in the tour (WIEL; SAHINIDIS, 1995). In Figure 3 we show a classic example produced by Picard and Queyranne (1978) in which they apply the TDTSP framework to model an one-machine sequencing problem. In this example, the setup cost depends not only on the machines involved but also on their position in the sequence. Each node (j, t) represents the job j in the position t, and the initial state of the machine is denoted by 0. In Figure 4 we present an alternative representation for this example according to Abeledo et al. (2013). It considers a complete graph with 5 states (0 to 4) and 4 positions (1 to 4) in which nodes are represented by intersections between lines in a grid. The translation of the example showed in Figures 3 and 4 to a basic travel planning problem is rather straightforward: in this context, each node can represent a city and its departure instant. However, this simple representation is not well suited to hold details about a complete journey.

As mentioned in Section 1.2, Bérubé, Potvin and Vaucher (2006) provide a more

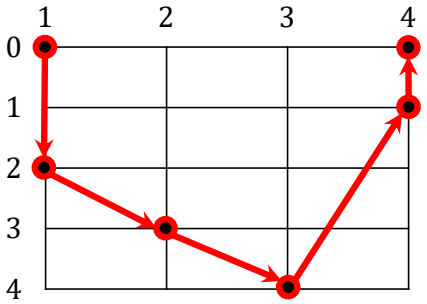

Figure 4 – Alternative graphical representation of the path (2,3,4,1) presented in Figure 3.

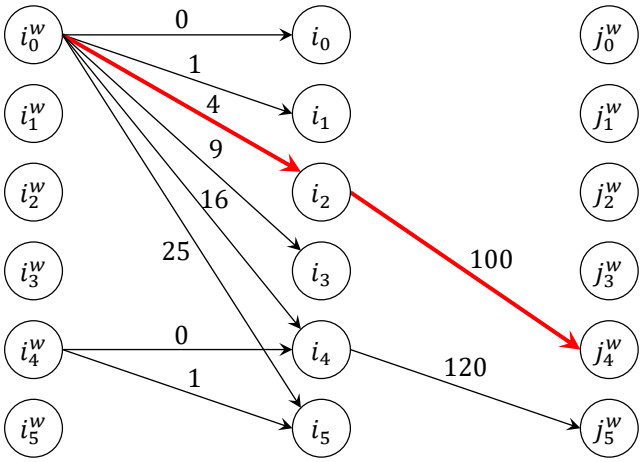

Figure 5 – Example of a space-time network of cities and flight connections (BÉRUBÉ; POTVIN; VAUCHER, 2006). The highlighted path is comprised by a) a waiting arc (i_0^w, i_2) which costs 4 and is 2 time units long and b) a travel arc (i_2, j_4) which costs 100 and is also 2 time units long.

complex model for a simpler version of the travel planning problem in which a single route of cities must be visited throughout many options. They propose a space-time network of cities and flight connections that also allows travelers to wait using a time-dependent shortest path problem (TDSPP) as a background framework. Figure 5 shows an example of an excerpt of such network in which are presented the waiting options of city i and the flight connections between i and j. An arc (i_{t1}, j_{t2}) represents a travel from city i to city j, departing at time $t1$ and arriving at time $t2$, with a cost $c_{ij}(t1)$ and delay $d_{ij}(t1) = t2 - t1$. In turn, an arc (i_{t1}^w, i_{t2}) represents a waiting period in city i from time $t1$ to time $t2$ and also has an associated cost and delay. In this particular model, the authors consider there are no flights departing from a city at the same time.

2.2 Inner city travel planning

Generally, the intercity planning is followed by a second planning level that takes place within the limits of each destination. At this bounded level, the travelers' goal

is to choose the worth visiting points of interest (POIs) as well as their order of visit besides defining the time spent at each point in regard to the daily time available for sightseeing (CHOUDHURY et al., 2010). As cited in (GAVALAS et al., 2014), personalized electronics tourist guides (PETs) may be used to derive customized routes and, as a result, are being incorporated in web and mobile applications. Usually, these guides comprise the following functionalities:

Recommendation: a list of POIs for each different tourist profile is generated. Each POI in the list is associated with a profit and a visit duration.

Route generation: algorithms are applied to generate personalized tourist itineraries throughout a combination of the list of POIs, the tourist-related data, the POI-related data and the transportation data.

Customization: tourists are able to modify the generated personalized route, i.e., add, remove or reorder POIs, according to their needs.

According to Gavalas et al. (2014), the generic problem of personalized tourist route generation, which is mainly associated with the route generation functionality of mobile tourist guides and PETs, has been defined as the tourist trip design problem (TTDP). The TTDP was first proposed by Vansteenwegen and Oudheusden (2007) and has its roots in the Orienteering Problem (OP), first studied by Tsiligirides (1984) and also known as the selective TSP (GENDREAU; LAPORTE; SEMET, 1998). As Gunawan, Lau and Vansteengwegen (2016) points out, OP is a combination of node selection and determining the shortest path between a set of selected nodes. Considering every location has a certain attraction value and the travel time between the points is known, the goal is to determine a route, limited in time, between some of the points to maximize the total value (VANSTEENWEGEN; OUDHEUSDEN, 2007). The similarity with TTDP becomes clear when we consider that these attraction values are assigned to locations based on tourist profiles and, besides that, the optimization process accounts for tourist, POI and transportation data.

As well as TSP, OP also has a generalized formulation in which the travel time between two nodes is time-dependent. To get to the next POI, for example, a tourist can choose between walking and using public transportation (GARCIA et al., 2013). Indeed, this last option impose delays related to the vehicles schedules and the tourist's boarding moment. In summary, as cited in (GUNAWAN; LAU; VANSTEENWEGEN, 2016), when the travel time between two nodes depends on the departure time at the first node we have a time dependent OP (TDOP). Another popular variations include the OP with time windows (OPTW) that imposes a restriction on the time a tourist can visit a point, the OP with multiple time windows (OPMTW) that considers more than a time window per location and the team

OP (TOP) in which the goal is to determine multiple tours (GUNAWAN; LAU; VANSTEENWEGEN, 2016). Some authors also explore the multi-objective facet of the problem. Mei, Salim and Li (2016) for example, investigate a multi-objective time-dependent OP (MOTDOP) to handle two of the most important factors when creating a path: time-dependent travel time and multiple preferences. Finally, since TTDP is based on OP, it also benefits from this fruitful variability of modeling options and can be resolved by solving an OP.

Similarly to the travel planning problem approached in this study, a practical solution for the TTDP is also expected to operate on an extremely time limited environment. For example, if the sequence of visits is revised by the tourist or he choses to stay longer in a POI the optimization method must be capable of performing an instantaneous re-plan (VANSTEENWEGEN; OUDHEUSDEN, 2007). Due to this time restriction, many studies approach the problem using heuristic strategies such as iterated local search (VANSTEENWEGEN et al., 2009), tabu search (SYLEJMANI; DIKA, 2011), simulated annealing (LIN; VINCENT, 2015) and memetic algorithm (DIVSALAR et al., 2014). In the other hand, some studies are more focused on reaching the global optima by applying exact techniques such as branch-and-bound (LEIFER; ROSENWEIN, 1994), dynamic programming (RIGHINI; SALANI, 2006) and branch-and-cut and price (POGGI; VIANA; UCHOA, 2010).

Apart from optimization strategies, other studies make use of traveler's geographical locations to mine promising visiting trends. Zheng et al. (2009), for instance, use GPS trajectories representing people's location histories to identify classical travel sequences between interesting locations, such as squares, public areas, shopping malls, restaurants, etc. Chen, Shen and Zhou (2011) also investigate the problem of discovering the most popular route between two locations by examining a collection of trajectories that represent traveling behaviors. In turn, Lu et al. (2010) use geotagged photos collected online to suggest customized travel route plans according to users' preferences.

In turn, other authors mix characteristics of both levels of travel planning in information or recommendation systems. Camacho, Borrajo and Molina (2001) for instance, propose a multi-agent information system that provides convenient itineraries considering user's preferences for a one destination trip. However, this system addresses a restricted amount of travel-related information besides focusing only on time minimization. In contrast, our approach is expected to deal with an expressive amount of real-world data to define a convenient broader itinerary through one or more destinations, considering different aspects of a touristic journey.

3 Travel setting and data collection

This chapter describes how we created a realistic travel data pool and how we set the traveler's default preferences that guide the decision making process of our optimization methods. Firstly, in Section 3.1, we explain what was the criteria to choose the destinations that compose our test cases. Next, in Section 3.2, we present our travel time window, i.e., the period in which the destinations will be visited. Then, in section 3.3, we show how real world flight data was collected from a Google API. Next, in Section 3.4, we explain how accommodation data was collected and what was the criteria to choose an option over another. Finally, in Section 3.5 we set the tourist's preferences regarding waiting and dwelling times.

3.1 Destinations

A report produced by Geerts (2016) ranks the top 100 city destinations in 2014 based on city arrivals. An arrival is related only to international tourists, i.e., any person visiting another country for at least 24 hours, for a period not exceeding 12 months, and staying in collective or private accommodation. According to this report, Europe is the most visited region of the world with 33% of popular destinations. Therefore, we chose 15 of these destination to compose our test cases. The selected cities, their countries, codes and number of arrivals are shown in Table 1. As a geographical reference, we also show the location of these cities in Figure 6.

Besides these 15 destinations, we chose Rio de Janeiro (RIO) to be our default starting city. This choice aims to mimic a travel in which a tourist from another continent visits some European destinations and then return to home coun-

Table 1 – List of the 15 European destinations chosen to compose the test cases of this study.

City	Country	IATA Code	#Arr. 2014 (mi)
London	England	LON	17.38
Paris	France	PAR	14.98
Istanbul	Turkey	IST	11.87
Antalya	Turkey	AYT	11.49
Rome	Italy	ROM	8.78
Milan	Italy	MIL	6.05
Barcelona	Spain	BCN	5.97
Amsterdam	Netherlands	AMS	5.71
Moscow	Russia	MOW	4.97
Berlin	Germany	BER	4.67
Madrid	Spain	MAD	4.17
Brussels	Belgium	BRU	3.10
Munich	Germany	MUC	3.02
Zurich	Switzerland	ZRH	2.37
Frankfurt	Germany	FRA	2.01

Figure 6 – Geographical location of the 15 destination cities chosen (map from https://www.google.cm/maps)

try. Therefore, any route must start in Rio, pass through one or many of the 15 selected destinations, and finally come back to Rio. Consequently, the first and the last slice of the total route will be the longest ones due to the continental distances involved.

The three-letter code adopted is issued by the International Civil Aviation Organization (IATA). This entity is a trade association for the world's airlines which represent 83% of total air traffic, supporting many areas of aviation activity besides helping to formulate the industry policy (IATA, 2016). IATA also has standardized codes for countries, airlines, taxes, aircrafts, timezones, routes and many other aviation details.

3.2 Travel time window

All travels must start and terminate between 2015/12/01 to 2016/01/10. This period of 41 days was chosen because it comprises two of the most important holidays of the year: Christmas and New Years' Eve. As shown in Figure 7, a further analysis on our travel data suggests that prices tend to increase around these dates, probably due to higher demand. The black bars represent the average minute cost of Christmas and New Years' Eve and the vertical lines separate weeks that start on Mondays

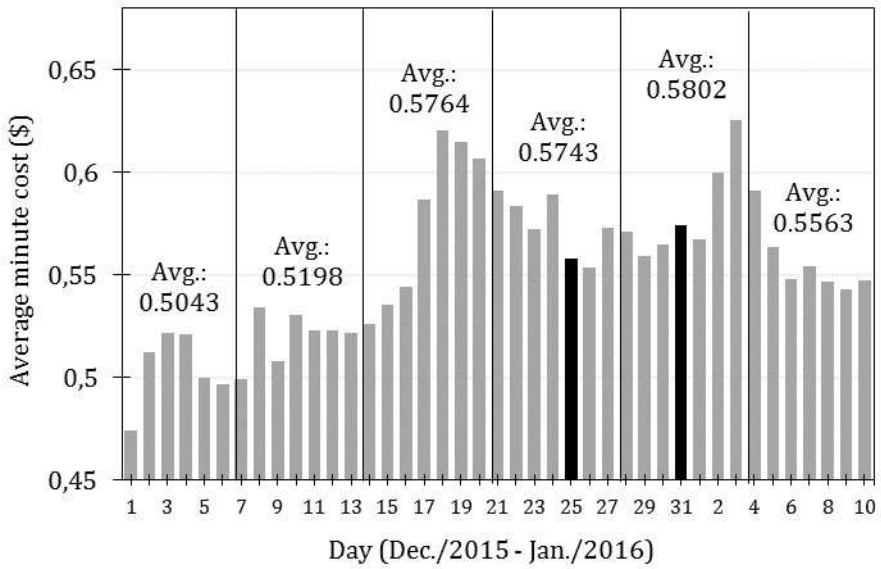

Figure 7 – A bar corresponds to the average price of all the 1440 minutes of each day that comprises the total time window. The price of a single minute is the average price of all available trips containing this minute in their departure/arrival interval. For example, if a travel costs 50$ and starts at minute 400 and ends at minute 500, we have a 100 minutes' travel, and each minute of this travel has a partial minute cost of $0.5. Since a single minute in the travel windows may cover several travels, its final value will be the average of the partial minute costs of all these travels.

and finish on Sundays. Inside each week box, we show the average minute cost of all days of that week. In average, the most expensive week to travel covers the New Years' Eve and the second most expensive week to travel is right before the Christmas week. Knowing about these busy periods is important when analyzing the behavior of the algorithm regarding the placement of the travels in the total travel span. Shorter itineraries, for example, would probably avoid the busiest and most expensive periods to travel.

3.3 Flights

All transportation data were collected using the Google QPX Express API. This API responds in real time to changing fares and availability, searching through several combinations of airline schedules, fares, tax rules, and seat availability (GOOGLE, 2016). Once flight data are extremely sensible to demand, prices and availability may change almost instantaneously, especially when data is collected close to the departure time. Therefore, in order to improve data consistency, we performed all data collection from 2015/07/21 to 2015/07/29, i.e., about 4 months before the start of the travel time window.

Each flight option returned by the API contains detailed information about a

```
{
  "request": {
    "slice": [
      {
        "origin": "RIO",
        "destination": "AMS",
        "date": "2016-07-13"
      }
    ],
    "passengers": {
      "adultCount": 1
    },
    "saleCountry": "US"
  }
}
```

Figure 8 – Example of a travel request sent to QPX Express API. It searches for trip options from Rio de Janeiro to Amsterdam departing in 2016/07/13 for a single passenger. The field "saleCountry" sets the currency of the fares to US dollars.

travel involving a city pair and a departure date. Since we have 16 cities counting with our start city, there are 240 (16x15) different city pairs. Additionally, once a flight may occur in any day of the time window for each pair, we executed 9.840 (240x41) queries on Google server to download all flight data. A query consists in a travel request similar to the example depicted in Figure 8, which searches for trip options from Rio de Janeiro (RIO) to Amsterdam (AMS) that depart in 2016/07/13. The request is written in JSON (JavaScript Object Notation), a lightweight data-interchange format based on a subset of the JavaScript programming language (ECMA, 2013).

For each travel request, the QPX server outputs a JSON object which contains an array containing all trip options available. This array has up to 500 itineraries that include all data elements needed for customer display and booking engines. In general, the greater the popularity of a route the bigger is the variety of options. The API also provides a formatted response viewer whose goal is to highlight the most important features of each solution. In Figure 9 we show the 3 first formatted solutions of 186 found for the query in Figure 8. Solution 1, for example, shows a trip option that costs 418.33 dollars and has 3 connections. It departs from GIG (Galeao International Airport - Rio de Janeiro, Brazil), stops in GRU(Guarulhos International Airport - São Paulo, Brazil) and subsequently in CMN (Mohammed V International Airport -Casablanca, Morocco) to finally arrive in AMS (Amsterdam Airport Schiphol - Amsterdam, Netherlands).

The arrival/departure times involved in any airport pair are, respectively, the

```
186 solutions found.

Solution# 1   Sale Price: USD418.33
   Slice 0
      AT 213 GIG 2016-07-13T18:25-03:00 CMN 2016-07-14T09:50+01:00
         GIG 2016-07-13T18:25-03:00 GRU 2016-07-13T19:35-03:00
         GRU 2016-07-13T20:40-03:00 CMN 2016-07-14T09:50+01:00
      AT 850 CMN 2016-07-14T12:25+01:00 AMS 2016-07-14T16:50+02:00

Solution# 2   Sale Price: USD698.77
   Slice 0
      G36817 SDU 2016-07-13T17:20-03:00 GRU 2016-07-13T18:35-03:00
      AT 213 GRU 2016-07-13T20:40-03:00 CMN 2016-07-14T09:50+01:00
      AT 850 CMN 2016-07-14T12:25+01:00 AMS 2016-07-14T16:50+02:00

Solution# 3   Sale Price: USD1131.35
   Slice 0
      G35837 GIG 2016-07-13T16:50-03:00 GRU 2016-07-13T17:55-03:00
      AT 213 GRU 2016-07-13T20:40-03:00 CMN 2016-07-14T09:50+01:00
      AT 850 CMN 2016-07-14T12:25+01:00 AMS 2016-07-14T16:50+02:00
```

Figure 9 – Formatted response stressing the most important features of the three first solutions found (within 186 available) for the travel request exhibited in Figure 8.

current times in origin and destination cities of these airports even when daylight saving times are concerned. The trailing hours after the signs `"-"` and `"+"` of each time stamp are offsets that indicate what is the time zone of each airport considering the UTC standard. The UTC (Coordinated Universal Time) is a time scale that separates world locations in 24 time zones being the last zone (Z or "Zulu") the world reference. In solution 1, for example, the first departure time occurs in Rio de Janeiro at `"2016-07-13T18:25-03:00"` and the last arrival time is in Amsterdam at `"2016-07-14T16:50+02:00"`. If we subtract the offsets to get the UTC Z reference we would have the following times for departure and arrival respectively: `"2016-07-13T21:25"` and `"2016-07-14T14:50"`. The total duration of the trip is the difference between these UTC times: 17 hours and 25 minutes (or 1045 minutes).

In contrast with the summarized solutions shown in Figure 9, A complete JSON response comprises all relevant flight data involved in a trip such as airports, cities, taxes, airlines, aircrafts, etc. In Appendix A we go through the complete response returned by the server, presenting all relevant trip features. However, although these features are required to create and portray a detailed itinerary for the passenger, many of them are unnecessary to our heuristic methods. In fact, as showed in Section 4.2 we only use some general features of each trip option to build our travel network, such as cost and departure/arrival times. Then, we created a summarized version of the trip option object (Figure 10) containing only the relevant information which is in fact used by our optimization algorithms. The remaining attributes that compose the summarized trip option objects are the following:

```
"tripOption": [{
  "saleTotal": "USD418.33",
  "id": "b4t4y5teOKPRSKYSYZdVu0001",
  "slice": [{
    "duration": 1045,
    "segment": [{
      "leg": [{
        "departureTime": "2016-07-13T18:25-03:00",
        "origin": "GIG",
      }]
    },
    {
      "leg": [{
        "arrivalTime": "2016-07-14T16:50+02:00",
        "destination": "AMS"
      }]
    }]
  }]
}]
```


Figure 10 – Shortened trip option containing only the relevant information that is indeed used by optimization methods of this study.

- **id:** unique identifier of the trip option. Our method generates a sequence of summarized trip options that can be further converted into a detailed itinerary by using this attribute to query additional travel information.
- **saleTotal:** total cost of the trip option considering fares and taxes.
- **duration:** total duration in minutes of the trip option including possible connections.
- **departureTime:** departure date and time of the flight operated in the first leg of the first segment.
- **origin:** code of the first departure airport among legs, i.e., the initial point of a the travel.
- **arrivalTime:** arrival date and time of the flight operated in the last leg of the final segment.
- **destination:** code of the last arrival airport among legs, i.e., the final point of a travel.

Each shortened trip object is then converted into a travel line of a flight data file that feeds up our algorithms. Each travel line consists on a travel arc of our graph formulation (Section 4.2) that possibly composes a final solution. In order

streamline the graph building process we preprocessed the summarized trip options, adding another relevant information besides combining some data. Initially, we combined the “departureTime” and “origin” attributes to create a departure label and the “arrivalTime” and “destination” attributes to create an arrival label. These labels will eventually identify our travel nodes as described in Section 4.3.

Table 2 shows an example of the collected flight data containing the 25 cheapest travels from Rio de Janeiro to Amsterdam departing in 2015/12/15. Although some travels have equal costs, they differ in departure/arrival times or duration. The airports involved are represented by their IATA codes: SDU (Santos Dumont - Rio de Janeiro, Brazil), GIG and AMS. The letters `"o"` and `"w"` in front of the airport’s codes are helper symbols (Section 4.3) required during the graph’s building process.

Table 2 – The 25 cheapest travel options from Rio de Janeiro (RIO) to Amsterdam (AMS) on 2015/12/15 sorted by travel cost. Travels with the same costs are s

Id	Dep. city	Arr. city	Departure label	Arrival label	Cost ($)	Duration (min.)
OW5RAiMU1wJMJQtbgH4WW3001	RIO	AMS	oSDU|2015-12-01T06:25-02:00	wAMS|2015-12-02T10:05+01:00	765.55	1480
OW5RAiMU1wJMJQtbgH4WW300F	RIO	AMS	oSDU|2015-12-01T06:25-02:00	wAMS|2015-12-02T13:35+01:00	765.55	1690
OW5RAiMU1wJMJQtbgH4WW300B	RIO	AMS	oSDU|2015-12-01T06:25-02:00	wAMS|2015-12-02T17:45+01:00	765.55	1940
OW5RAiMU1wJMJQtbgH4WW300J	RIO	AMS	oGIG|2015-12-01T07:43-02:00	wAMS|2015-12-02T10:05+01:00	766.34	1402
OW5RAiMU1wJMJQtbgH4WW300c	RIO	AMS	oGIG|2015-12-01T07:43-02:00	wAMS|2015-12-02T13:35+01:00	766.34	1612
OW5RAiMU1wJMJQtbgH4WW3006	RIO	AMS	oGIG|2015-12-01T07:43-02:00	wAMS|2015-12-02T17:45+01:00	766.34	1862
OW5RAiMU1wJMJQtbgH4WW3005	RIO	AMS	oGIG|2015-12-01T19:58-02:00	wAMS|2015-12-02T15:40+01:00	1,249.61	1002
OW5RAiMU1wJMJQtbgH4WW3004	RIO	AMS	oGIG|2015-12-01T19:58-02:00	wAMS|2015-12-02T17:40+01:00	1,249.61	1122
OW5RAiMU1wJMJQtbgH4WW3008	RIO	AMS	oGIG|2015-12-01T19:58-02:00	wAMS|2015-12-02T15:40+01:00	1,249.61	1002
OW5RAiMU1wJMJQtbgH4WW300E	RIO	AMS	oSDU|2015-12-01T11:35-02:00	wAMS|2015-12-02T10:00+01:00	1,252.35	1165
OW5RAiMU1wJMJQtbgH4WW3002	RIO	AMS	oSDU|2015-12-01T12:19-02:00	wAMS|2015-12-02T13:35+01:00	1,252.35	1336
OW5RAiMU1wJMJQtbgH4WW300j	RIO	AMS	oSDU|2015-12-01T12:19-02:00	wAMS|2015-12-02T10:00+01:00	1,252.35	1121
OW5RAiMU1wJMJQtbgH4WW300e	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T19:05+01:00	1,451.27	1330
OW5RAiMU1wJMJQtbgH4WW300G	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T20:00+01:00	1,451.27	1385
OW5RAiMU1wJMJQtbgH4WW300C	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T19:05+01:00	1,451.27	1330
OW5RAiMU1wJMJQtbgH4WW300L	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T13:55+01:00	1,451.27	1020
OW5RAiMU1wJMJQtbgH4WW300K	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T20:00+01:00	1,451.27	1385
OW5RAiMU1wJMJQtbgH4WW300O	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T18:05+01:00	1,451.27	1270
OW5RAiMU1wJMJQtbgH4WW3009	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T11:45+01:00	1,451.27	890
OW5RAiMU1wJMJQtbgH4WW300T	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T11:45+01:00	1,451.27	890
OW5RAiMU1wJMJQtbgH4WW3003	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T10:25+01:00	1,451.27	810
OW5RAiMU1wJMJQtbgH4WW300D	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T13:05+01:00	1,451.27	970
OW5RAiMU1wJMJQtbgH4WW300P	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T11:15+01:00	1,451.27	860
OW5RAiMU1wJMJQtbgH4WW300A	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T11:15+01:00	1,451.27	860
OW5RAiMU1wJMJQtbgH4WW3007	RIO	AMS	oGIG|2015-12-01T17:55-02:00	wAMS|2015-12-02T13:05+01:00	1,451.27	970

3.4 Hotels

Accommodation data were collected manually from Expedia (expedia.com), an accommodation website that provides budget beds. Once our formulation does not intend to account for beds availability in different hotels, we collected only a unique accommodation cost per city and day of travel window. Despite this cost being chosen arbitrarily, it followed a simple pattern: the bed should be cheap, preferably in a shared dorm and belong to a well-rated hotel. Figure 11 shows average accommodation costs for the 15 destinations given the time window of 41 days. The average sized spikes stand for the weekends and the biggest spike represents an increase in accommodation cost due to the New Years' Eve. Concerning the accommodations' arrival/departure policy, we consider that all hotels have equal hours of check-in (14h) and check-out (12h).

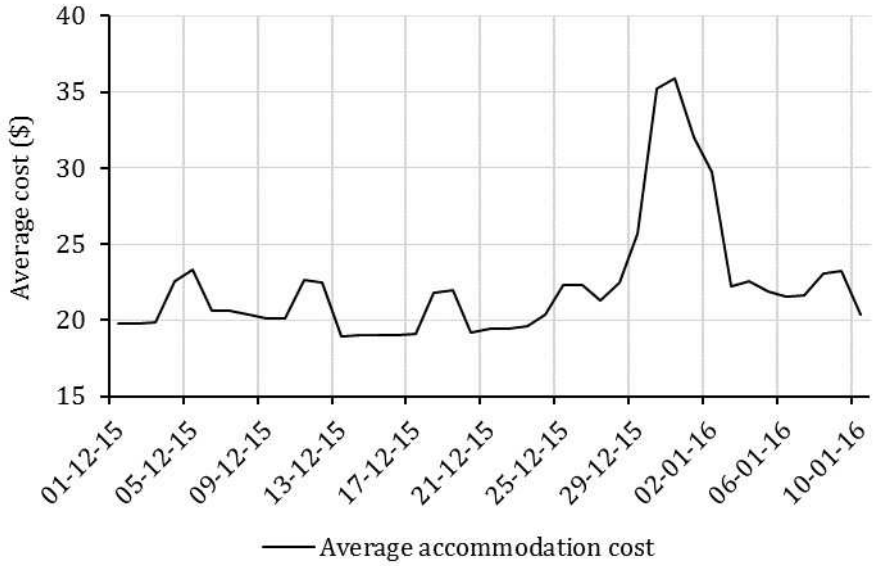

Figure 11 – Average dwelling cost for the 15 destination cities on each date of the time window.

3.5 Dwelling and waiting

We assume that the desired dwelling time in each destination is known beforehand, so that a tourist can buy all flight's tickets and book all hotels in advance. In an ideal itinerary a tourist would always arrive in the city right after the check-in time and leave the city right before check-out time. This optimal settlement of arrival and departure times is rather convenient: the tourist would always have access to a bedroom during its stay in a city besides taking full advantage of the paid accommodation time. Since real world itineraries often do not follow this ideal schedule, we also want to minimize the waiting time, whether to board transportation or to get a bed in an accommodation. Doing so, we contribute to ease the annoyance of

not having access to a proper room once itineraries with large waiting times tend to be avoided. Therefore, in order to ensure that a tourist would not spend many hours without a bed we created a policy to decide whether a tourist: 1) pays for the previous night (when arrival is much earlier than check-in time), 2) pays for the next night (when departure is much later than check-out) and 3) waits for check-in or departure times.

In some cases, a tourist is also prone to wait, and accommodation's managers are aware of this trend. Excepting the bedroom, tourists usually have access to the other dependencies of an accommodation around the booked time, i.e., the time between check-in and check-out. In fact, many hotels have waiting rooms where tourists are able to wait for a bed when arriving, or for a transportation when leaving the city. When the arrival time is close to the check-in time for example, some tourists prefer to wait inside the accommodation dependencies until having access to a proper room. Others will prefer to lock their luggage inside or outside the accommodation and start their sightseeing activities, going back to the accommodation after the check-in time has passed. In contrast, when the departure time of the city is after the check-out time, the tourist may choose to spend the remaining time visiting other places and/or waiting for the departure in the accommodation or airport. However, for some tourists, having access to a bedroom is mandatory, mainly when the waiting hours are long enough to be early in the day or late in the night. Furthermore, how long a tourist is willing to wait is generally a subjective matter, but once our approach is tailored to a budget traveler we defined long possible waiting times around the time block defined by check-in and check-out times. Therefore, prior to check-in time there is a waiting window $wt_{cIn} = 7\text{h}$ which stands for the maximum time that a tourist is willing to wait before having access to a bed. In turn, there is a check-out time window $wt_{cOut} = 14h$ representing the maximum time a tourist may wait to departure without having access to a bed. Figure 12 shows all possible time blocks in which a tourist may arrive in a day d (I, II, III) and also all time blocks he may depart in the day $d + 1$ (III, IV, V).

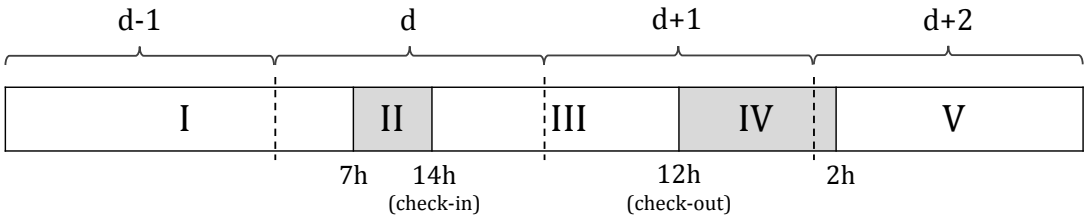

Figure 12 – Visit payment policy depending on arriving and departing times.

Considering the default check-in time $t_{cIn} = 14h00$ and the default check-out time $t_{cOut} = 12h00$, each block has a unique charging policy depending on the arrival time t_a in day d and the departure time t_d in day $d + 1$. First, regarding the arrival moment, the following time blocks may be reached:

I) $t_a < (t_{cIn} - wt_{cIn})$**:** Tourist arrives before the start of the waiting window for check-in and needs to book previous night.

II) $(t_{cIn} - wt_{cIn}) \leq t_a < t_{cIn}$**:** Tourist arrives within the waiting window, and must wait for $t_{cIn} - t_a$ minutes to have access to a room.

III) $t_a \geq t_{cIn}$**:** Tourist arrives within its "paid window". The current day thereafter is the start of the booking time.

Next, in regard to the departure moment, the following time blocks may be reached:

III) $t_d \leq t_{cOut}$**:** Tourist departs within the paid time having full access to the accommodation dependencies until the departure time. There is no need to pay any further amount since its stay began in the check-in time of the previous day.

IV) $t_{cOut} < t_d \leq (t_{cOut} + wt_{cOut})$**:** Tourist must wait for $t_d - t_{cOut}$ minutes to depart from city.

V) $t_d > (t_{cOut} + wt_{cOut})$**:** Tourist departs after the waiting window for check-out and needs to book the current night.

4 The travel network

This chapter comprises all the details related to the building and evaluation of a travel network created from a single sequence of visits. Firstly, in Section 4.1, we show how the travel data mentioned in Chapter 3 is formally represented. Then, in Section 4.2, we describe what we did to adapt the TDSPP framework so that it could properly hold our travel data. Next, Section 4.3 shows an example of this adaptation for a short permutation. Then, Section 4.4 presents a strategy to quickly build the travel network in parallel. Finally, Sections 4.5 and 4.6 describe how the classic Dijkstra algorithm is applied to determine the best travel itinerary and how this itinerary can be outlined to highlight its main aspects.

4.1 Data representation

In Chapter 3 we described all data required to properly plan a travel. In this section we go further and create a formal representation for all elements of this data set, so that we can easily address the data elements of the problem when designing approaches to solve it. In order to improve clarity, we separate the traveler preferences (Section 4.1.1) and the travel industry data (Section 4.1.2).

4.1.1 Traveler preferences

The traveler preferences are the customizable elements of an itinerary that must be provided by a traveler in order to build a tailored solution. These elements are represented as follows:

- I: Set containing the cities in which the journey may start.
- C: Set of destination cities, i.e., the cities in which the tourist intends to stay.
- F: Set containing the cities in which the journey may end.
- d_{start}: Minimum starting instant (date and time in minutes) to start the journey.
- d_{end}: Maximum ending instant (date and time in minutes) to finish the journey.
- $Tmin_c$: Minimum dwelling time in city $c \in C$.
- $maxExtraTime_c$: Maximum spare time in city $c \in C$ (beyond $Tmin_c$).

In Figure 13 we present a GUI prototype showing how this input data can be represented by input components in a travel planner application. The traveler preferences portrayed are the following: $I = F = \{$Rio de Janeiro$\}$, $C = \{$Paris, Lon-

don}, $Tmin_{RIO} = Tmin_{LON} = 3$, $maxExtraTime_{RIO} = maxExtraTime_{LON} = 3$, $d_{start} = 01/12/16$ and $d_{end} = 31/12/16$.

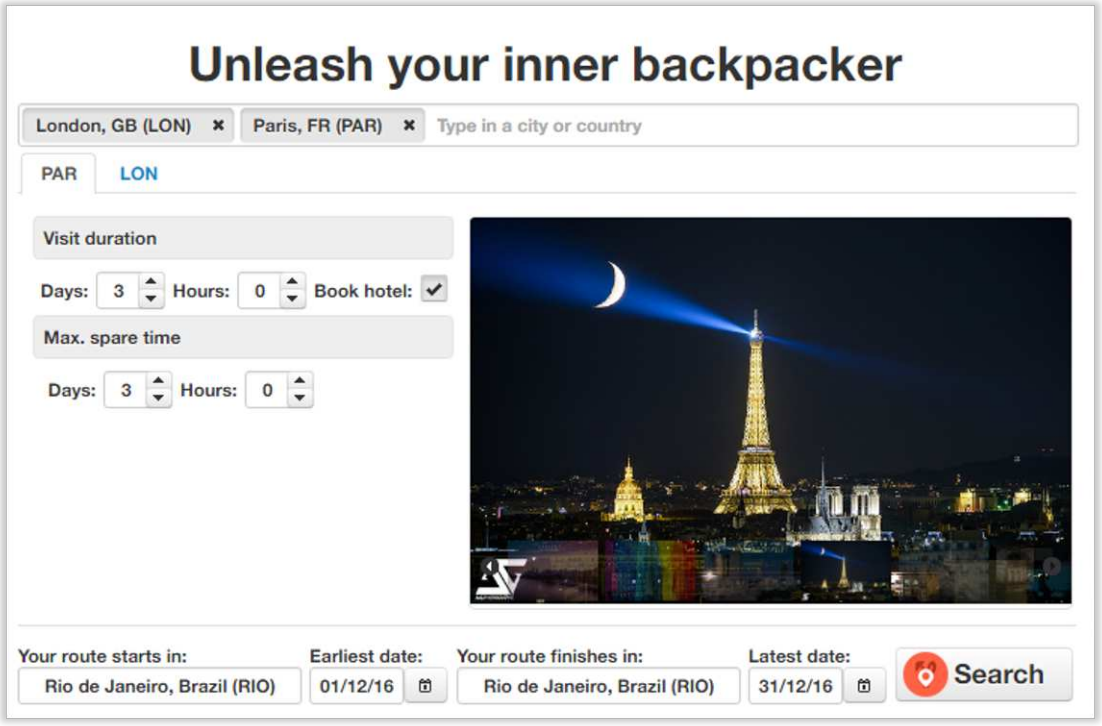

Figure 13 – GUI prototype of a travel planner application. Clicking on "Search" button would start the process of finding the best itinerary, including flights and accommodations, for a trip comprising Paris and London.

4.1.2 Travel industry

The travel industry data correspond to the flights/hotels information required to build a tailored solution for a given set of traveler preferences. We represent these data as follows:

- D_s: Set of departure airports of the city $s \in (I \cup C)$.
- A_e: Set of arrival airports of the city $e \in (C \cup F)$.
- $T_{i_s j_e}$: Set of departure times t_1 from airport $i_s \in D_s$ to airport $j_e \in A_e$. Additionally, $d_{start} \leq t_1 < d_{end}$.
- $T^{t_1}_{i_s j_e}$: Set of arrival times t_2 of travels to airport $j_e \in A_e$ coming from airport $i_s \in D_s$ in time $t_1 \in T_{i_s j_e}$. Additionally, $t_1 < t_2 \leq d_{end}$.
- $V_{flights}$: Set of existing flights. Each flight is defined by a tuple (s, e, i_s, j_e, t_1, t_2) representing a travel from city $s \in (I \cup C)$ to city $e \in (C \cup F)$ with $e \neq s$, departing of airport $i_s \in D_s$ in time $t_1 \in T_{i_s j_e}$ and arriving in airport $j_e \in A_e$ in time $t_2 \in T^{t_1}_{i_s j_e}$.

In order to improve clarity and legibility we defined the following helper sets:

- V_{stays}: Set of viable stays. Each stay is defined by a tuple (c, j_c, i_c, t_1, t_2) representing a stay in city $c \in C$, starting in the arrival airport $j_c \in A_c$ and finishing in airport $i_c \in D_c$. The t_1 starting time of the stay may be any arriving time of a travel departing from any airport $i_s \in D_s$ in which city $s \in (I \cup C)$ with $s \neq c$. Therefore, considering t_d as the departure time of an airport i_s, $t_1 \in T^{t_d}_{i_s j_c}$. In turn, the t_2 end time of the stay may be any t_2 departure time from the airport i_c to any airport $j_e \in A_e$ of city $e \in (C \cup F)$ with $e \neq c$. Hence, $t_2 \in T_{i_c j_e}$, with $t_2 > t_1$. Additionally, $Tmin_c \leq t_2 - t_1 \leq Tmin_c + maxExtraTime_c$, i.e., only stay options in which the total duration complies to the minimum and maximum time in city c are included.

- AR_{j_e}: Set of arrival times in airport $j_e \in A_e | e \in (C \cup F)$. Therefore, $\forall \ \ j_e \in A_e | e \in (C \cup F)$, $AR_{j_e} = \{t_a | (\exists i_s \in D_s | s \in (I \cup C), s \neq e) \wedge (\exists t_d \in T_{i_s j_e} | t_a \in T^{t_d}_{i_s j_e})\}$.

- DP_{i_s}: Set of departure times in airport $i_s \in D_s | s \in (I \cup C)$. Therefore, $\forall \ \ i_s \in D_s | s \in (I \cup C)$, $DP_{i_s} = \{t_d | (\exists j_e \in A_e | e \in (C \cup F), e \neq s) \wedge (\exists t_d \in T_{i_s j_e})\}$.

- $V^{s,e}_{flights}$: Subset of $V_{flights}$ containing the flights from city s to city e.

Additionally, we also defined helper variables to easily address other components of travel and stay arcs:

- $tt^{t_1 t_2}_{i_s j_e}$: Duration of the flight tuple $(s, e, i_s, j_e, t_1, t_2) \in V_{flights}$.

- $st^{t_1 t_2}_{i_c j_c}$: Duration of the stay tuple $(c, j_c, i_c, t_1, t_2) \in V_{stays}$.

- $wt^{t_1 t_2}_{i_c j_c}$: Waiting time (Section 3.5) embedded in the total duration $st^{t_1 t_2}_{i_c j_c}$ of stay tuple $(c, j_c, i_c, t_1, t_2) \in V_{stays}$.

Finally, the costs of each travel and stay are stated as follows:

- $P^{t_1 t_2}_{i_s j_e}$: Price of the flight tuple $(s, e, i_s, j_e, t_1, t_2) \in V_{flights}$.

- $S^{t_1 t_2}_{i_c j_c}$: Price of the stay tuple $(c, j_c, i_c, t_1, t_2) \in V_{stays}$.

4.2 Graph formulation

According to Bérubé, Potvin and Vaucher (2006), searching for the best order of visits in a travel planning problem is a complex task and would require the use of heuristic or approximate methods. In fact, an important part of our strategy to solve the travel planning problem consists in heuristically generating sequences of cities and producing their correspondent travel graphs. Therefore, despite our

graph representation does not need to account for a complete network with all travel options at once, it does need to create a travel graph for a single sequence of cities. However, a regular graph representation G = (V, A), where $V = \{c_1, c_2, ..., c_n\}$ is the set of cities and $A = \{(s, e)|s, e \in V\}$ is the set of airline connections is not appropriate to hold flight data. In fact, each element in A only indicates that there is a connection from city s to city e, but does not specify the airports involved nor the departure/arrival times of the flight.

In order to build a proper space-time network, we first define a sequence of visits $S = (c_{start}, c_1, c_2, c_3, ..., c_n, c_{end})$ in which c_{start} represent the origin city where a travel must start and c_{end} represent the final city where a travel must end, i.e., $c_{start} \in I$ and $c_{end} \in F$. In turn, cities from c_1 to c_n are the destination cities in set C. From the S sequence we are able to define the set of possible from/to city pairs $P_2 = \{(S_i, S_{i+1})|S_i, S_{i+1} \in S, i \in \{0, ..., n\}\}$ which provides the isolated trips within the complete journey. Through the definition of P_2, we are finally able to determine the set of the travel graph's vertices $V = \{[s, i_s, t_1], [e, j_e, t_2]|(s, e) \in P_2, (s, e, i_s, j_e, t_1, t_2) \in V_{flights}\}$. Each possible flight between two cities in P_2 defines the departure and the arrival vertices at once. Hence, a vertex is identified by three information: a city, a departure/arrival airport and a departure/arrival time.

The set of flight arcs of a graph based on the sequence S is directly derived from the V set being defined as $A_{flights} = \{([s, i_s, t_1], [e, j_e, t_2]) \mid (s, e) \in P_2,\ [s, i_s, t_1], [e, j_e, t_2] \in V\}$. Therefore, any travel from city s to city e costs $P^{t_1 t_2}_{i_s j_e}$ monetary units, starts at time t_1 in airport i_s, finishes at time t_2 in airport j_e and lasts $t_2 - t_1$ time units. Furthermore, our approach acknowledges that a travel may start from different airports at the same time, and also from the same airport at the same time, as long as it finishes in different arrival times.

In turn, the set of dwelling arcs is always generated on the fly, based on the vertexes already set for the space-time network of a sequence S. One key characteristic of the travel vertexes that allows the creation of these arcs is the clear differentiation among departure and arrival airports: as shown in Section 4.1.2, we separate the departure airports $i_s \in D_s \ \ \forall s \in (I \cup C)$ from the arrival airports $j_e \in A_e \ \ \forall e \in (C \cup F)$. This differentiation enables us to define the possible dwelling times in a city $c \in C$ as arcs that start in an arrival airport j_c at time t_1, and finish in a departure node i_c at time t_2. Therefore, a dwelling arc can be seen as a travel from a city to itself, which abides by the following restrictions:

1. The start vertex (c, j_c, t_1) of an accommodation arc must be an arrival vertex of an arc whose departure vertex belongs the previous city in the sequence;

2. The end vertex (c, i_c, t_2) of an accommodation arc must be a departure vertex of an arc whose arrival vertex belongs to the next city in the sequence;

3. The total duration $d = (t_2 - t_1)$ of a dwelling arc $([c, j_c, t_1], [c, i_c, t_2])$ must be between $Tmin_c$ and $Tmin_c + maxExtraTime_c$.

Let $P_3 = \{(S_i, S_{i+1}, S_{i+2}) | S_i, S_{i+1}, S_{i+2} \in S, i \in \{0, ..., n-1\}\}\}$ be the set of possible sub-sequences of S in which the middle element is the city currently being visited, the first element is the previous city in the travel sequence and the last element is the next city in the travel sequence. Through P_3, the set of dwelling arcs can be defined as $A_{stays} = \{([c, j_c, t_1], [c, i_c, t_2]) | (s, c, e) \in P_3, [s, i_s, t_0], [c, j_c, t_1], [c, i_c, t_2], [e, j_e, t_3] \in V, t_2 > t_1, Tmin_c \leq t_2 - t_1 \leq (Tmin_c + maxExtraTime_c)\}$. In regard to the cost $S^{t_1 t_2}_{j_c i_c}$ of a dwelling arc, it primarily corresponds to the summation of the costs of all nights booked in the city's accommodation during the complete stay. Therefore, a dwelling arc in city c costs $S^{t_1 t_2}_{j_c i_c}$ monetary units, starts at time t_1 in airport j_c, finishes at time t_2 in airport i_c, lasts $t_2 - t_1$ time units wherein $wt^{t_1 t_2}_{j_c i_c}$ time units corresponds to waiting time. Finally, the set containing all arcs of a space-time network of a sequence of cities S is $A = (A_{flights} \cup A_{stays})$.

Our restrictions on vertices and arcs creation are extremely important to decrease the complexity of the model. In contrast, if all possible vertexes were created, many of them would never be used. Once our travel horizon is 41 days long and we are using real travel data, which is discretized in minutes, each arrival/departure airport of our space-time network would have at least 5.040 (41x24x60) time steps. Without any restriction, the number of dwelling options would also be bigger once it would comprise all possible options between the arrival and departure nodes. The extra time spent to build a simplified graph pays off since it becomes easier to find a solution for it.

Once the modeling is finished, we add two dummy nodes to execute the Dijkstra shortest path algorithm (explained in detail in Section 4.5). The first is a dummy start node, which is connected to all origin nodes of the starting city, and the second is a dummy end node in which we connect all final nodes of the starting city. The arcs created to accomplish these connections do not hold any values: they serve only as a medium to access all departure nodes of the origin city from a start point and leave all arrival nodes of the final city to reach an end point.

4.3 Travel data adaptation

Figure 14 shows how our data is adapted to the space-time network described in Section 4.2. It simulates a travel to London (LON) and Paris (PAR) starting from Rio de Janeiro (RIO) in which a traveler must stay 1 to 3 days in each city ($Tmin_{LON} = Tmin_{PAR} = 1$ and $maxExtraTime_{LON} = maxExtraTime_{PAR} = 2$). Each node's label contains an airport code and the date/time of departure or arrival. For better differentiating nodes, we added some marks at the beginning of the airports' labels: `"o"` and `"f"` marks distinguish origin and final airports of the starting

city. The "w" mark identifies arrival airports in a city whereas airports codes alone represent the departure airports. Any travel begins in the dummy node "START" and selects one of the possible paths passing through RIO, LON, PAR and RIO again to finish in the dummy node "END". A possible path, for example, would pass through the nodes 1, 2, 3, 4, 5 and 6. Sometimes, however, arrival and departure nodes of a destination city are not valid candidates to integrate an accommodation arc due to the previously mentioned restrictions. For instance, a path starting in node [RIO, oGIG, 06/12/15 20:00] would never be able to complete the travel plan once its arrival node [LON, wLGW, 07/12/15 07:00] does not have any valid accommodation option. In fact, a model will be unfeasible when this kind of event is consistently replicated throughout the cities, being impossible to generate and adequate travel itinerary.

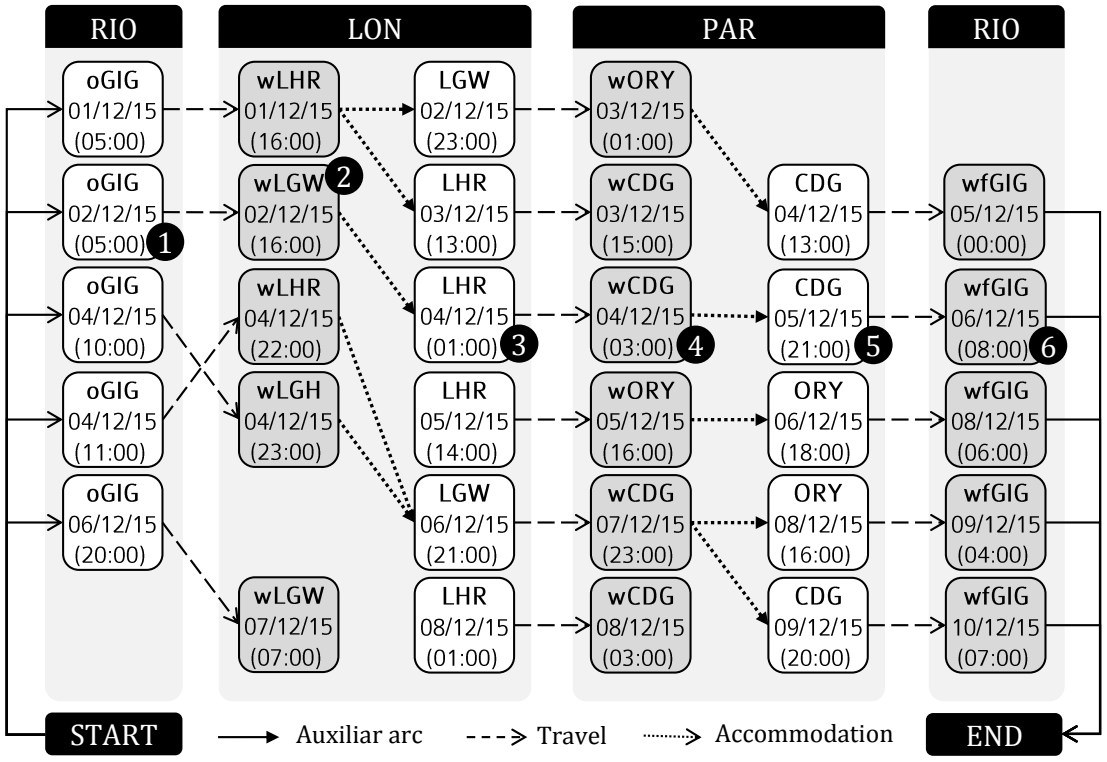

Figure 14 – Example of a space-time network for the route (RIO, LON, PAR, RIO). Arcs features are suppressed in favor of picture's visibility.

To further illustrate our travel data adaptation we also provide the content of the sets defined in Section 4.2. First, we have the unit sets I and F containing the origin city $I = F =$ {RIO} and the set C of destination cities, $C =$ {LON,PAR}. Next, we have the ordered sequence of cities $S =$ (RIO, LON, PAR, RIO) and the sets $P_2 =$ {(RIO,LON), (LON,PAR), (PAR,RIO)} and $P_3 =$ {(RIO, LON, PAR), (LON, PAR, RIO)}. Finally, we define the sets of existing flights between all pairs of cities in P_2. All sets below are subsets of $V_{flights}$:

From the set of travels $V_{flights}^{RIO,LON}$, $V_{flights}^{LON,PAR}$ and $V_{flights}^{PAR,RIO}$ we define the set of of vertices V and the sets of arcs $A_{flights}$ and A_{stays}:

$V_{flights}^{RIO,LON}$ = {
(RIO, LON, oGIG, wLHR, 01/12/15 05:00, 01/12/15 16:00), (RIO, LON, oGIG, wLGW, 02/12/15 05:00, 02/12/15 16:00),
(RIO, LON, oGIG, wLGH, 04/12/15 10:00, 04/12/15 23:00), (RIO, LON, oGIG, wLHR, 04/12/15 11:00, 04/12/15 22:00),
(RIO, LON, oGIG, wLGW, 06/12/15 20:00, 07/12/15 07:00) }

$V_{flights}^{LON,PAR}$ = {
(LON, PAR, LGW, wORY, 02/12/15 23:00, 03/12/15 01:00), (LON, PAR, LHR, wCDG, 03/12/15 13:00, 03/12/15 15:00),
(LON, PAR, LHR, wCDG, 04/12/15 01:00, 04/12/15 03:00), (LON, PAR, LHR, wORY, 05/12/15 14:00, 05/12/15 16:00),
(LON, PAR, LGW, wCDG, 06/12/15 21:00, 07/12/15 23:00), (LON, PAR, LHR, wCDG, 08/12/15 01:00, 08/12/15 03:00)}

$V_{flights}^{PAR,RIO}$ = {
(PAR, RIO, CDG, wfGIG, 04/12/15 13:00, 05/12/15 00:00), (PAR, RIO, CDG, wfGIG, 05/12/15 21:00, 06/12/15 08:00),
(PAR, RIO, ORY, wfGIG, 06/12/15 18:00, 08/12/15 06:00), (PAR, RIO, ORY, wfGIG, 08/12/15 16:00, 09/12/15 04:00),
(PAR, RIO, CDG, wfGIG, 09/12/15 20:00, 10/12/15 07:00)}

V = {
[RIO, oGIG, 01/12/15 05:00], [RIO, oGIG, 02/12/15 05:00], [RIO, oGIG, 04/12/15 10:00], [RIO, oGIG, 04/12/15 11:00],
[RIO, oGIG, 06/12/15 20:00], [LON, wLHR, 01/12/15 16:00], [LON, wLGW, 02/12/15 16:00], [LON, wLHR, 04/12/15 22:00],
[LON, wLGH, 04/12/15 23:00], [LON, wLGW, 07/12/15 07:00], [LON, LGW, 02/12/15 23:00], [LON, LHR, 03/12/15 13:00],
[LON, LHR, 04/12/15 01:00], [LON, LHR, 05/12/15 14:00], [LON, LGW, 06/12/15 21:00], [LON, LHR, 08/12/15 01:00],
[PAR, wORY, 03/12/15 01:00], [PAR, wCDG, 03/12/15 15:00], [PAR, wCDG, 04/12/15 03:00], [PAR, wORY, 05/12/15 16:00],
[PAR, wCDG, 07/12/15 23:00], [PAR, wCDG, 08/12/15 03:00], [PAR, CDG, 04/12/15 13:00], [PAR, CDG, 05/12/15 21:00],
[PAR, ORY, 06/12/15 18:00], [PAR, ORY, 08/12/15 16:00], [PAR, CDG, 09/12/15 20:00], [RIO,wfGIG, 05/12/15 00:00],
[RIO,wfGIG, 06/12/15 08:00], [RIO,wfGIG, 08/12/15 06:00], [RIO,wfGIG, 09/12/15 04:00], [RIO,wfGIG, 10/12/15 07:00] }

$A_{flights}$ = {
([RIO, oGIG, 01/12/15 05:00], [LON, wLHR, 01/12/15 16:00]), ([RIO, oGIG, 02/12/15 05:00], [LON, wLGW, 02/12/15 16:00]),
([RIO, oGIG, 04/12/15 10:00], [LON, wLGH, 04/12/15 23:00]), ([RIO, oGIG, 04/12/15 11:00], [LON, wLHR, 04/12/15 22:00]),
([RIO, oGIG, 06/12/15 20:00], [LON, wLGW, 07/12/15 07:00]), ([LON, LGW, 02/12/15 23:00], [PAR, wORY, 03/12/15 01:00]),
([LON, LHR, 03/12/15 13:00], [PAR, wCDG, 03/12/15 15:00]), ([LON, LHR, 04/12/15 01:00], [PAR, wCDG, 04/12/15 03:00]),
([LON, LHR, 05/12/15 14:00], [PAR, wORY, 05/12/15 16:00]), ([LON, LGW, 06/12/15 21:00], [PAR, wCDG, 07/12/15 23:00]),
([LON, LHR, 08/12/15 01:00], [PAR, wCDG, 08/12/15 03:00]), ([PAR, CDG, 04/12/15 13:00], [RIO,wfGIG, 05/12/15 00:00]),
([PAR, CDG, 05/12/15 21:00], [RIO,wfGIG, 06/12/15 08:00]), ([PAR, ORY, 06/12/15 18:00], [RIO,wfGIG, 08/12/15 06:00]),
([PAR, ORY, 08/12/15 16:00], [RIO,wfGIG, 09/12/15 04:00]), ([PAR, CDG, 09/12/15 20:00], [RIO,wfGIG, 10/12/15 07:00])}

A_{stays} = {
([LON, wLHR, 01/12/15 16:00], [LON, LGW, 02/12/15 23:00]), ([LON, wLHR, 01/12/15 16:00], [LON, LHR, 03/12/15 13:00]),
([LON, wLGW, 02/12/15 16:00], [LON, LHR, 04/12/15 01:00]), ([LON, wLHR, 04/12/15 22:00], [LON,LGW, 06/12/15 21:00]),
([LON, wLGH, 04/12/15 23:00], [LON, LGW, 06/12/15 21:00]), ([PAR, wORY, 03/12/15 01:00], [PAR, CDG, 04/12/15 13:00]),
([PAR, wCDG, 04/12/15 03:00], [PAR, CDG, 05/12/15 21:00]), ([PAR, wORY, 05/12/15 16:00], [PAR, ORY, 06/12/15 18:00]),
([PAR, wCDG, 07/12/15 23:00], [PAR, ORY, 08/12/15 16:00]), ([PAR, wCDG, 07/12/15 23:00], [PAR, CDG, 09/12/15 20:00])}

4.4 Network parallel building

The graph building is one of the most performed procedures of the method. In fact, every route evaluation demands its own travel graph building. Fortunately, some parts of the graph construction, such as, assessing and calculating arc values, may be parallelized. Therefore, to streamline the travel graph building, we applied a parallel process. Let $S = (c_{start}, c_1, ..., c_i, ..., c_n, c_{end})$ be the route that we are about to evaluate and P_2 the set of possible origin/destinations pairs in S as explained in Section 4.2. Starting from a void travel graph G, every pair $(S_i, S_{i+1}) \in P_2$ is processed in a different thread that performs the following operations:

- Add to G all travel arcs between the dummy node `"START"` and the departure vertices of S_i when $S_i = c_{start}$;
- Add to G all travel arcs between the arrival nodes of S_i and the dummy node `"END"` when $S_i = c_{end}$;
- Add to G all departure/arrival nodes of cities S_i and S_{i+1} and their correspondent travel arcs;
- Add to G all dwelling arcs between cities S_{i+1} and S_{i+2};

Figure 15 shows how this strategy can be applied to the example network depicted in Figure 14. The labeled boxes encompassing groups of vertices and arcs represent the parcel of work performed by each thread. The performance improvement produced by this building strategy is shown in detail in Section 5.4.2.

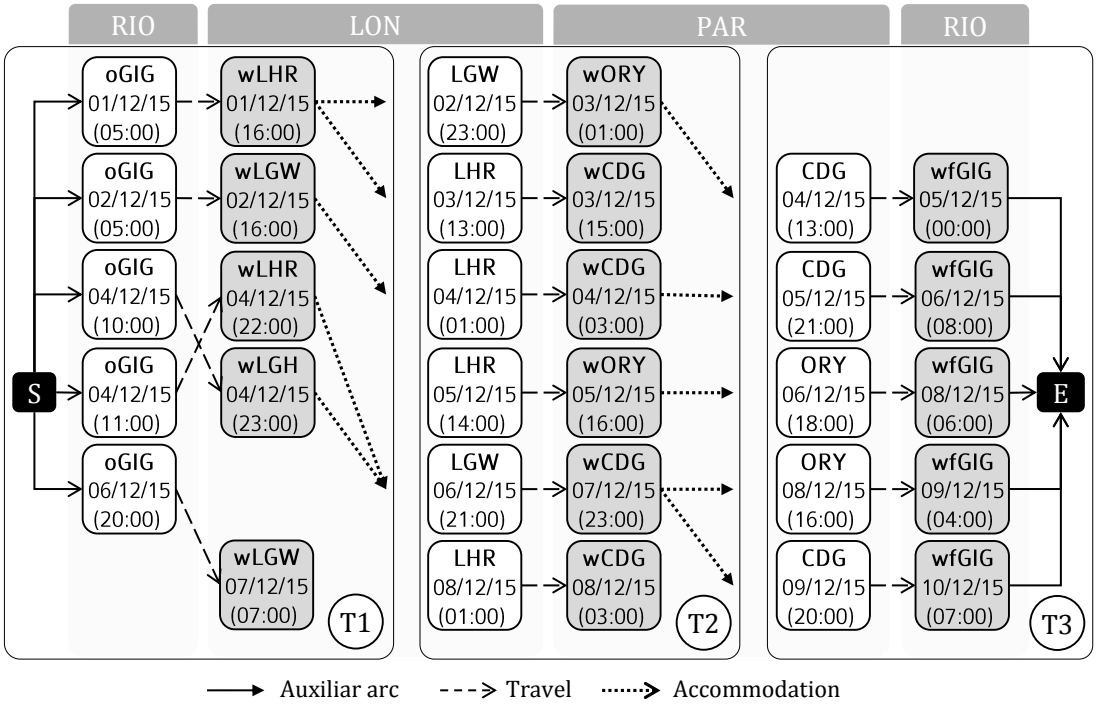

Figure 15 – Parallel building process applied to the example depicted in Figure 14. Vertices and arcs are created in parallel by threads T1, T2 and T3. The vertices labeled with `"S"` and `"E"` correspond to the dummy nodes `"START"` and `"END"` respectively.

4.5 Finding the best itinerary of a travel graph

Any sequence of visits generates a unique travel graph comprised of dummy nodes, travel vertexes, flight and accommodation arcs. The shortest path that connects the two dummy nodes of this graph corresponds to the optimal solution for the travel planning problem on a fixed sequence of cities. This optimal solution can be generated in regard to any arc's features, such as cost or time. Thereafter, in this study the feature set to assign values to arcs depends on the approach being applied: single-objective or bi-objective. However, regardless the approach, we apply the classic Dijkstra algorithm to determine the shortest path for the travel network generated from a route. As pointed out in (CORMEN, 2009), a Dijkstra procedure requires a directed graph containing a set V of n vertices and a set E of m directed edges with nonnegative weights and also a source vertex in V. Our graph model adapted from (BÉRUBÉ; POTVIN; VAUCHER, 2006) fulfils all these requirements once all flight and dwelling arcs are oriented and hold positive values, and there is

a dummy source from where all possible paths must start. Figure 16 and Figure 17 show different states of the application of this algorithm on our travel network. From the state depicted in Figure 16, the algorithm iteratively explores the graph, setting the minimum path to reach each node until all nodes are visited. The symbols $f_1, ..., f_{16}$ and $s_1, ..., s_{10}$ represent the features related to flight and stay arcs respectively. Therefore, f symbols can assume flight fares or travel times whereas s symbols can assume accommodation costs or waiting times.

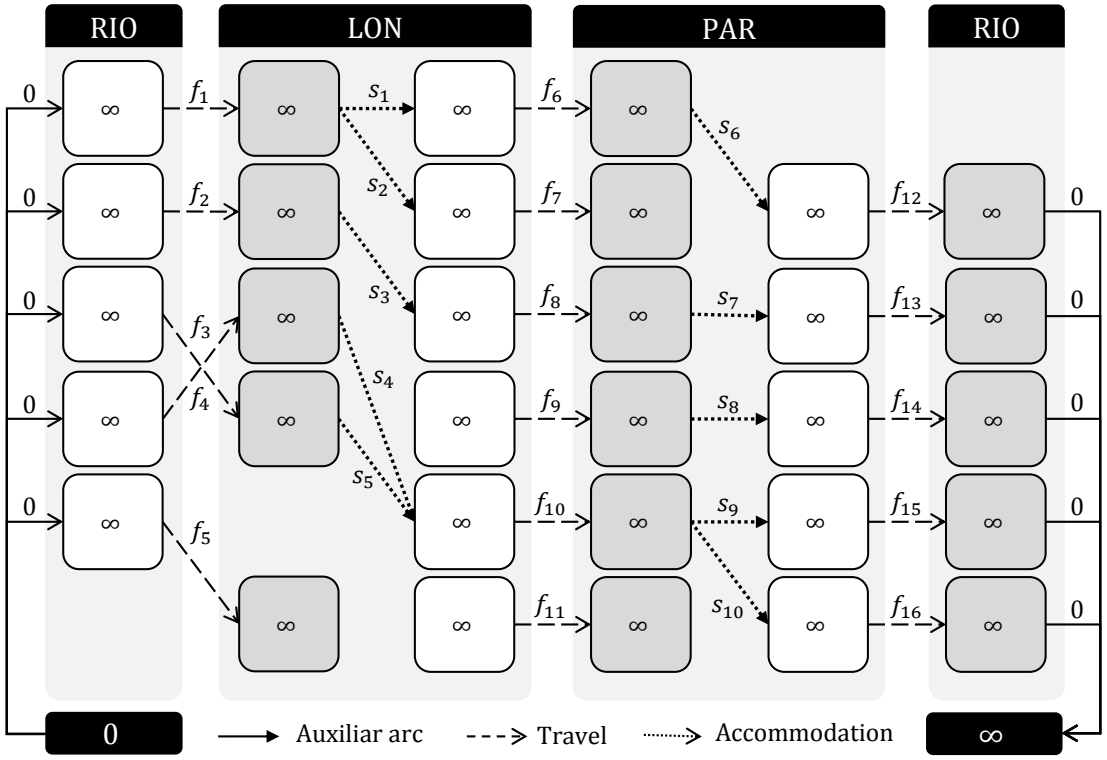

Figure 16 – First step of the Dijkstra algorithm on the space-time network of example 14.

Initially, regardless the feature chosen, the minimum cost/time to reach any node is set to infinity. As Figure 17 shows a state before the end of the algorithm once the value to reach the last node (`END`) was not yet set. The cost to reach every node is the summation of all previous costs in the path. The value of the end node will be the lowest among all the arrival nodes in RIO. In order to keep track of the path, every time the minimum path to reach a node is updated we also update a variable that stores the previous visited node. At the end of the shortest path algorithm, we are able to get the complete path by iteratively checking the previous nodes of each element starting from the `END` node.

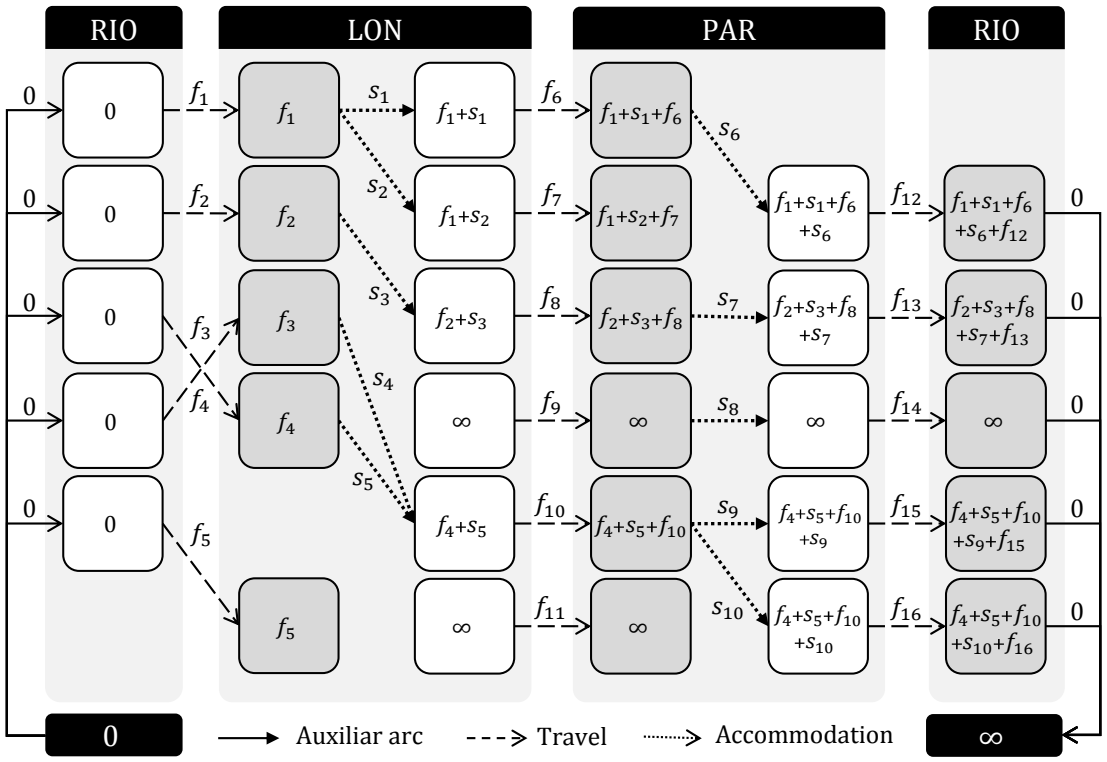

Figure 17 – Second to last step of the Dijkstra algorithm on the space-time network of example 14.

4.6 Solution outline

The shortest path generated through the application of the Dijkstra algorithm on a travel network contains enough information to present a solution for a given route. In Figure 18 we show how the details of this path could be portrayed to stress the most important features of a journey. The itinerary was produced based on the following input data: $I = F = \{RIO\}$, $C = \{PAR, LON\}$, $Tmin_{PAR} = Tmin_{LON} =$ 3 days, $d_{start} =$ 2015/12/01 and $d_{end} =$ 2016/01/10. Instead of textual labels, we decided to present the information using a set of symbols in order to ease the understanding. Firstly, regarding the travel data, the dollar sign represents cost, the hourglass represents waiting, and the clock symbolizes total duration. The first line summarizes the features of the journey: the tourist would wait for 43h25 and pay \$1,867.19, including flights and stays, to spend at least 3 days in each destination city. The waiting time is the summation of all the journey's waitings whether they are transportation times or waiting times to get a bed/board transportation. Additionally, the summarized solution also presents the total length of the journey, which is 17 days and 15 hours long, from 2015/12/26 to 2016/01/05. The next lines present all cities involved in the journey as well as all flights between these cities. The first and last cities are marked with a "home" symbol and do not contain any further information besides their name and code. In turn, the destination cities, are marked with a "building" symbol to suggest a third party accommodation. Every

accommodation line contains the total cost and duration of the stay and also the waiting time that a tourist must endure in the city without access to a room. Finally, the travel lines are represented by the airplane symbol and show the flights' data between two consecutive cities of the route. This data include departure/arrival airports, travel time and flight fare.

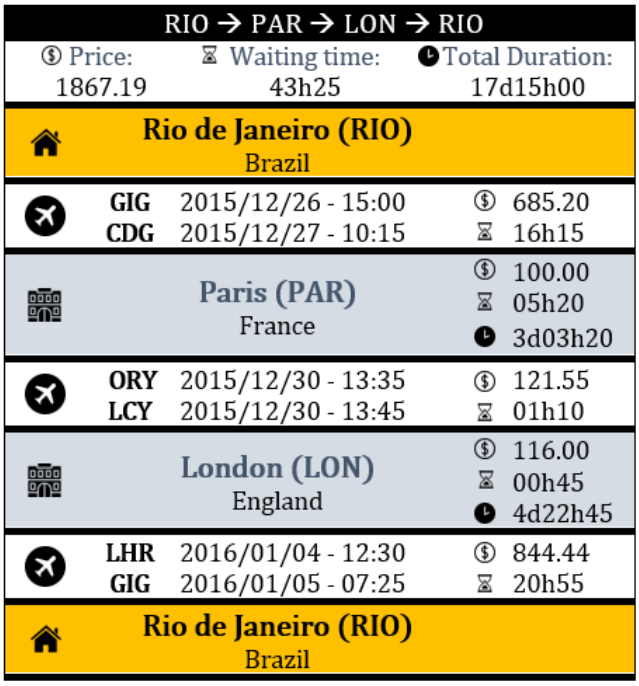

Figure 18 – Solution outline of the route {Rio de Janeiro, Paris, London, Rio de Janeiro}.

5 Single-objective approach

This section compiles all strategies that we implemented to solve the single-objective travel planning problem. This approach considers that a tourist is only interested in determining what is the travel plan with the lowest price. Firstly, in Section 5.1 we propose an integer programming formulation based on the input data defined in Section 4.1. Although not implemented, this formulation consists on a theoretical effort to formalize the problem. Next, in Section 5.2, we present two solving strategies to deal with this particular version of the problem. The first is a parallel brute-force algorithm (Section 5.2.1) and the second is an ILS algorithm (Section 5.2.2). Then, in Section 5.3, we show how our test cases were set to assess the quality of the results of our heuristic implementation when compared with the optimal solutions reached by the brute-force method. Finally, in Section 5.4, we present the results of our test cases and discuss the limitations of our approaches.

5.1 Integer programming formulation

Our integer programming formulation for the single-objective travel planning problem relies on a binary decision variable $X_{i_s j_e}^{t_1 t_2}$. It assumes the value 1 only when the final itinerary comprises the flight represented by the tuple $(s, e, i_s, j_e, t_1, t_2) \in V_{flights}$ or the stay represented by the tuple $(c, i_c, j_c, t_1, t_2) \in V_{stays}$ (considering $c = s = e$). Otherwise, the variable is zeroed.

Given the data definition in Section 4.1 and the decision variable, our travel planning problem is stated as follows:

$$\min Z = \sum_{\forall(s,e,i_s,j_e,t_1,t_2)\in V_{flights}} P_{i_s j_e}^{t_1 t_2} X_{i_s j_e}^{t_1 t_2} + \sum_{\forall(c,j_c,i_c,t_1,t_2)\in V_{stays}} S_{j_c i_c}^{t_1 t_2} X_{j_c i_c}^{t_1 t_2} \tag{5.1}$$

subject to:

$$\sum_{j_e \in A_e} \sum_{\substack{s \in (I \cup C) \\ s \neq e}} \sum_{i_s \in D_s} \sum_{t_1 \in T_{i_s j_e}} \sum_{t_2 \in T_{i_s j_e}^{t_1}} X_{i_s j_e}^{t_1 t_2} = 1 \quad \forall e \in (C \cup F) \tag{5.2}$$

$$\sum_{i_s \in D_s} \sum_{\substack{e \in (C \cup F) \\ e \neq s}} \sum_{j_e \in A_e} \sum_{t_1 \in T_{i_s j_e}} \sum_{t_2 \in T_{i_s j_e}^{t_1}} X_{i_s j_e}^{t_1 t_2} = 1 \quad \forall s \in (I \cup C) \tag{5.3}$$

$$\sum_{\forall(c,j_c,i_c,t_1,t_2)\in V_{stays}} X_{j_c i_c}^{t_1 t_2} = 1, \forall c \in C \tag{5.4}$$

$$\sum_{\substack{s\in(I\cup C)\\ s\neq c}} \sum_{i_s\in D_s} \sum_{t_1\in T_{i_s j_e}} X_{i_s j_c}^{t_1 t_2}[(s,c,i_s,j_c,t_1,t_2)\in V_{flights}] = \sum_{i_c\in D_c} \sum_{t_3\in DP_{i_c}} X_{j_c i_c}^{t_2 t_3}[(c,j_c,i_c,t_2,t_3)\ \in\ V_{stays}], \tag{5.5}$$

$$\forall\ \ c\in C$$

$$\forall\ \ j_c\in A_c$$

$$\forall\ \ t2\in AR_{j_c}$$

$$\sum_{j_c\in A_c} \sum_{t_2\in AR_{j_c}} X_{j_c i_c}^{t_2 t_3}[(c,j_c,i_c,t_2,t_3)\ \in\ V_{stays}] = \sum_{\substack{e\in(C\cup F)\\ e\neq c}} \sum_{j_e\in A_e} \sum_{t_4\in T_{icje}^{t3}} X_{i_c j_e}^{t_3 t_4}[(c,e,i_c,j_e,t_3,t_4)\in V_{flights}] \tag{5.6}$$

$$\forall\ \ c\in C$$

$$\forall\ \ i_c\in D_c$$

$$\forall\ \ t_3\in DP_{i_c}$$

$$X_{i_s j_e}^{t_1 t_2}\in\{0,1\},\quad \forall(s,e,i_s,j_e,t_1,t_2)\in V_{flights} \tag{5.7}$$

$$X_{j_c i_c}^{t_1 t_2}\in\{0,1\},\quad \forall(c,j_c,i_c,t_1,t_2)\in V_{stays} \tag{5.8}$$

The goal of the objective function (5.1) is to minimize the total costs of a travel schedule that encompasses flights and stays. The remaining elements guarantee that the schedule abides by the inherited constraints of a touristic travel. Firstly, equations (5.2) and (5.3) establishes that there is a single travel arriving in a city in $(C \cup F)$ and also a single travel departing from a city in $(I \cup C)$. Hence, any route must start from a city in I and finish in a city in F, passing only once through all destination cities in C. In other words, a tourist cannot spend the predefined minimal time in each city throughout many short visits. Additionally, equation (5.4) guarantees that there is a single stay in each city in C. Next, equations (5.5) and (5.6) establishes that there must be only one dwelling arc that connects the arriving node into the departure node. Finally, we state the binary nature of our decision variable when it denotes a flight occurrence (5.7) and a stay occurrence (5.8). Notice we do not create a stay length constraint since it is already inherited from V_{stays} set.

Figure 19 shows how an itinerary can be graphically represented considering that: $I = F = \{\texttt{RIO}\}$, $D_{RIO} = A_{RIO} = \{\texttt{GIG, SDU}\}$, $C = \{\texttt{LON,PAR}\}$, $D_{LON} = A_{LON} = \{\texttt{LGW, LHR}\}$, $D_{PAR} = A_{PAR} = \{\texttt{ORY, CDG}\}$, $d_{start} = 0$ time units, $d_{end} = 20$ time units, $Tmin_{LON} = Tmin_{PAR} = 3$ time units. The following decision variables are assigned *true* to create this itinerary: $X_{RIO_{GIG}LON_{LGW}}^{2,5}$, $X_{LON_{LGW}LON_{LGW}}^{5,10}$, $X_{LON_{LGW}PAR_{CDG}}^{10,11}$, $X_{PAR_{CDG}PAR_{ORY}}^{11,15}$, and $X_{PAR_{ORY}RIO_{SDU}}^{15,17}$.

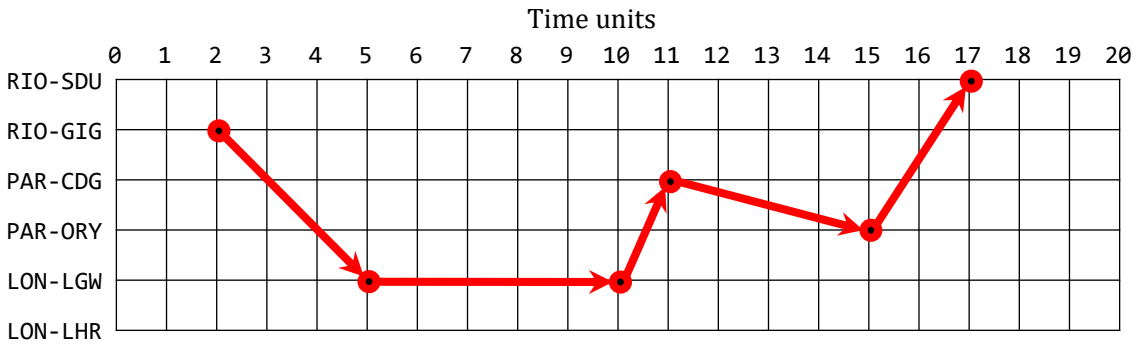

Figure 19 – Example of a graphical representation of a viable itinerary for an instance of the travel planning problem using a short time window of 21 time units (from 0 to 20).

5.2 Solving strategies

The following sections present the two solving strategies we applied to solve the single-objective version of the problem: a brute-force algorithm (Section 5.2.1) and the iterated local search heuristic (Section 5.2.2). The objective function of both strategies differ slightly from the objective function presented in our integer programming formulation. Since our single objective implementation accounts only for the travel total cost, we decided to set a penalty of $1 monetary unit per waiting time hour in an accommodation. This penalization policy is an effort to not completely discard the time component from the optimization process. In fact, this modification may improve the quality of the solutions once it may favor schedules with lower waiting times over schedules with similar overall costs but with longer waiting periods. Evidently, this penalty is not applied to our bi-objective implementation once it is already taking in consideration total waiting times.

5.2.1 Parallel single-objective brute-force algorithm

As a comparative benchmark, we implemented a brute-force method to find the optimal solution for some of our simpler test cases. Firstly, considering a touristic travel through n cities, we generate all $n!$ possible routes. Then, for each route, we build its correspondent travel graph and check the cost of traversing this graph applying the shortest path algorithm. To streamline the process, we parallelized the building and checking process so that 24 routes are checked at a time. This number is twice bigger the available number of threads of our processor (check hardware configurations in Section 5.3.1) to guarantee that its cores are as busy as possible. Indeed, the calculation of the shortest path of any route involves memory bounded operations such as reading travel data and writing graph components so that the

CPU could be idle at some moments. In Section 5.4.3 we present how large are the test cases tractable by this algorithm.

5.2.2 ILS (Iterated Local Search)

The objective of ILS is to improve upon stochastic Multi-Restart Search by sampling in the broader neighborhood of candidate solutions and using a Local Search technique to refine solutions to their local optima (BROWNLEE, 2011). Hence, ILS explores a sequence of solutions created as perturbations of the current best solution, the result of which is refined using an embedded heuristic. In our approach, a Double Bridge movement performs these perturbations and the local search refinement is performed by a complete 2-Opt search. Algorithm 1 shows how we implement the ILS. We first generate an initial route using a greedy method and then we repeat the perturbation/refinement process as many times as the number specified in the parameter *maxNoImproveIt*. If any improvement in route cost is found, we update the current best solution and restart the counting of iterations so that another round of at least *maxNoImproveIt* iterations will take place. Therefore, our method is tailored to analyze several candidate solutions until no remarkable improvement is made. The following sections describe the details of each component procedure of ILS.

Algorithm 1 Iterated Local Search

```
1:  function ILS(maxNoImproveIt)
2:      it ← 0
3:      initial ←greedy()
4:      current ←2-Opt(initial)
5:      repeat
6:          it ← it + 1
7:          perturbated ← doubleBridge(current)
8:          perturbatedLSearch ← 2-Opt(perturbated)
9:          if getCost(perturbatedLSearch) < getCost(current) then
10:             current ← perturbatedLSearch
11:             it ← 0
12:     until it ≤ maxNoImproveIt
13:     return current
```

5.2.2.1 Greedy initial solution

According to Cormen (2009) a greedy algorithm always makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Although this strategy does not always yield an optimal result due to its short-sight nature, it is able to rapidly produce a solution that may be a promising starting point for a more complex heuristic. Our ILS initial solution is generated by the *greedy*() procedure (line 3 of Algorithm 1) which implements the following greedy algorithm:

from the start city, we select the next city whose addition to route will cause the smallest increment in the travel's total cost. From each new added city, the process is repeated until there are no more cities to add.

5.2.2.2 2-Opt local search

A local search consists in a general applicable approximation technique for combinatorial optimization problems that presupposes a neighborhood structure (VERHOEVEN; AARTS; SWINKELS, 1995). In a broader sense, we are dealing with the TSP, subsequently, a solution for our problem consists fundamentally in a sequence of visits. Therefore, any modification in this sequence produces a neighbor solution. The 2-Opt() method of Algorithm 1 implements a complete 2-Opt local search in a route, i.e., it applies all available 2-Opt movements. As described in (LENSTRA, 1997), a 2-Opt movement swaps two elements of a route mirroring all elements within, so that, after the changes, a route that once crossed over itself, will not. For an n-sized route there are $m = (n * (n-1))/2$ possible 2-Opt movements. Thus, our local search executes m 2-Opt movements and chooses the one that produces the lowest cost route. An example of the application of a 2-Opt movement in a sequence of 6 destination cities is depicted in Figure 20.

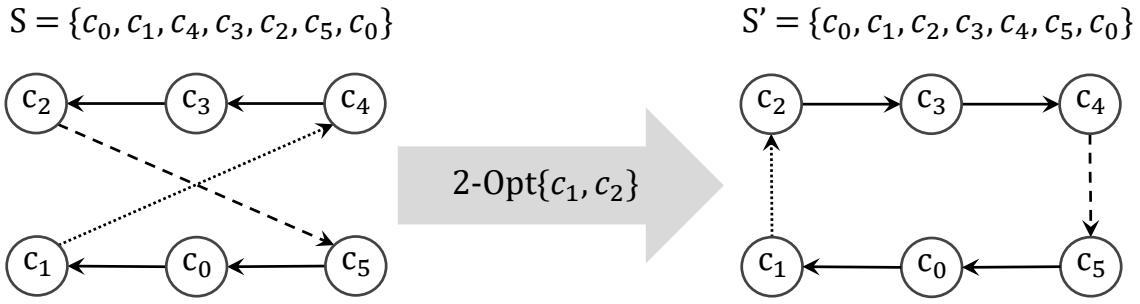

Figure 20 – Application of the 2-Opt movement $\{c_1, c_2\}$ in a sequence of visits S.

In order to decrease the execution time of the 2-Opt process, we implemented a second version of this algorithm applying function parallelism, i.e., executing each step of the sequential local search algorithm in parallel (VERHOEVEN; AARTS; SWINKELS, 1995). Therefore, each possible movement in the 2-Opt local search is performed by a separated thread. A route is calculated and the best travel route is updated every time a new promising solution is found. Figure 21 shows how this parallel process is applied to initial route S with an estimated cost of \$350. Once the start and end cities are fixed, the 2-Opt method has 5 cities to work with. Hence, there are 10 possible 2-Opt movements that are processed individually.

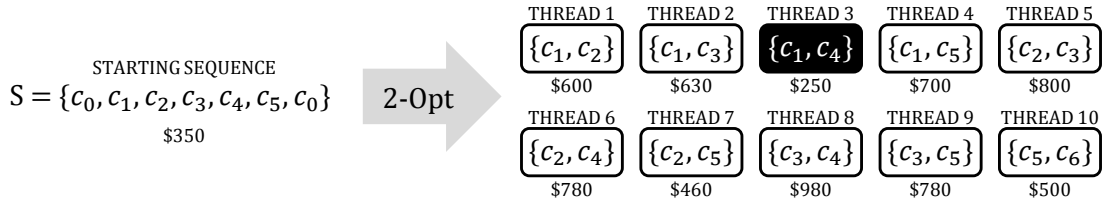

Figure 21 – Example of a parallel 2-Opt local search applied to a sequence S. Among all possible movements performed by the threads from 1 to 10, the $\{c_1, c_4\}$ movement of thread 3 is the best neighbor of S once it produces the lowest cost route.

5.2.2.3 Memory set

Any route generated either by 2-Opt local search or Double Bridge perturbation methods is saved in a set, so that every time a new route is to be calculated, i.e., have its travel graph generated and shortest path determined, we first check if the route is already in the memory set. If so, the route was previously calculated and either was discarded by the optimization process or is the current best available route. In both cases, we save some computing time since we do not need to recalculate candidate solutions already discovered in previous iterations.

5.2.2.4 Double Bridge perturbation

A Double Bridge (DB) consists in a 4-Opt movement (LENSTRA, 1997). It is usually used as a perturbation to escape from a local optimum once its outcome is hardly reproduced by a solely application of the 2-Opt. Algorithm 2 shows how we adapted the procedure proposed in (BROWNLEE, 2011) in which a permutation is partitioned into 4 pieces (a,b,c,d) and is put back together in a specific and jumbled ordering (a,d,c,b). Differently from the author's original method in which positions p_1, p_2 and p_3 of movement m are generated at random inside the function, in our implementation the movement is received as an argument together with the permutation that will be perturbed.

Algorithm 2 Apply DB-movement m to permutation $perm$

```
1: function applyDBMovement(perm, m)
2:     part1 ← perm[0...m.p1] + perm[m.p3...perm.size]
3:     part2 ← perm[m.p2...m.p3] + perm[m.p1...m.p2]
4:     return part1 + part2
```

Figure 22 shows an example of a Double Bridge movement applied to the sequence $S = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$. This movement is represented by a sequence of cities $\{c_1, c_3, c_5\}$ whose departing arcs are removed to be further relinked in different aways. Notice that c_7 is not included in the sequence since Algorithm 2 always select the last arc to be removed.

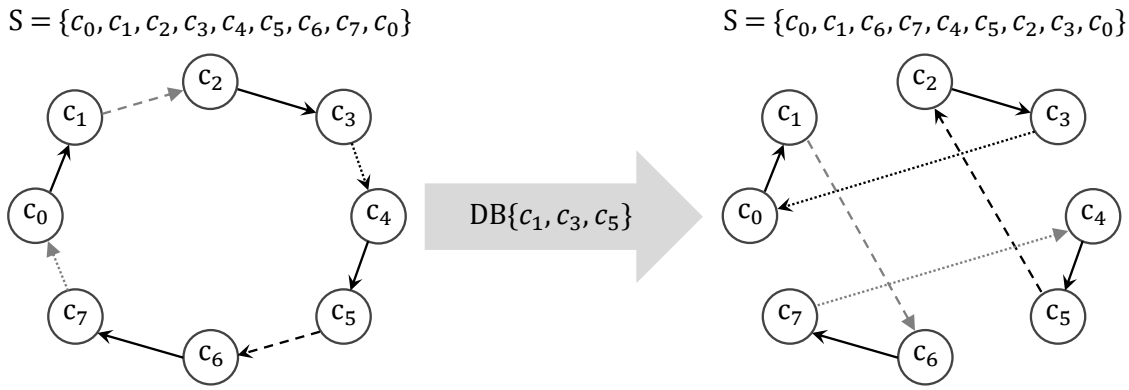

Figure 22 – Example of a double bridge movement applied to a travel route with 7 destination cities. The partitions a = {c_0,c_1}, b = {c_2,c_3}, c = {c_4,c_5} and d = {c_6,c_7} are reconnected in the order a, d, c, b.

Algorithm 3 Double-bridge

```
1:  function doubleBridge(perm)
2:      dbTested ← ∅
3:      repeat
4:          if dbTested.size = listDB.size then
5:              break
6:          repeat
7:              m ← dbM[rand(dbM.size)]
8:          until !dbTested.add(m)
9:          dbPerm ← applyDBMovement(perm, m)
10:     until !memorySet.add(dbPerm)
```

Prior to the method's execution, we generate the list of all possible DB-movements for a given route size. Then, during the application of the DB method to a route, we randomly apply the DB-movements from this list until we found a route that is not in the memory set (Section 5.2.2.3), i.e., a candidate route that was not yet discovered. Table 3, for example, shows the available movements when a 7-destinations route like the one portrayed in Figure 22 is concerned.

Table 3 – The 35 available DB-movements for a 7-destinations route. City c_0 is the origin city.

c_0, c_1, c_2	c_0, c_2, c_3	c_0, c_3, c_5	c_1, c_2, c_3	c_1, c_3, c_5	c_2, c_3, c_4	c_2, c_5, c_6
c_0, c_1, c_3	c_0, c_2, c_4	c_0, c_3, c_6	c_1, c_2, c_4	c_1, c_3, c_6	c_2, c_3, c_5	c_3, c_4, c_5
c_0, c_1, c_4	c_0, c_2, c_5	c_0, c_4, c_5	c_1, c_2, c_5	c_1, c_4, c_5	c_2, c_3, c_6	c_3, c_4, c_6
c_0, c_1, c_5	c_0, c_2, c_6	c_0, c_4, c_6	c_1, c_2, c_6	c_1, c_4, c_6	c_2, c_4, c_5	c_3, c_5, c_6
c_0, c_1, c_6	c_0, c_3, c_4	c_0, c_5, c_6	c_1, c_3, c_4	c_1, c_5, c_6	c_2, c_4, c_6	c_4, c_5, c_6

In spite of being chosen at random, we guarantee that a movement is not applied more than once into a route. Algorithm 3 shows how this process is accomplished:

we use a set called $dbTested$ that stores all DB-movements already tested so that the process of finding a new movement finishes only when an original movement is found ($dbTested.add(m)$ returns true). In turn, the complete DB method only finishes if a totally original route is found ($memorySet.add(dbPerm)$ returns true) or all available movements in $listDB$ were already tested ($dbTested.size = dbM.size$).

Additionally to exploring only a single perturbation per route, we also implemented a parallel version of the double bridge in which n perturbations are performed at once. In Figure 23 we present a flowchart that illustrates how this parallel strategy is integrated in the ILS method stated in Algorithm 1. Firstly, the greedy method produces an initial solution S that is subsequently submitted to a parallel 2-Opt local search method that returns a solution S_B corresponding to the best neighbor of S. Then, S_B undergoes to n different perturbations stored in the memory set dbM corresponding to the random positions $p_1, p_2, ..., p_n$. Next, all perturbed solutions $S_1, S_2, ..., S_n$ are individually optimized by 2-Opt local search methods and the best result among the double bridge threads, i.e., the solution that has the minimum objective value is the final outcome of the double bridge process S_{DB}. Finally, if the solution produced by the parallel double bridge process S_{DB} is better than the current best solution S_B, it becomes the new best solution and the process is repeated once again. Otherwise, the method is terminated since we have limited to 1 the maximum number of iterations with no improvement ($maxNoImproveIt$). This limitation avoids long execution times once it guarantees that the process is repeated only while there is room for improvement.

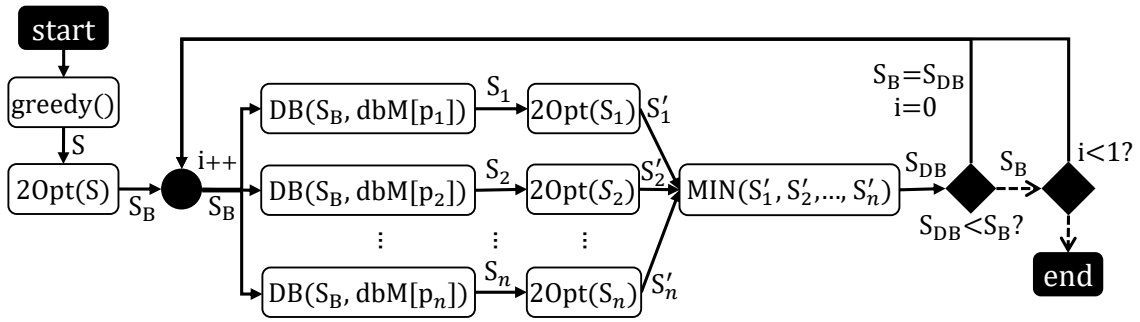

Figure 23 – Flowchart of a modified version of Algorithm 1 stressing the parallel implementation of the double bridge perturbation.

5.3 Simulation

In this section we present our testing environment (Section 5.3.1), the configuration of our instances (Section 5.3.2) and the performance metric used to access them (Section 5.3.3).

5.3.1 Settings of hardware and data

In order to restrict the size of the travel graph and improve performance, we limit the number of flights per flight connection to the top 25 cheapest flights. Therefore, considering the 9,840 flight connections, we have 246.000 (9.840x25) possible flights. All data were copied in advance to RAM memory so that accessing any information had minimal latency. The method was coded in Java and all tests were executed using one node of a computer cluster. This particular node had two 2.66 GHz Xeon processors (6 cores, 12 thread) and a reserved 7GB RAM space to run our application.

5.3.2 Instances

In this section we show how we set the values of the input data defined formally in Section 4.1.1 to produce our test cases. Based on the travel settings previously determined in Chapter 3, we set the fixed input data, i.e., the input data that will not change over the execution of all test cases. Therefore, for any instance tested in this study:

- The origin/final city is Rio de Janeiro, i.e., $I = F = \{\texttt{RIO}\}$;
- The travel time window is 40 days long, starting in $d_{start} = 2015/12/01$ and finishing in $d_{end} = 2016/01/10$;
- The maximum extra time that a tourist may spend in a city c is 2 days, i.e., $maxExtraTime_c = 2$ days;

Then, we need to define the set of destination cities C and also the minimum dwelling time $Tmin_c$ in each city $c \in C$. However, it is precisely the variation of these remaining data that creates our different test cases. In fact, one of the main goals of our instances is to assess the impact of this variation in the performance of our solving methods. Firstly, let $C_{all} = \{\texttt{LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER}\}$ be the set of the 15 destination cities that we were able to collect travel data (Section 3.1). If all possible subsets of C_{all} were considered to create our test cases, we would have at least 32.767 instances, i.e., the amount of elements of $\mathbb{P}(C_{all})$ without the empty set. This number is rather considerable since we still need to combine these elements with different dwelling times. Therefore, we chose a subset of $\mathbb{P}(C_{all})$ to be part of our test cases. We first combine the cities in C_{all} in 15 ordered lists with 14 elements each. As shown in Table 4, we did it by iteratively shifting left the first element of a list created from the collection C_{all}. Since the maximum length of each list is 14, a shifted element which is removed from the first position will eventually reappear in the last position, as if we were working on a circular list.

Table 4 – The 15 possible destination sets built over our list of cities.

#	Destination sets
1	LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH
2	PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER
3	FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON
4	MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR
5	IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA
6	AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW
7	MAD, ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW, IST
8	ROM, MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW, IST, AMS
9	MUC, MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW, IST, AMS, MAD
10	MIL, BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW, IST, AMS, MAD, ROM
11	BCN, AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC
12	AYT, BRU, ZRH, BER, LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL
13	BRU, ZRH, BER, LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN
14	ZRH, BER, LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT
15	BER, LON, PAR, FRA, MOW, IST, AMS, MAD, ROM, MUC, MIL, BCN, AYT, BRU

Then, for each possible destination list, we create 14 possible subsets with an increasing number of cities (from 1 to 14), starting from the first destination. For the first set of Table 4 for example, we would get a group of destination subsets $G_1 = \{$`{LON}`, `{LON, PAR}`, `{LON, PAR, FRA}`, `{LON, PAR, FRA, MOW}`, `{LON, PAR, FRA, MOW, IST}`, `{LON, PAR, FRA, MOW, IST, AMS}`, ..., `{LON, PAR, FRA, ..., ZRH}`$\}$. At the end of this process we are able to define a group $G_{CitySets}$ containing 210 (15×14) possible destination sets of variable sizes.

Regarding the dwelling time, we intend to assess the behaviour of our methods even when unusual inputs are chosen. We do not prevent, for example, the creation of travel plans in which a tourist does not have enough time to properly visit the cities. Therefore, we determined that the dwelling time $Tmin_c$ in a city c may range from 1 to 30 days. However, in order to produce more regular test cases, we considered that all cities c of a set $C \in G_{CitySets}$ are equally assigned to a unique dwelling time. Besides that, for maintaining the correctness of the instances, the total amount of time spent in a city must not surpass our travel time window of 41 days. We further decrease this threshold by 2 days, in order to guarantee that a tourist has at least this time span to spend waiting or inside transportation. Therefore, the sum of all $Tmin_c$ of the cities $c \in C$ must be lower than 42 days. Following these specifications, we created Table 5 with all available test cases in this study.

Table 5 – Relation of test cases generated for assessing the exact method. Every row ends with the subtotal of instances for each number of cities.

#Cities	Dwelling time (days)	Destination set	Subtotal
1	1 – 30	1 – 15	450
2	1 – 19	1 – 15	285
3	1 – 13	1 – 15	195
4	1 – 9	1 – 15	135
5	1 – 7	1 – 15	105
6	1 – 6	1 – 15	90
7	1 – 5	1 – 15	75
8	1 – 4	1 – 15	60
9	1 – 4	1 – 15	60
10	1 – 3	1 – 15	45
11	1 – 3	1 – 15	45
12	1 – 3	1 – 15	45
13	1 – 3	1 – 15	45
14	1 – 2	1 – 15	30
		Total number of test cases:	1665

5.3.3 Performance measure

The quality of our single-objective heuristic is comprised of two factors: total cost and execution time. Regarding the total cost, a solution s for a given test case t from Table 5 is as good as its deviation from the optimal solution approximates to zero. Once the best solution for the test case t is known through the application of the brute force method, we use a Relative Percentage Deviation (RPD) metric to determine how far our heuristic generated solutions are from reaching the optimal outcome. This metric is calculated as follows:

$$RPD_{\mathrm{s}} = \frac{f_{\mathrm{s}} - f_{\mathrm{best}}}{f_{\mathrm{best}}} \times 100\% \tag{5.9}$$

In turn, regarding the execution time, we measure how many seconds each test case take to execute in our testing environment, using the ILS algorithm described in Section 5.2.2. The brute force method is cast out of this comparison since it takes several minutes to reach a solution even for test cases with a moderate number of cities. The execution time metric is also used to measure the quality of our parallel algorithms, i.e., if they were able to produce any remarkable speedup.

5.4 Results and discussion

The following sections present the results we achieved after applying and executing the solving strategies of Section 5.2. Besides evaluating the impact of the variation of the number of cities and the dwelling time duration on the execution time, we also intend to assess the overall quality of our heuristic methods when compared to the exact approach.

5.4.1 Density of a travel network: edges & vertices

In this section we evaluate how the building elements of our travel network, i.e., edges and vertices, are affected by the number of cities, dwelling times and max. extra times. Firstly, in Figure 24 we show how the average amount of edges and vertices of a travel graph is affected by the number of cities. To create this graph, we run our graph building procedure in the different destinations subsets of $G_{CitySets}$ (Section 5.3.2) as if they were sequences, group the solutions by total number of cities and take the average number of edges and vertices for each group. To create the estimations of a 1-city travel for example, we would take the average number of edges and vertices of the routes in R ={ {RIO, LON, RIO}, {RIO, PAR, RIO}, ..., {RIO,BER,RIO}}.

As expected, the average amount of edges and vertices consistently grows as the number of destination cities increases. The most complex travel graph for example would involve 14 cities, almost 300,000 edges and 20,000 vertices. Concerning the number of vertices, each sequential city pair in a route has at most 2,050 vertices once there are 25 trip options per pair, 2 vertices per trip option (departure and arrival) and 41 available days. In addition to that, regardless the travel plan, there will be also at most 2,050 vertices (including departure and arrival) in the origin city, besides the ghost vertices START and END. Following these estimations, if we take for example a 1-city travel, it would produce a travel graph with 4,102 vertices (2,050 + 2,050 + 2). However, this number is about 2 times bigger than the number presented in Figure 24 which is about 2,000 vertices. Nevertheless, this result is completely coherent once not all trip options involve unique vertices. Take for example the vertices involved in the first 3 travel options from Table 2:

- oSDU|2015/12/01 06:25-02:00;wAMS|2015/12/02 10:05+01:00
- oSDU|2015/12/01 06:25-02:00;wAMS|2015/12/02 13:35+01:00
- oSDU|2015/12/01 06:25-02:00;wAMS|2015/12/02 17:45+01:00

Although the 3 options of this excerpt involve 2 vertices each, there are only 4 distinct vertices once the departure vertex oSDU|2015/12/01 06:25-02:00 is connected to 3 arrival vertices: wAMS|2015/12/02 10:05+01:00, wAMS|2015/12/02 13:35+01:00 and wAMS|2015/12/02 17:45+01:00. This situation is very common since many trips may either start from the same departure vertex and finish in different arrival vertices or start in different departure vertices and finish in the same arrival vertex due to, for example, differences on the carriers operating the flight or number of connections.

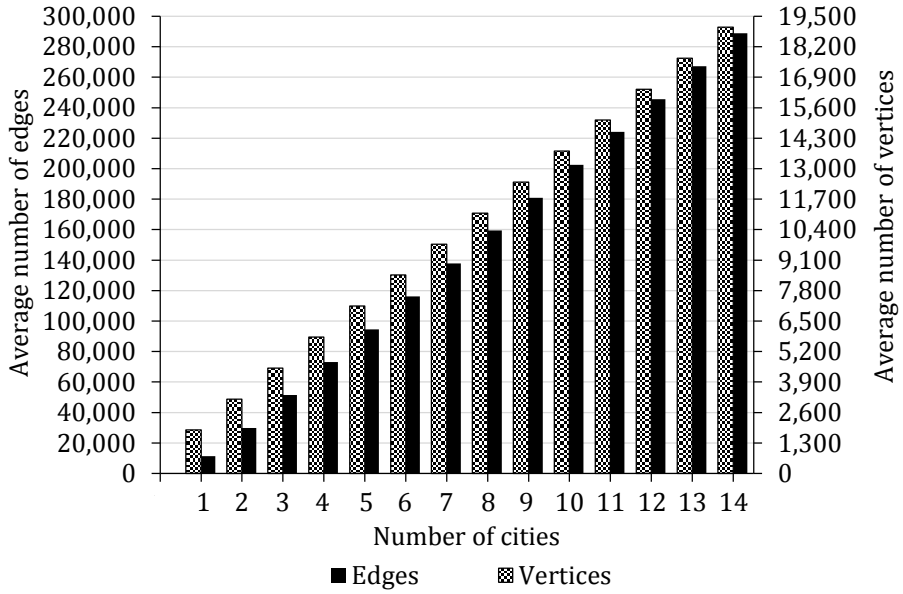

Figure 24 – Impact of the number of cities in the amount of edges and vertices considering a dwelling time of 1 day and a max extra time of 2 days. The number of edges surpass in many times the number of vertices.

In contrast, all available trip options become travel edges so that there are up to 1,025 travel edges connecting each city pair. Besides that, no matter the number of destinations, the travel network will have ideally up to 1,025 edges connecting the dummy node `START` (start edges) and the origin city and also up to 1,025 nodes connecting the final city to the dummy node `END` (end edges). Additionally, for each city, part of the arrival nodes are connected to departure nodes to form dwelling arcs according to the constraints defined in Section 4.2. In Figure 25 we show how the amount of each type of edge is affected by the dwelling time in a 1-destination route. As can be verified, the sum of start and end nodes does not surpass 1,000 once the amount of unique origin and final vertices is probably lower than 2,050. Indeed, the average amount of all vertices for a 1-destination does not surpass 2,000 as showed in Figure 24. In this particular example, there are 2 city pairs (origin city/destination city and destination city/final city) so that the expected number of travel edges is up to 2.050. In fact, the trip edge column of Figure 25 worths approximately this value. As the amount of dwelling days vary, only the number of dwelling edges decay while the amount of start, end and trip edges remain unaltered. The lower the dwelling time in a city the bigger is the number of valid dwelling arcs that may be created taking in consideration the time component of an arrival node. In our 1-travel destination example, when the dwelling time is set to 1 day, the tourist may arrive in the destination in almost any day which is within the 41-days travel window. Inversely, there are few arriving nodes that can be expanded into dwelling edges when the dwelling time is 30 days. In this case, the tourist must arrive in the city at most until the 10th day of the travel window, thereafter limiting the number of valid dwelling options.

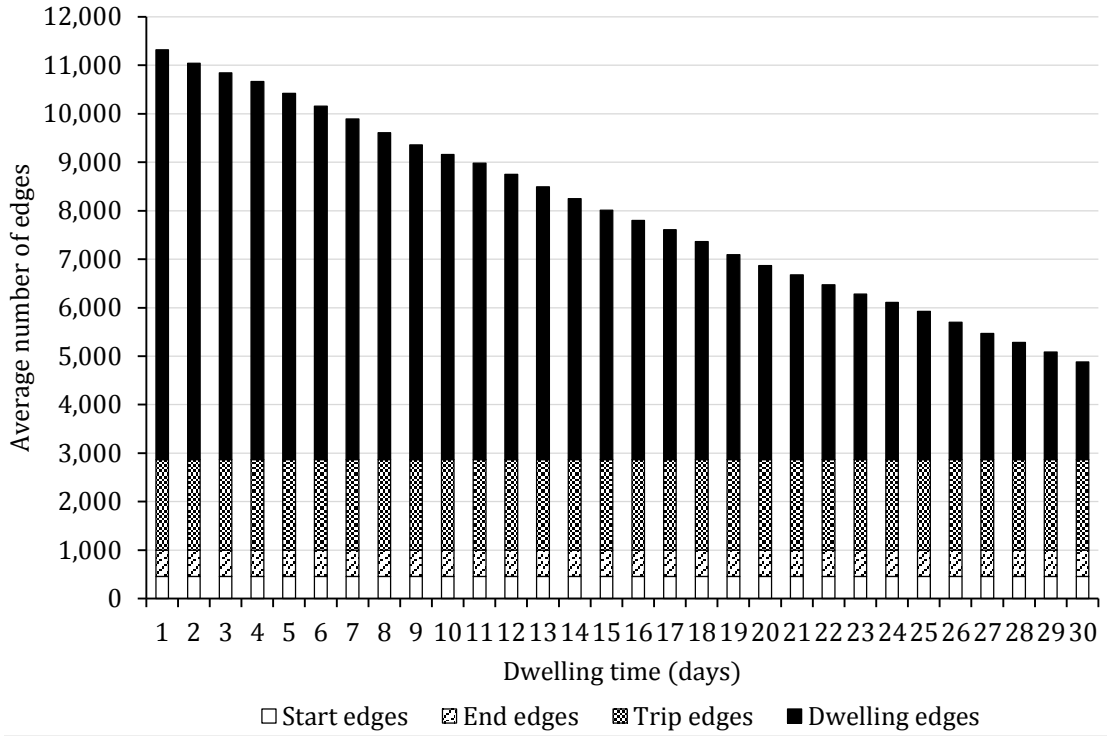

Figure 25 – Impact of the dwelling time in the number of edges for a 1-destination travel. While the number of edges related to the trip, start and end auxiliary nodes remain fixed, the number of dwelling edges decreases as the dwelling time increases. The average number of vertices (1851) remain the same regardless the dwelling time.

Although we set the max extra time in a city c to 2 days ($maxExtraTime_c = 2$) we also tested the influence of this variable in the number of edges. In Figure 26 we show the results of varying the extra period from 1 to 5 days in routes involving from 1 to 5 cities. The results follow the same pattern regardless the city: while the number of start edges, end edges and travel edges remain the same, the amount of dwelling edges gets bigger as the extra dwelling period increases. This trend is a result of the addition of a number of valid dwelling options that emerge as soon as the dwelling time limit in a city increases.

5.4.2 Travel network building: sequential vs. parallel

In order to assess the suitability of our network building parallel method described in Section 4.4, we compare it with a sequential implementation. Instead of simultaneously adding the data of all possible city pairs of a sequence S, the sequential version follows a regular building process in which the travel network is created gradually, i.e., the vertexes and arcs of a city pair are added to the travel network only after the vertexes and arcs of the predecessor pair in the sequence were completely added.

For testing these building algorithms we adopted a subset of the test cases described in Table 5: for each possible destination set in Table 4, we vary the number

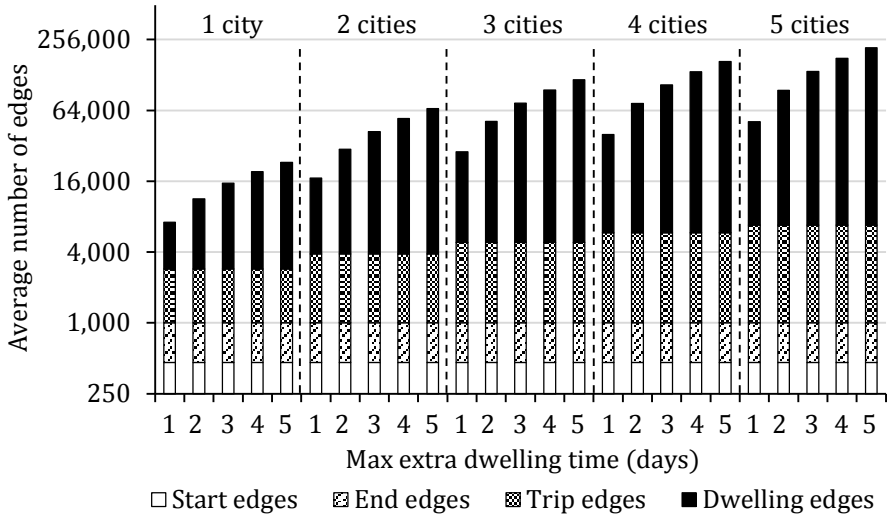

Figure 26 – Average amount of edges considering an increasing number of max over dwelling times for 5 routes with different number of cities. The graph is portrayed in logarithmic scale (base 2).

of cities from 6 to 14 and restrict the dwelling times to 1 and 2 days only. This restriction guarantees that we will execute the same quantity of test cases for each number of cities. These settings result in a set of test cases which comprises 270 instances (15 collections of cities $\times$ 9 sets of different sizes $\times$ 2 dwelling options), 30 for each different number of cities. The greedy algorithm presented in Section 5.2.2.1 is applied to each instance to rapidly create a starting route for the building methods. Then, for each cities' set of different sizes, we calculate the average execution time for the 30 possible instances, using both of our network building versions, and show the results in Figure 27.

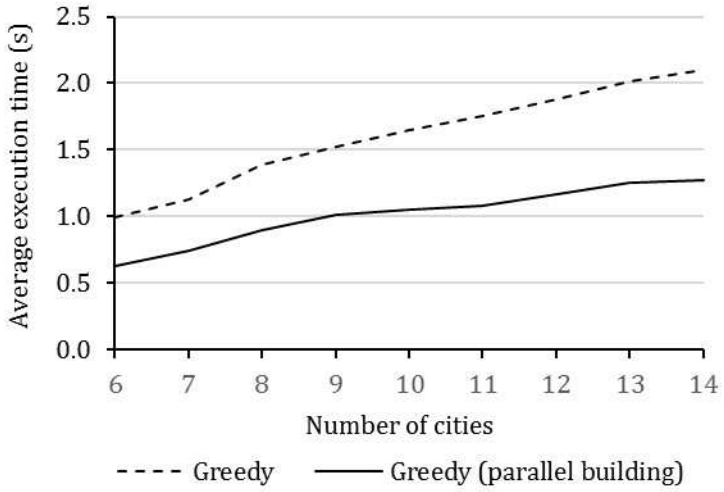

Figure 27 – Sequential travel network building versus parallel travel network building on a sequence of visits generated by a greedy algorithm. The parallel approach reached in average a 1.6x speedup over the sequential implementation.

Our parallel approach has reached in average a 1.6× speedup over the sequential implementation, taking about 1.3 seconds to create the travel network for a single route when 14 cities are concerned. Regarding the dwelling time, these results may be considered as an upper bound in a sense that longer dwelling times would reach, in average, lower execution times. In fact, as discussed in Section 5.4.1, the smaller the dwelling time and the bigger is the number of cities, the more complex is the travel graph and the longer is the execution time. Therefore, if longer dwelling times were considered, the average time taken to build a travel network for routes containing from 6 to 14 destinations would not surpass the values depicted in Figure 27. Once the graph building is one of the most executed procedures of our heuristic method, we decided to set the parallel implementation as the default travel graph building policy of this study.

5.4.3 Brute force method

We run our parallel single objective brute force method using all instances presented in Table 5 and show the average processing time by number of cities in Figure 28. Due to the combinatorial nature of the TSP, each new city added to a route greatly increases the number of possible travel sequences since we are dealing with a factorial rate of growth. When the route comprises 9 cities for example, there are 362,880 possible routes, and the average execution time to calculate all possible routes considering our 60 test cases was 31h and 22 minutes. The challenge posed by a 10-destinations test case is much bigger: our brute force method would have to examine 3,628,800 candidate routes to reach the best solution. Therefore, for more than 9 destinations, it would be impractical to determine an exact solution due to the high execution time involved in the process.

In contrast, from 1 to 5 cities, there is no need to apply a heuristic since the optimum solution is determined in short time by checking all possible routes. Even for 6 cities, it would take around 3 minutes to get an optimal solution. However, we still insist in applying an heuristic approach when 6 cities are concerned due to the singular characteristics of our testing environment which, besides preprocessing the input data, comprises a high performance computer cluster node with enabled parallelism. Indeed, in a real world setting, many delays would probably be involved once a regular server running the application would have a limited amount of resources to share among several users. On the contrary, up to 5 cities even if the method lasted 1s to determine the shortest path of a travel graph and this procedure was performed sequentially, it still would take less than 3 minutes to calculate all possible routes.

Hence, for testing the quality of our heuristic approach we use a subgroup of 285 test cases containing all instances from Table 5 that have from 6 to 9 destination cities. In Table 6 we separate the average execution times by number of cities and

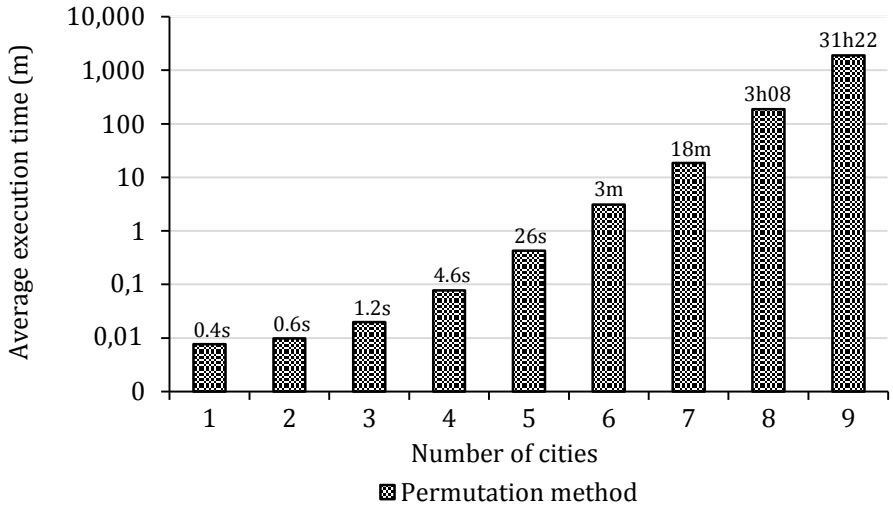

Figure 28 – Average execution time of the brute-force approach for the test cases presented in Table 5. Results were grouped by number of cities and presented in logarithmic scale (base 10).

Table 6 – Average execution time per dwelling time of the brute-force approach for tests cases containing 6, 7, 8 and 9 cities.

	Dwelling time (days)					
#Cities	**1**	**2**	**3**	**4**	**5**	**6**
6	0:03:13	0:03:09	0:03:08	0:02:54	0:02:41	0:02:44
7	0:20:27	0:18:35	0:17:33	0:17:26	0:17:02	-
8	3:34:08	3:09:35	2:56:37	2:51:18	-	-
9	35:41:47	32:41:53	29:43:03	28:09:19	-	-

dwelling time. As discussed in Section 5.4.1 the smaller the dwelling time the more complex is the travel network, and this additional complexity is converted in longer execution time no matter the number of cities. While a difference in dwelling times represent a couple of minutes in a 6-destinations instance, it may represent up to 7h hours when 9 cities are considered.

5.4.4 Parallel 2-Opt

The complete 2-Opt local search is the leading procedure in the exploration for a promising solution. As stated in Algorithm 1, it is repeated every time a perturbation is performed so that any reduction in its processing time would immediately lead to a decrease in the execution time of the ILS method as a whole. In Figure 29 we show a comparison of both sequential and parallel implementations of our 2-Opt algorithm. Similarly to the tests performed in Section 5.4.2 for the graph building procedure, we selected all test cases from Table 5 that comprises 6 to 14 cities and 1 to 2 dwelling days totalizing 270 instances. In this particular test, both 2-Opt versions implement the parallel building method not using the memory set to discard solutions. Nevertheless, the parallel implementation of the 2-Opt local search is able

to reach a solution in about half of the execution time of a regular sequential implementation, regardless the number of the destinations. This relation remains valid even when the number of parallel 2-Opt movements are expressive: if 14 destinations are considered, for example, 91 threads are created to perform all movements available. Due to the successful outcome of the parallel 2-Opt implementation, it became the default local search procedure of our ILS algorithm.

When integrated to the ILS algorithm, the parallel 2-Opt process is further enhanced by the memory set's utilization. Since all routes created during the execution of a single test case are stored in a memory set, many threads are immediately terminated when they are about to check a route that is already in the list. This performance enhancement play a big role especially when a small number of cities are concerned. For 6-destinations test cases for example, there are only 720 different routes so that it is much more likely that the iterative application of the 2-Opt algorithm eventually reaches a previously calculated route.

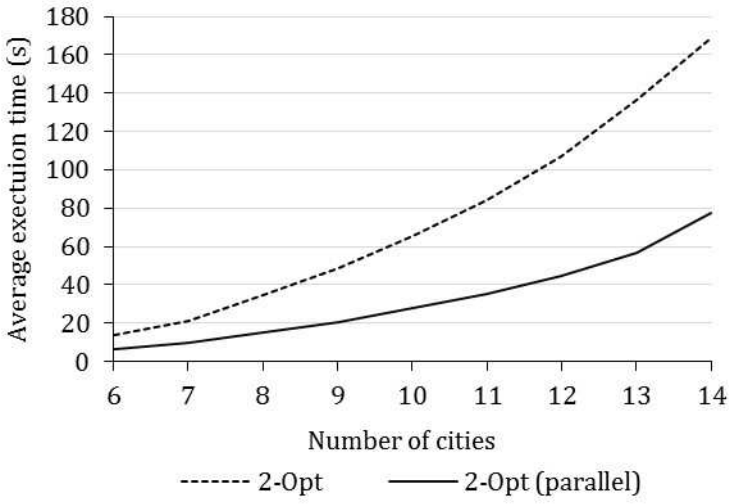

Figure 29 – 2-Opt speedup due to parallel implementation.

5.4.5 ILS: sequential DB vs. parallel DB

In this section we assess the performance of several ILS configurations concerning the double bridge strategies of Section 5.2.2.4. Firstly, we combine the ILS method presented in Section 5.2.2 with the sequential double bridge approach in which we simply repeat the perturbation/local search duet for $maxNoImproveIt$ times. In this case, we are especially interested in evaluate how the number of iterations affects the quality and the total processing time of a solution. To do so, we create 5 configurations in which we vary the $maxNoImproveIt$ parameter from 1 to 5 iterations. Then , for better identification, we label these configurations as follows: `1_NO_IMP`, `2_NO_IMP`, `3_NO_IMP`, `4_NO_IMP` and `5_NO_IMP`. Next, regarding our parallel implementation of the double bridge method, we intend to determine how many

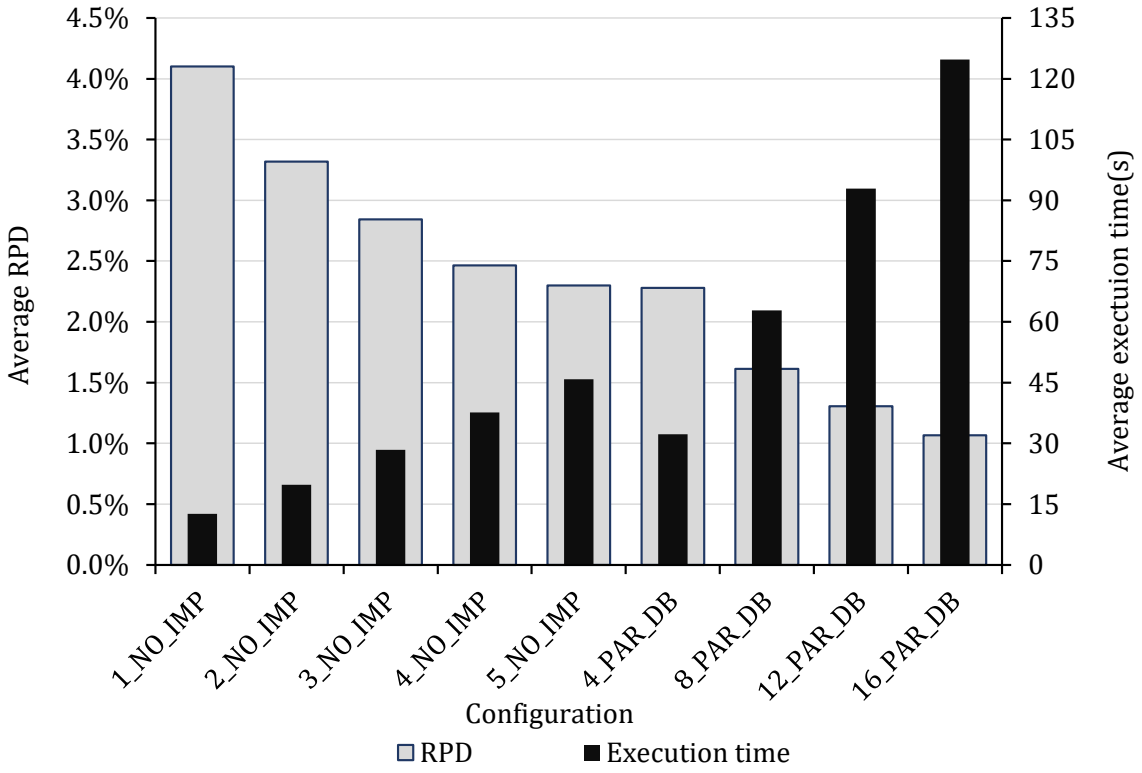

Figure 30 – Relation between average RPD and average execution time for the tested configurations. The lower is the average RPD the better is the configurations' quality.

perturbations can be performed in parallel without harming the ILS performance. In this case, we set the $maxNoImproveIt$ parameter to 1 iteration, and focus only in the variation of the number of parallel double bridge movements, creating therefore, 4 configurations in which this number is set to 4, 8, 12 and 16 parallel perturbations. Finally, we label these parallel double bridge settings as follows: 4_PAR_DB, 8_PAR_DB, 12_PAR_DB and 16_PAR_DB.

Due to the findings of Section 5.2.1 regarding the outcomes of the brute force method, we run the 9 configurations of our heuristic approach in 285 instances comprising from 6 to 9 cities. The quality of a solution is calculated throughout the application of the RPD metric. The total cost of the solution produced by an ILS configuration applied to a single test case assumes the f_s variable of the formula while the optimal value found by the brute force approach for the same test case is assigned to variable f_{best}. The average quality and execution time of each configuration regarding all test cases available are shown in Figure 30. The configurations were sorted by their RPD values from 16_PAR_DB (lowest RPD) to 1_NO_IMP (highest RPD).

As expected, the ILS algorithm in which we embedded our sequential double bridge procedure was able to reach better solutions as the number of iterations grows. However, the improvement caused by the increment of the $maxNoImproveIt$

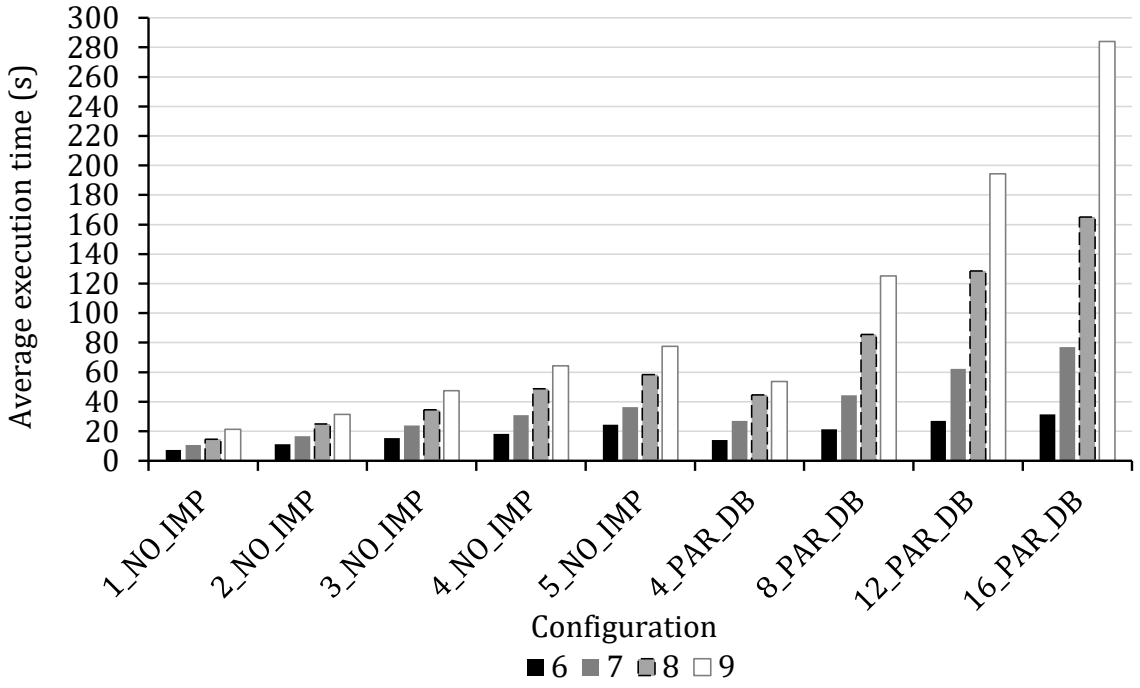

Figure 31 – Average execution time by number of destinations for each ILS configuration available.

variable gets smaller from a configuration to another: the configuration `2_NO_IMP`, for instance, is in average almost 1% better than the configuration `1_NO_IMP` whereas the difference between the configurations `4_NO_IMP` and `5_NO_IMP` is lower than 0.25%. In contrast, the average execution time of a configuration increases steadily, around 8s for each iteration added. Regarding the ILS version comprising the parallel double bridge method, our configurations were also able to consistently decrease the average RPD value as the amount of parallel perturbations is increased. The higher the number of parallel threads, the smaller is the average quality improvement from one configuration to another. Just as the `n_NO_IMP` configurations, the execution time grows in a nearly fixed rate: starting from 4 parallel perturbations, the addition of 4 new double bridge threads increases about 30s to the average execution time. Hence, in general, the higher is the quality of a configuration the longer it takes to reach its solutions.

In Figure 31 we show the average execution time by number of cities for each configuration. Generally, the higher the number of destinations, the longer is the execution time, regardless of the configuration. As discussed in Section 5.4.1 there is a direct relation among the number of destinations and the complexity of the travel network. For some number of destinations, the average execution time of the configurations far surpasses our threshold of 1 minute. It is extremely important to enforce this limitation once a real world application relies on volatile travel data, so that a response has to be swiftly built before any major changes occur in the data pool. In Figure 32 we show the average RPD value also grouped by number of cities

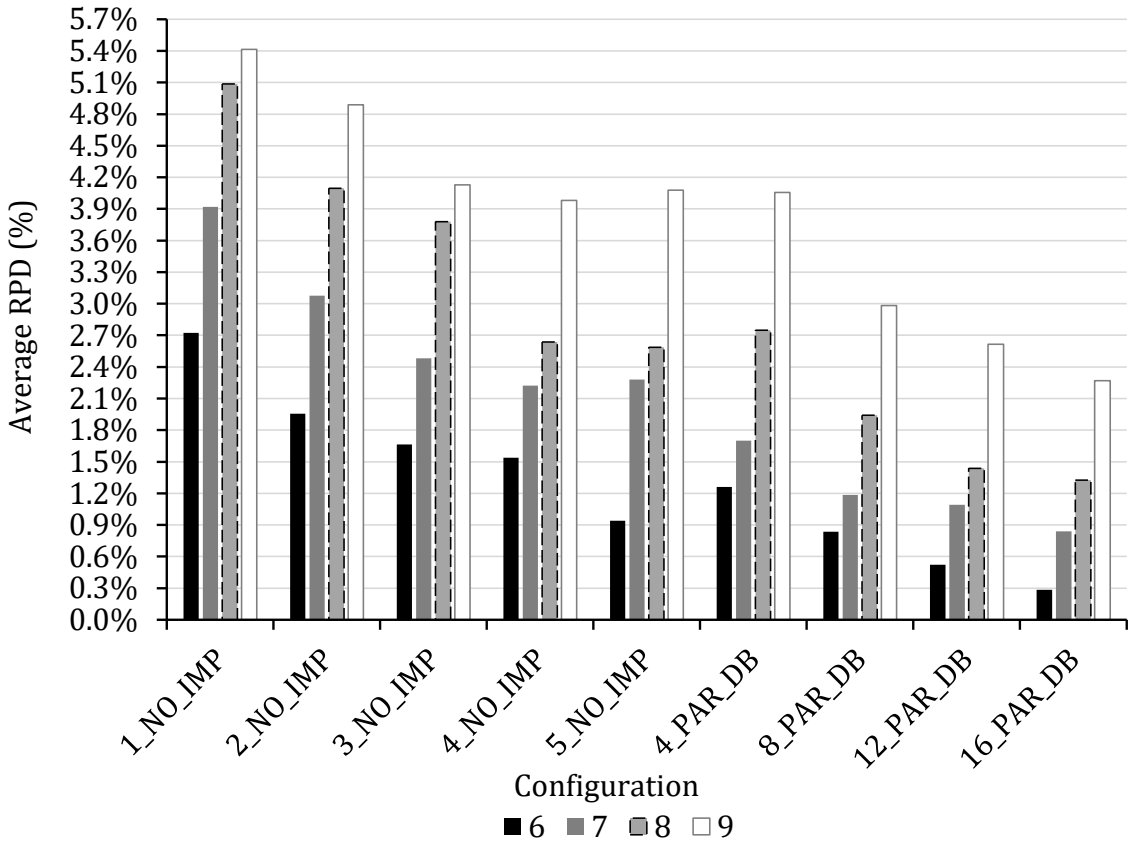

Figure 32 – Average RPD by number of destinations for each ILS configuration available.

for each configuration and in Table 7 we show how many optimal solutions each configuration was able to produce. As the travel network becomes more complex, the quality of the solutions systematically decreases: all configurations were able to reach at least 10 optimal solutions when 6 cities were considered but few could reach more than 1 optimal solution for 9 destinations. As good as some of these configurations may produce quality solutions to a particular test case, their solutions would be misleading if they took several minutes to be calculated. Therefore, for each number of cities we choose a configuration that a) does not take more than 1 minute to execute and b) has the best quality among its counterparts. To have a clearer idea about how much money a 0.1% increase in the RPD metric represents, we calculated the average prices of the optimal solutions by the number of destinations available. Then, for 6, 7, 8 and 9 cities we have the following average overall costs: \$2089.18, \$2126.21, \$2185.60 and \$2331.85. Thereafter a 0.1% growth in the RPD metric would represent an increase of \$2.09, \$2.13, \$2.19 and \$2.33 respectively.

Firstly, for 6 cities our `16_PAR_DB` method was able to reach the optimal solution in 62 of 90 test cases, diverging in average only 0.3% from the optimal results. Therefore, once the average price of a 6-destinations travel is \$2,089.18, the average solution produced by this method is at most 7 dollars more expensive than the optimal solution besides taking in average 30 seconds to be executed. Next, for

Table 7 – Number of test cases in which a configuration reached the optimal solution (RPD value equals 0) separated by number of destinations. The total column sums up the amount for each configuration and the ratio column shows the percentage that this amount represents in the 285 test cases.

	#Cities (total of test cases)					
Configuration	**6(90)**	**7(75)**	**8(60)**	**9(60)**	**Total**	**Ratio**
16_PAR_DB	62	25	8	4	99	34.7%
12_PAR_DB	48	25	10	2	85	29.8%
8_PAR_DB	37	20	4	3	64	22.5%
4_PAR_DB	27	11	1	0	39	13.7%
5_NO_IMP	31	7	2	0	40	14.0%
4_NO_IMP	27	9	1	1	38	13.3%
3_NO_IMP	26	3	3	0	32	11.2%
2_NO_IMP	21	5	3	1	30	10.5%
1_NO_IMP	14	5	1	0	20	7.0%

7-destinations test cases, our `8_PAR_DB` stands out by processing its solutions in about 44s besides reaching the optimal solution in 25 of 60 test cases, diverging in average 1.1% from the optimal results, i.e., less then $24 above the average optimum price. Then, for 8 cities, the configuration that less diverged from the optimum solution (2.7% or $59.01 more expensive) while being able to be processed in less than 1 minute (44s) was the `4_PAR_DB` method. Finally, when 9 cities are concerned, the `4_PAR_DB` configuration was again the best viable options, taking 53s to reach solutions in average 4.1% ($95.60) divergent from te optimal results.

6 Bi-objective approach

This chapter compiles all strategies that we implemented to solve the bi-objective travel planning problem in which a tourist is interested in minimizing, besides cost, the total waiting time of his journey. Similarly to cost calculation, the waiting time is comprised of transport and accommodation components. In fact, we consider as waiting, the time that a tourist spends moving between cities and also the time he can not access accommodation's facilities, i.e., the non-booked time he must wait before check-in (to have access to a bedroom) and after check-out (to board transportation). As for our single-objective implementation, this version also has a crucial execution time limitation: a set of solutions has to be achieved as fast as possible since travel data is extremely volatile. Knowing this, we also set our stop criteria as a limit of 1 minute of execution. Firstly, in Section 6.1, we explain how the concept of Pareto fronts can be applied to this study. Then, in Section 6.2 we explain how the objective function of each travel route is calculated. Next, in Sections 6.3 and 6.4, we present the strategies proposed by Deb et al. (2002) to separate a set of solutions into Pareto fronts and to classify solutions within the limits of a single front. Next, in Section 6.5, we present two solving strategies to deal with this particular version of the problem. The first is a parallel brute-force bi-objective algorithm (Section 6.5.1) and the second is a classic framework, the nondominated sorting genetic algorithm II (NSGA-II) (Section 6.5.2). Then, in Section 6.6, we show what test cases of Section 5.3 are executed, introduce the performance metric applied to assess the quality of our NSGA-II implementation, and present the calibration parameters of the genetic algorithm. Finally, in Section 6.7, we present the results of our methods and discuss what GA parameters are best suited for each group of test cases.

6.1 Pareto front

According to Zhou et al. (2011), a *multiobjective optimization problem* (MOP) can be mathematically formulated as follows:

$$\begin{aligned} &\min F(x) = \ (f_1(x), ..., f_m(x)) \\ &s.t. \ \ x \in \Omega \end{aligned} \tag{6.1}$$

where Ω is the decision space and $x \in \Omega$ is a decision vector. $F(x)$ consists of m objective functions $f_i : \Omega \ \rightarrow \ R, i = 1, ..., m$, where R^m is the objective space. In this study, we are dealing with a biobjective problem ($m = 2$) once we intend to minimize both cost and waiting time of a journey. The decision vectors, in turn, consist of the possible sequences of destinations. As mentioned in Section 1.2,

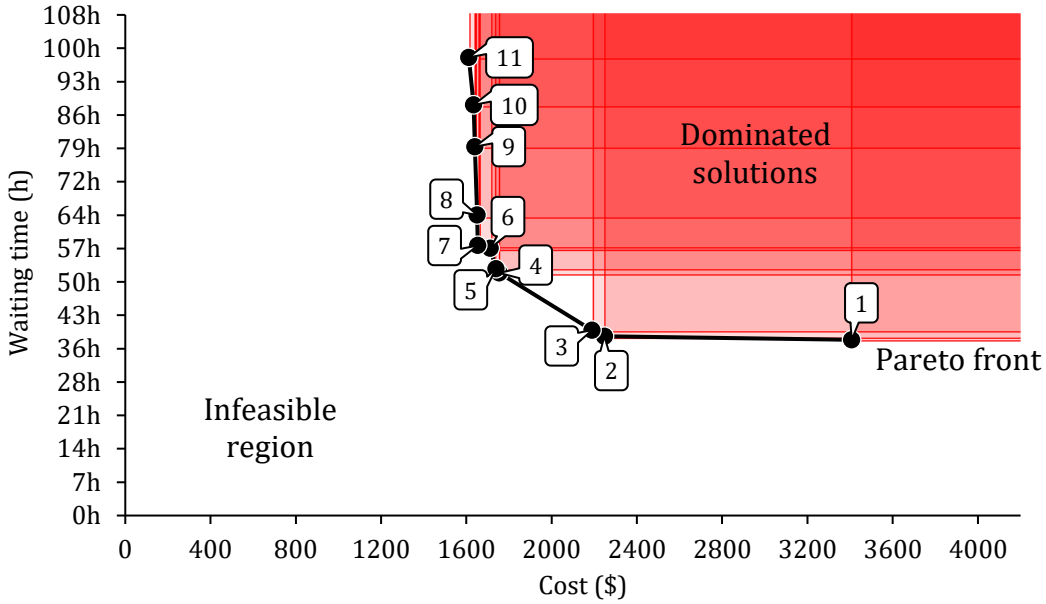

Figure 33 – Optimal Pareto front for a 5 cities route.

seldom there is a solution able to optimize all objectives at once so that Pareto optimal solutions must be provided to the decision maker. As cited in (ZHOU et al., 2011), the Pareto optimality concept was proposed by Edgeworth and Pareto, and can be formally defined as follows:

Definition 1. A vector $u = (u_1, ..., u_m)$ is said to dominate another vector $v = (v_1, ..., v_m)$, denoted as $u \prec v$, iff $\forall i \in 1, ..., m, u_i \leq v_i$ and $u \neq v$.

Definition 2. A feasible solution $x^* \in \Omega$ of problem 6.1 is called a Pareto optimal solution, iff $\nexists y \in \Omega$ such that $F(y) \prec F(x^*)$. The set of all the Pareto optimal solutions is called the Pareto set (PS), denoted as $PS = \{x \in \Omega | \nexists y \in \Omega, F(y) \prec F(x)\}$. The image of the PS in the objective space is called the Pareto front (PF) $PF = \{F(x) | x \in PS\}$.

Figure 33 shows the Pareto front for the following input data: $I = F = \{RIO\}$, $C = \{FRA, MOW, IST, AMS, MAD\}$, $Tmin_c = 1$ and $maxExtraTime_c$ =2 for $c \in C$, d_{start} = 2015/12/01 and d_{end} = 2016/01/10. This front is comprised of 11 solutions that together dominate all the solutions in the upper right colored area but are not strictly dominated by each other. The infeasible area does not have any solution due to the problem's constraints. Indeed, a utopia solution would be (0, 0), i.e., a travel with no cost and no waiting.

After the optimization method's completion, the decision maker faces a set of Pareto solutions and have to choose the one which suits best to his preferences. This approach is classified in (ZHOU et al., 2011) as a posteriori method since it uses the DM's preference information after the search process. This method matches perfectly with the travel planning problem since the traveler assumes the role of the decision maker, being responsible for choosing the best itinerary of the Pareto Set.

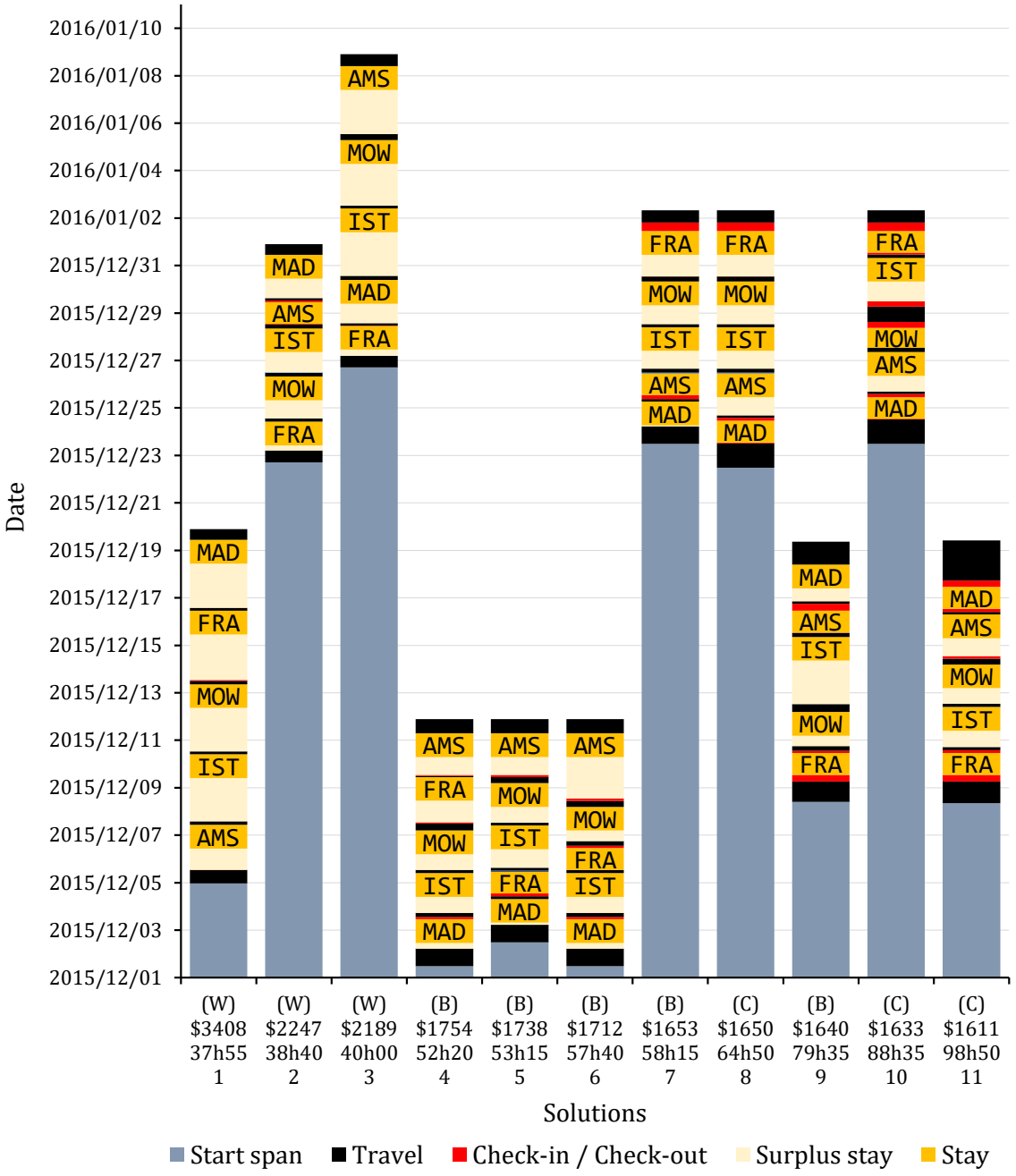

Figure 34 – Detailed Pareto front for the 5 cities route example summarized in Figure 33. Below each solution we show a number corresponding to to the solution's label besides its total cost and waiting and also a letter W(wait), C(cost), B(balanced) to identify what objective was used when applying the shortest path algorithm. Section 6.2 explains the strategy behind this objective labeling.

Figure 34 shows how this set could be further detailed to be presented to a traveler so that it is easier to decide for an option given overall solution's features.

We stacked all time related elements that compose a solution so that the decision maker can rapidly identify where each possible journey is located in the travel window. Every solution begins with a "Start span" which consists of the interval

between the departure instant of the first flight and the minimum starting instant of the journey. While the "Stay" intervals represent the planned visiting period in a city, the "Surplus stay" intervals show how much extra dwelling time the travel planner method booked for the city, apart from the specified amount of 1 day. The purpose of this differentiation is showing how the actual time spent in each city normally surpasses the predefined dwelling time, due to the max extra time relaxation of 2 days, in order to get the best deal. Indeed, finding the best solution, sometimes involves booking extra nights in order to enhance the journey as a whole. The cheapest travel to the next city in the sequence may be scheduled some days ahead of the originally planned dwelling period, so that, from a global perspective, it is advantageous to book extra nights and wait for this cheap travel. On solution 9 for example, the stay in Istanbul is only 4h lower than the maximum extra time, being 2 days and 20h long (from 2015/12/12 14:30 to 2015/12/15 10:30). In turn, the black intervals represent the flight's travel times of each solution. Notice that all solutions share thick black intervals in the start/end since these correspond to the intercontinental flights from/to the start city Rio de Janeiro. We also created "Check-in/Check-out" intervals in order to clearly show to the decision maker what solutions contain the greatest amount of waiting. Starting from these overall information, a tourist may finally choose an option and verify the itinerary details. In Figure 35, for example, we show the details of solution 9 using the outline structure presented in Section 4.6.

The Pareto solutions of Figure 34 were sorted in descending order of cost, but due to the bi-objective nature of the problem they also ended up showing an increasing waiting time pattern. In general the more expensive the solution the smaller is the waiting: the optimization method will make use of all available monetary resources to buy the fastest flights and avoid waiting periods. Solution 1, for example, books extra nights in all cities to avoid uncomfortable waitings and also has some of the thinnest black/red intervals of the solutions' set. Analogously, the trade-off among cost and waiting is clearly perceptible in solution 11: its last flight, for example, is almost 2 days long. Apart from the extremes, the middle solutions tend to present more balanced options. When comparing the 4^{th} and the 11^{th} solutions for example, the traveler has to check if it worths paying around $100 more to wait approximately 50h less. This decision making process adds an additional level of freedom once the traveler no longer needs to rely in a single option, having the opportunity to effectively decide what is best.

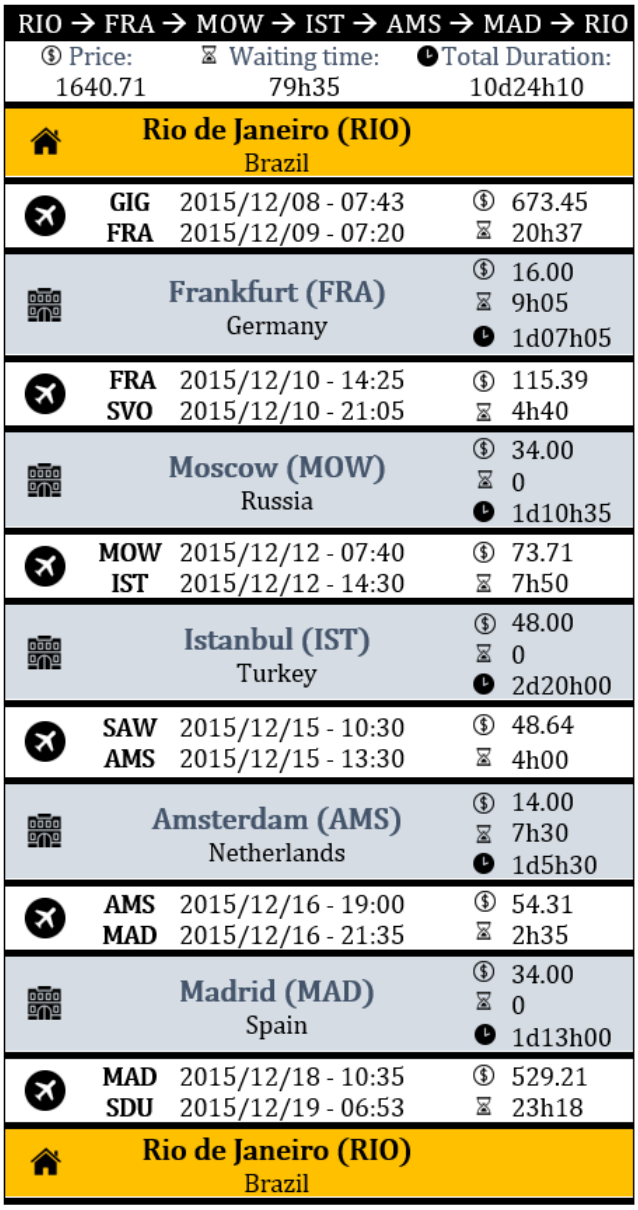

Figure 35 – Details of the solution 9 presented in Figure 34.

6.2 Balanced objective function

Each travel route, i.e., cities' permutation, generates a unique travel graph comprised of flight and accommodation arcs that have both cost and time components. Therefore, a shortest path algorithm would have to consider one or both of these objectives in order to achieve a solution. Since we also want to determine the best sequence of visits, applying a multi-objective shortest path approach in each possible permutation would be very time consuming. Hence, we chose to apply a multi-objective strategy only on route determination, not on route's path evaluation. However, although our shortest path algorithm does not produce a nondominated front of paths for a given route, it produces at least two extremes of this front: the shortest path method is run twice, considering each objective at a time. Besides

that, we also produce a third path, based on a combination of these two objectives. For each travel/dwelling arc, we define a weighted value b corresponding to a balanced summation of cost and waiting data:

$$b = w_1 * cost(\$) + w_2 * waiting(h)$$

Further examination in our travel data showed that in average, each travel hour costs about 30 monetary units, i.e., the cost is 30 times bigger than the waiting time. Thereafter, the simple sum of cost and waiting would not be effective for determining a balanced value once the waiting time would be overly misrepresented. Hence, we chose the weight parameters as follows: $w_1 = 1.0$ and $w_2 = 5.0$. This way we are able to increase the relevance of the waiting time and also keep the prominence of the cost when determining the shortest path. From another perspective, we are stating that 1h of the tourist's time worths 5 monetary units. In a realistic travel planner application, these weights could be part of the traveler's profile and set by him/her prior execution.

6.3 The fast nondominated sort

In order to identify the Pareto fronts of a population of candidate solutions, we applied the fast nondominated sort proposed by Deb et al. (2002) and showed in Algorithm 4. For M objectives and N candidates, this approach requires $O(MN^2)$ computations. Each solution p has a domination count n_p, which stores the number of solutions that dominate the solution p, and also a set S_p to store the solutions dominated by p. Each solution p is compared with the q remaining solutions and the n_p variable is incremented every time q dominates p. In turn, the set S_p receives a new element q every time p dominates q. After comparing all candidates against the current p solution, if n_p still remains zeroed, p is not dominated by any other solution, hence, it belongs to the first front (F_1). Once n_p and S_p were determined by every p solution in population P, and F_1 is already set, the algorithm iteratively determine the other fronts by updating the n_q values of each solution q dominated by a solution p pertaining to the front F_i, being i the front counter. Figure 36 shows an example containing 29 solutions which were separated into 4 fronts. Among these solutions, there is the optimal Pareto front presented in Figure 33. The colored boxes mark the dominated area of each solution, i.e., every solution within the box of a point p belongs to S_p. Conversely, a solution q which is in the area of n boxes is dominated by n solutions, i.e., $n_q = n$.

Algorithm 4 Fast non dominated sort presented in (DEB et al., 2002)

```
1:  function FAST-NON-DOMINATED-SORT(P)
2:      for each p ∈ P do
3:          S_p = ∅
4:          n_p = 0
5:          for each q ∈ P do
6:              if p ≺ q then                          ▷ If p dominates q
7:                  S_p = S_p ∪ {q}                    ▷ Add q to the set of solutions dominated by p
8:              else if q ≺ p then
9:                  n_p = n_p+1                        ▷ Increment the domination counter of p
10:         if n_p = 0 then                            ▷ p belongs to the first front
11:             p_rank = 1
12:             F_1 = F_1 ∪ {p}
13:     i = 1                                          ▷ Initialize the front counter
14:     while F_i ≠ ∅ do
15:         Q = ∅                                      ▷ Used to store the members of the next front
16:         for each p ∈ F_i do
17:             for each q ∈ S_p do
18:                 n_q = n_q - 1
19:                 if n_q = 0 then                    ▷ q belongs to the next front
20:                     q_rank = i+1
21:                     Q = Q ∪ {q}
22:         i = i + 1
23:         F_i = Q
```

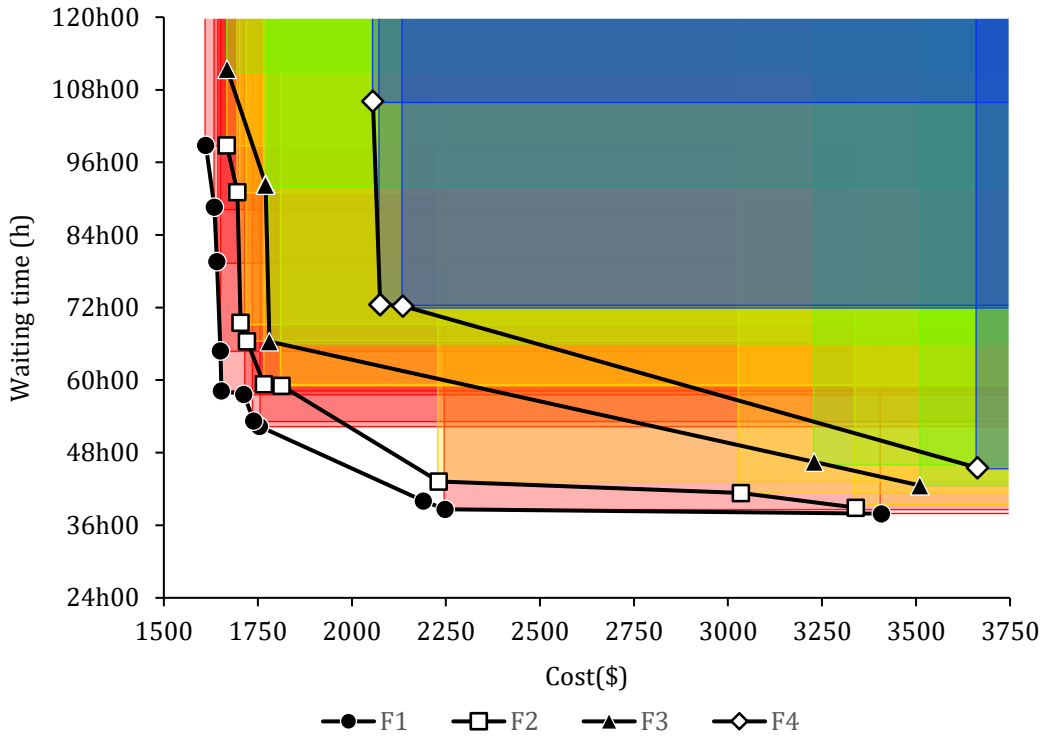

Figure 36 – Plot of a population of candidate solutions for a 5 cities route example after applying the fast nondominated sort algorithm. Each colored box highlights the area dominated by a single solution.

6.4 Ranking front's solutions

After applying the nondominated sort procedure we are able to separate a population of solutions into Pareto fronts. Next, to rank the individuals of a single front, we apply the crowding distance metric proposed in (DEB et al., 2002), so that we can compare two solutions for their extent of proximity. The lower a solution's crowding distance, the more crowded it is by other solutions. In fact, a small crowding distance indicates a lack of unique features, or relevant differences among a solutions' immediate counterparts. Algorithm 5 shows the details of the implementation of this ranking strategy. For each solution p, the crowding distance corresponds to the sum of distance values for each objective, i.e., the normalized difference in the function values of the two immediate adjacent solutions of p, considering one objective at a time. Table 8 illustrates the application of the crowding distance metric on the Pareto front presented in Figure 33. The clusterization of solutions in the middle of the front is backed up by the crowding distance values: solutions 5,6 and 7 have the smallest distances once they have not significant differences among themselves. On a travel planning application, some of them could even be suppressed to ease the decision making process. In contrast, the boundary points must be part of the set of candidate solutions once they present the limitations of the optimization method regarding each objective.

Algorithm 5 Solution crowding distance assignment presented in (DEB et al., 2002)

1: **function** CROWDING-DISTANCE-ASSIGNMENT(I)
2: $l = |I|$ ▷ number of solutions in I
3: **for** $i = 0$ to $i = l$ **do**
4: $I[i]_{distance} = 0$ ▷ initialize distance
5: **for each** objective m **do**
6: $I = \text{sort}(I,m)$ ▷ sort using each objective value
7: $I[1]_{distance} = I[l]_{distance} = \infty$ ▷ boundary points are always selected
8: **for** $i = 2$ to $(l - 1)$ **do** ▷ for all other points
9: $I[i]_{distance} = I[i]_{distance} + (I[i + 1].m - I[i - 1].m)/(f_m^{max} - f_m^{min})$

6.5 Solving strategies

The following sections present the solving strategies we used to approach the bi-objective version of the travel planning problem. Section 6.5.1 shows the brute-force method created to reach the exact solutions and Section 6.5.2 describes the framework used to produce approximated solutions.

Table 8 – Crowding distances of the Optimal Pareto front for the 5 cities example presented in Figure 33.

Label	Route	Cost	Waiting	Crowding distance
11	[RIO, FRA, IST, MOW, AMS, MAD, RIO]	1611.61	98h50	∞
1	[RIO, AMS, IST, MOW, FRA, MAD, RIO]	3408.09	37h55	∞
2	[RIO, FRA, MOW, IST, AMS, MAD, RIO]	2247.65	38h40	0.71
3	[RIO, FRA, MAD, IST, MOW, AMS, RIO]	2189.98	40h00	0.50
4	[RIO, MAD, IST, MOW, FRA, AMS, RIO]	1754.45	52h20	0.47
9	[RIO, FRA, MOW, IST, AMS, MAD, RIO]	1640.71	79h35	0.40
8	[RIO, MAD, AMS, IST, MOW, FRA, RIO]	1650.36	64h50	0.36
10	[RIO, MAD, AMS, MOW, IST, FRA, RIO]	1633.87	88h35	0.33
7	[RIO, MAD, AMS, IST, MOW, FRA, RIO]	1653.36	58h15	0.15
6	[RIO, MAD, IST, FRA, MOW, AMS, RIO]	1712.39	57h40	0.13
5	[RIO, MAD, FRA, IST, MOW, AMS, RIO]	1738.65	53h15	0.11

.

6.5.1 Parallel bi-objective brute-force approach

As a comparative benchmark, we implemented a brute-force method to find the optimal solution for some of our short test cases. Firstly, we define a void Pareto optimal set P which will eventually hold the optimal front. Considering a route of size n, we progressively generate all $n!$ possible travel sequences: a set of 24 permutations are evaluated at a time through parallelization of the building and checking processes. As mentioned in Section 6.2, after building the graph we check the shortest path from origin to destination regarding three objectives: cost, waiting and fitness. Hence, at the end of checking, we have a temporary set T with 72 solutions. This set is then merged with the current set P, and the fast nondominated sort (Section 6.3) is applied to rank the individuals in nondominated fronts. After the sorting, we keep only the current best Pareto front in P, excluding all other fronts from this set. The process of evaluating, merging, sorting and excluding is then repeated until there are no combinations to evaluate. In the end, the set P will contain the overall nondominated solutions, i.e., the optimal Pareto front for a n cities travel. In Section 6.7.1, we show all the optimal fronts reached by this strategy and discuss how our threefold evaluation method was able to fill the solution space.

6.5.2 NSGA-II

According to Zhou et al. (2011), there is a growing interest in applying evolutionary algorithm (EA) on MOP due to their population-based nature. By evolving a pool of solutions, *multiobjective evolutionary algorithms* (MOEAs) are able to approximate the Pareto optimal set in a single run. Therefore, for generating an approximated solution, we applied a popular framework: the non-dominated sorting genetic al-

gorithm II (NSGA-II) proposed in (DEB et al., 2002). The method was originally created to alleviate the common problems of its predecessor, mainly related with lack of elitism, high computational complexity of nondominated sorting and the need for a diversity-preservation parameter. Algorithm 6 summarizes the steps we follow to adapt the NSGA-II algorithm from (DEB et al., 2002).

Algorithm 6 NSGA-II algorithm adapted from (DEB et al., 2002)

```
1:  function NSGAII(N,C)
2:      t_start = getCurrentTime()
3:      P_0 = createRandomPop(N,C)                          ▷ Section 6.5.2.2
4:      P_0 = P_0+ getGreedyChromossomes(C)
5:      fast-non-dominated-sort(P_0)                        ▷ Section 6.3
6:      crowding-distance-assigment(P_0)                    ▷ Section 6.4
7:      Q_0 = make-new-pop(P_0)                             ▷ Section 6.5.2.3
8:      t = 0
9:      while getCurrentTime() - t_start < 1min do
10:         R_t = P_t ∪ Q_t
11:         F = fast-non-dominated-sort(R_t)                ▷ Section 6.3
12:         P_{t+1} = ∅
13:         i = 1
14:         repeat                                          ▷ Section 6.5.2.4
15:             crowding-distance-assigment(F_i)            ▷ Section 6.4
16:             P_{t+1} = P_{t+1} ∪ F_i
17:             i = i + 1
18:         until |P_{t+1}| + |F_i| ≤ N
19:         sort(F_i, ≺_n)
20:         P_{t+1} = P_{t+1} ∪ F_i[1 : (N − |P_{t+1}|)]
21:         Q_{t+1} = make-new-pop(P_{t+1})                 ▷ Section 6.5.2.3
22:         t = t + 1
```

6.5.2.1 Chromosomes creation and evaluation

Firstly, regarding the chromosome design of a solution, we follow the classic permutation representation of TSP, i.e., each city in a travel sequence represents a gene. Every chromosome produces a correspondent travel graph which is built in parallel, as explained in Section 5.4.2. In turn, a travel graph built from a permutation generates three different solutions throughout the application of the Dijkstra algorithm (Section 4.5), each of which in regard to a different objective, i.e., cost, waiting and balanced value, as explained in Section 6.2. Since evaluating a single solution is computational expensive, we also make use of a memory set to record all chromosomes generated throughout the method's execution. Therefore, whenever a new chromosome is to be created, its uniqueness can be checked against this memory set.

6.5.2.2 The initial population

The method starts by creating a parent population P_0 of size N containing randomly created chromosomes from the destination set C. In order to not rely only on randomness to build the initial population, we insert among the individuals a set of chromosomes produced through the application of the greedy algorithm presented in Section 5.2.2.1. Then, 3 greedily constructed chromosomes, one for each possible objective, are inserted in the first population. Next, the elements of population P_0 are ranked: first we separate the fronts through the non dominated sort algorithm (Section 6.3) and then we rank the elements of each front using the crowding distance assignment (Section 6.4).

6.5.2.3 Offspring creation: selection, crossover and mutation

Based on parent population P_i, a offspring population Q_i is produced throughout the following genetic algorithm phases:

Selection: We use a tournament selection strategy to determine what individuals of the population are chosen for mating. As described in (TALBI, 2009), this strategy consists in applying a tournament in k randomly chosen members of the group to select the best one. Hence, selecting n individuals implies running the tournament selection n times. In this study, the size k of the tournament group is set to 3. Every solution has 2 features that may be made accountable to define the winner of a tournament: the rank and the crowding distance. In general, the best solution is the one with lower rank. However, if two solutions belong to the same front, the one located in a lesser crowded region (bigger crowding-distance) is chosen.

Crossover: Parent solutions from the selected population are chosen under a crossover probability p_c, and the genetic information of its routes are crossed using the order crossover (OX) operator to generate the offspring population. After randomly selecting two crossover points, the method creates an offspring by copying the part between the two points from parent 1 and then copying the elements from parent 2 that were not already selected (TALBI, 2009). Next the process is repeated inverting the parents. Figure 37 illustrates how this OX operator is applied to produce the first offspring in a 9 cities example. While the offspring preserves only the relative order of the elements within the reorganized sections of parent 2 (2), it preserves the relative order and the absolute positions of the elements from parent 1(1). When the process is finished we check the offspring against our memory set of solutions. If they are not completely original, the process is repeated until a novel pair is found or the total number of possible combinations of crossover points p_1 and p_2 is reached. When no original offspring can be produced, the method simply returns the last produced offspring. To streamline the process, we parallelize the offspring

creation so that each pair of parents is processed in a separate thread responsible for performing the OX operation, evaluating and returning the 2 offspring.

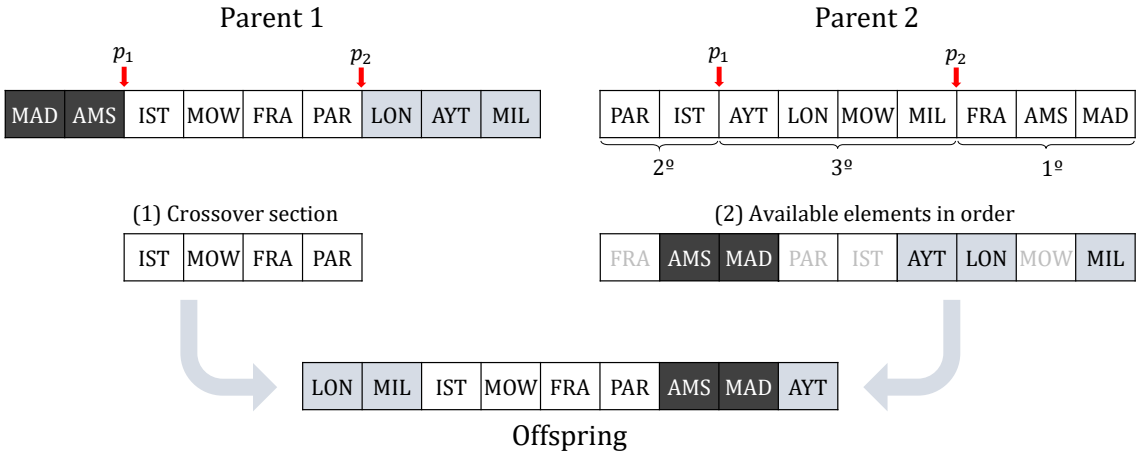

Figure 37 – Application of the order crossover (OX) on elements {`MAD, AMS, IST, MOW, FRA, PAR, LON, AYT, MIL`} and {`PAR, IST, AYT, LON, MOW, MIL, FRA, AMS, MAD`}. The background colors, black, white and gray, separate the sections of parent 1 defined by the randomly chosen points p_1 and p_2.

Mutation: The mutation operator applies the double bridge movement (Section 5.2.2.4) on the crossed over population with probability p_m. As well as in crossover, the mutation process strives to produce original individuals by checking the mutated chromosomes against our memory list. When all double bridge movements are not able to produce a novel individual, the process simply returns the last mutant. Additionally, to streamline the mutation phase, all individuals selected for mutation are processed in parallel.

6.5.2.4 Building the next population

After creating the first parent and offspring populations the method starts the generational loop which relies on execution time. Hence, while the total execution time is lower than 1 minute, the algorithm will continue to produce new generations. Subsequently, the offspring population is combined with the parent population forming the population $R_t = P_t \cup Q_t$ of size $2N$. The R_t solution is then sorted according to nondomination. The best fronts of R_t, i.e., the ones with lower ranks, are going to be part of the next generation. This combination is one of the most important steps of the method once it allows the preservation of elite members of the parent population. Therefore, the fronts of R_i are selected one at a time to construct a new population P_{t+1}, starting from the best front F_1. However, once P_{t+1} must be of size N, eventually the inclusion of a front F_i would exceed the size limitation of the population. Then, In order to choose the individuals of the front F_i that will be part of the population P_{t+1}, a sorting procedure by crowding-distance is applied.

Thereafter, the solutions that have larger crowding distance values are chosen to integrate the remaining room in P_{t+1}. Next, the population P_{t+1} of size N is submitted do the same processes of selection, crossover and mutation to create a new population Q_{t+1}.

6.6 Simulation

In this section we describe how our instances were set up (Section 6.6.1), what GA parameters of the NSGAII framework are varied to create different configurations (Section 6.6.2 and what is the performance metric we use to distinguish between Pareto fronts (Section 6.6.3).

6.6.1 Instances

The tests for our bi-objective version of the problem use some configurations already defined in Section 5.3, in which we explained how we performed the simulation for the single-objective approach. For instance, the simulation environment presented in Section 5.3.1 and the general configuration of the instances presented in Section 5.3.2 remain the same. However, due to the findings presented in Section 4.6, we chose to execute only an excerpt of the total set of test cases presented in Table 5. Firstly, in Section 5.4.3, we showed that a brute force implementation is a satisfactory approach to generate and check all possible permutations for test cases including less than 6 cities. On the other hand, a test case with 9 cities takes more than 20 hours to finish. Concerning these characteristics of the exact method, we limited to 180 the number of test cases to assess the quality of our NSGA-II approach. From 6, 7 and 8 cities, we test 4 different dwelling time lengths (from 1 to 4 days) for each one of the 15 destinations presented in Table 4. To further attenuate the arising of misleading values caused by the random nature of the method, the final value of a test case is the average of 5 executions.

6.6.2 Genetic algorithm parameters

Table 9 shows the set of calibration parameters we tested in order to produce a quality outcome from the genetic algorithm. In contrast with the original implementation by Deb et al. (2002), our method cannot run for several generations or adopt a large population since evaluating a single route is very time consuming. In fact, as pointed out before, the execution time must be confined within 1 minute. Hence, we limited the population size to at most 60 individuals. Regarding the crossover probability, we kept the value proposed in the literature in which approximately 90% of the individuals are expected to mate. The NSGA-II convergence is not harmed by a high crossover rate since the method is elitist by design: the best so-

lutions among parent and offspring populations often integrate the fronts of the next population. In contrast, to ensure a controlled level of variance in population, the mutation probability is kept small. We are especially interested in checking if this procedure is irrelevant to find quality fronts ($p_m = 0$). The combination between these parameters produces 12 possible configurations. As a result the NSGA-II method is run 12 times for each test case described in Section 6.6.1.

Table 9 – Calibration parameters of NSGA-II.

Parameter	**Values**
Population size (N)	24, 36, 42, 60
Crossover probability (p_c)	0.9
Mutation probability (p_m)	0, 0.1, 0.2

6.6.3 Performance measure

As a performance metric we chosen the additive binary ϵ-Indicator($I_{\epsilon+}$) proposed in (ZITZLER et al., 2003). Since the Pareto optimal front is known for under 9 cities' test cases, our goal on calibrating the NSGA-II heuristic consists of finding the configuration capable of reaching the lowest ϵ-Indicator value in less than 1 minute. Let our minimization problem have m objectives and let A and B be approximated fronts to be compared. A vector $u = (u_1, ..., u_m)$ is said to ϵ-dominate another vector $v = (v_1, ..., v_m)$, denoted as $u \succeq_{\epsilon+} v$, if and only if $\forall 1 \leq i \leq m : u_i \leq \epsilon + v_i$ for a given $\epsilon > 0$. Hence, the binary ϵ-Indicator can be defined as

$$I_{\epsilon+}(A, B) = \inf_{\epsilon \in \mathbb{R}} \{\forall v \in B \quad \exists \quad u \in A : u \succeq_{\epsilon+} v\}$$

According to Zitzler et al. (2003), the relation $u \succeq v$ means that the vector u weakly dominates the vector v, i.e., u is not worse than v in all objectives, what implies that in some cases u and v can be equal. Therefore, a vector u ϵ-dominates another vector v if after adding each objective value in v a factor of ϵ, the resulting objective vector is still weakly dominated by u. The additive binary ϵ-indicator $I_{\epsilon+}(A, B)$ can be calculated as follows:

$$I_{\epsilon+}(A, B) = \max_{v \in B} \min_{u \in A} \max_{1 \leq i \leq m} u_i - v_i$$

In order to evaluate several fronts at once against the optimal Pareto front R, we create a set F of approximated Pareto fronts. Next, to get a more regular ϵ-Indicator value, we normalized the values of all fronts $f \in F \cup R$. Firstly, we defined the biggest and the smallest values for each objective considering the solution vectors of all fronts in $F \cup R$. The biggest values are stored in vector $b = (b_1, ..., b_m)$ and the smallest values in vector $s = (s_1, ..., s_m)$. The normalized additive binary ϵ-Indicator is calculated as:

$$I_{\epsilon+}(f, R) = \max_{v \in R} \min_{u \in f} \max_{1 \leq i \leq m} \frac{u_i - s_i}{b_i - s_i} - \frac{v_i - s_1}{b_i - s_i} \forall f \in F$$

Figure 38-*b*) illustrates how the original relation between fronts $F1$ and $F4$ of Figure 36 has to be modified so that $F4$ ϵ-dominates $F1$. For all $u \in F1$ we added $I_{\epsilon+}(F4, F1) = 0.49$ on all objectives of vector u so that $F4 \succeq F1$, i.e., every $v \in F1$ is weakly dominated by at least one $u \in F4$.

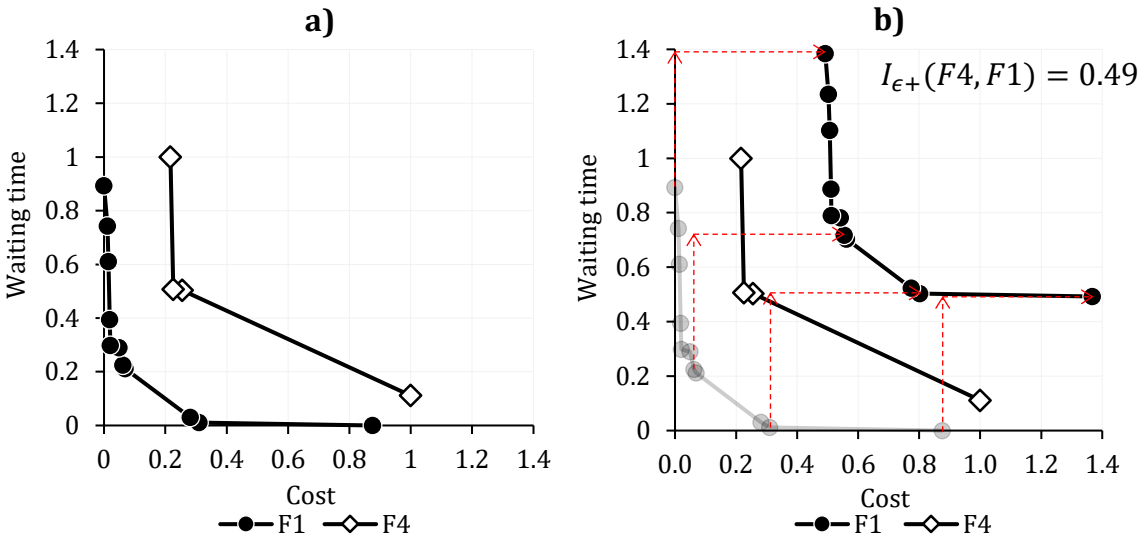

Figure 38 – Application of additive ϵ-Indicator on normalized fronts $F1$ and $F4$ of Figure 36. In *a*) we show the original relation between fronts $F1$ and $F4$ and in *b*) we show front $F1$ shifted by a value $I_{\epsilon+}(F4, F1) = 0.49$. The red arrows are 0.49 long and help to identify the shift.

6.7 Results and discussions

The following sections present the results we achieved after applying and executing the solving strategies of Section 6.5. Section 6.7.1 presents the performance of the brute-force method and discuss the efficacy of our threefold evaluation method. In turn, Section 6.7.2 investigates the best NSGA-II configuration depending on the amount of destinations.

6.7.1 Brute force method

The brute force method took about 3m, 30m and 4h46m to reach the Pareto fronts for test cases with 6, 7 and 8 cities respectively. Table 10 shows the execution time details. In contrast with the single objective implementation, besides calculating the routes, the bi-objective brute force method also needs to apply the nondomination sorting every time a new group of 24 permutations are created. This additional procedure causes a time overhead as big as the number of destinations involved in the process. Comparing with the single objective approach that took in average 3m06s, 18m30s and 3h08m to evaluate all possible solutions for 6, 7 and 8 cities, the bi-objective approach took further 30s, 12m and 1h38m only to manage the fronts'

creation. Considering a single permutation takes in average 0.42s to be evaluated (when 8 cites are concerned), if we had considered the 9 cities test cases it would take approximately 43h of processing time to discovery the route of a single test case. This expressive delay was the main motivation why we kept the test cases below 9 cities.

Table 10 – Average execution time of the bi-objective brute force approach per dwelling time and number of destinations.

	Dwelling time (days)				
#Cities	**1**	**2**	**3**	**4**	**Average**
6	0:03:38	0:03:34	0:03:30	0:03:28	0:03:32
7	0:32:54	0:31:17	0:29:56	0:28:45	0:30:43
8	5:21:03	4:55:14	4:34:10	4:16:51	4:46:50

In Figure 39 we plotted all optimal Pareto fronts found for the 180 test cases. As explained in Section 6.5.1, a front may be comprised of solutions created from 3 different shortest path strategies. Indeed, every front's solution also stores what objective was used to create the travel itinerary and when we differentiate the solutions by their objectives, a clusterization phenomenon can be noticed. Since clusters C1 and C3 where built using only a single objective (cost and waiting time respectively), their solutions occupy opposite areas of the complete set of fronts. In contrast, the cluster C2 encompasses the intermediate area between C1 and C3, once its solutions are generated through a balanced function that encompasses cost and waiting time. Therefore, the weights we set in Section 6.2 to establish a balanced value were effective in finding a middle ground between these two opposite objectives. In Figure 40, we provide a closer look in the fronts created for the test cases concerning 3 dwelling days. Once we separated the fronts by number of destinations, it is possible to notice a layer formation process in which, the smallest is the number of destinations, the cheaper is the cost and the shorter is the waiting time. In turn, Figure 41 shows all the fronts created for test cases concerning an 8 destination trip. Again, there is a clear layer formation process of the solutions, depending on the amount of dwelling days. In both cases there is an overlap area of solutions what suggests that, from the point of view of the decision making process, there might be some occasions in which searching for relaxed inputs may lead to fruitful solutions. Take for example the situation of a tourist who is planning a route that does not necessarily need to comprise all the destinations, i.e., the possible addition of some cities in the route is a bonus. The travel planner could include in the results solutions with similar characteristics (cost and waiting) but that somehow enhance the tourist's itinerary either by staying more days in a city or visiting bonus destinations.

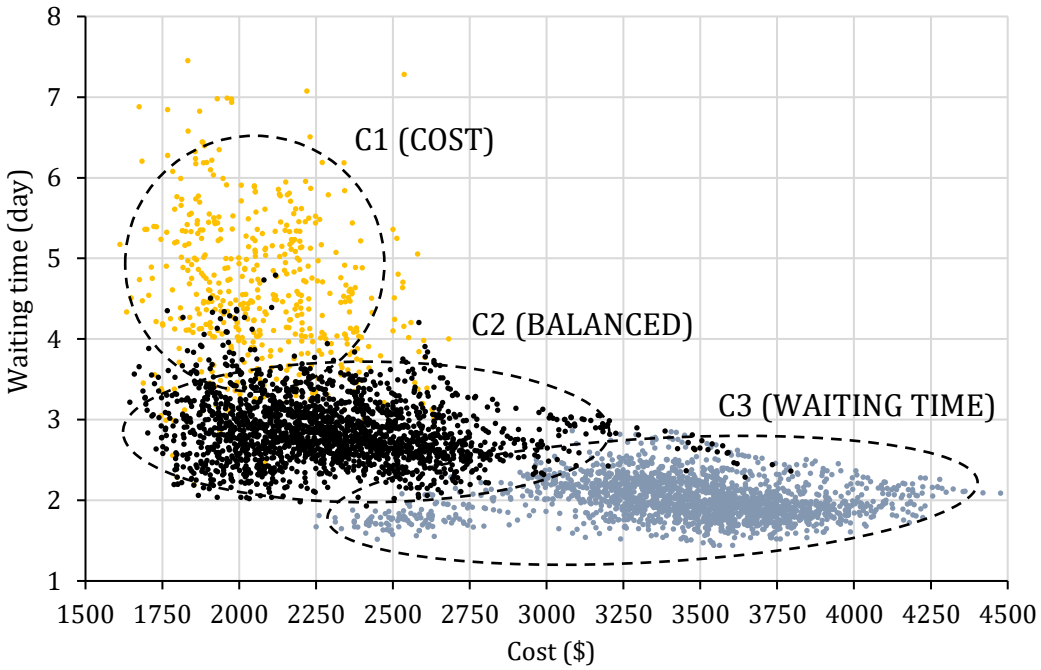

Figure 39 – Optimal Pareto fronts of the 180 test cases for 6, 7 and 8 cities and 1, 2, 3 and 4 visit days. Solutions are differentiated by the objective used to reach them (cost, waiting or balanced approach).

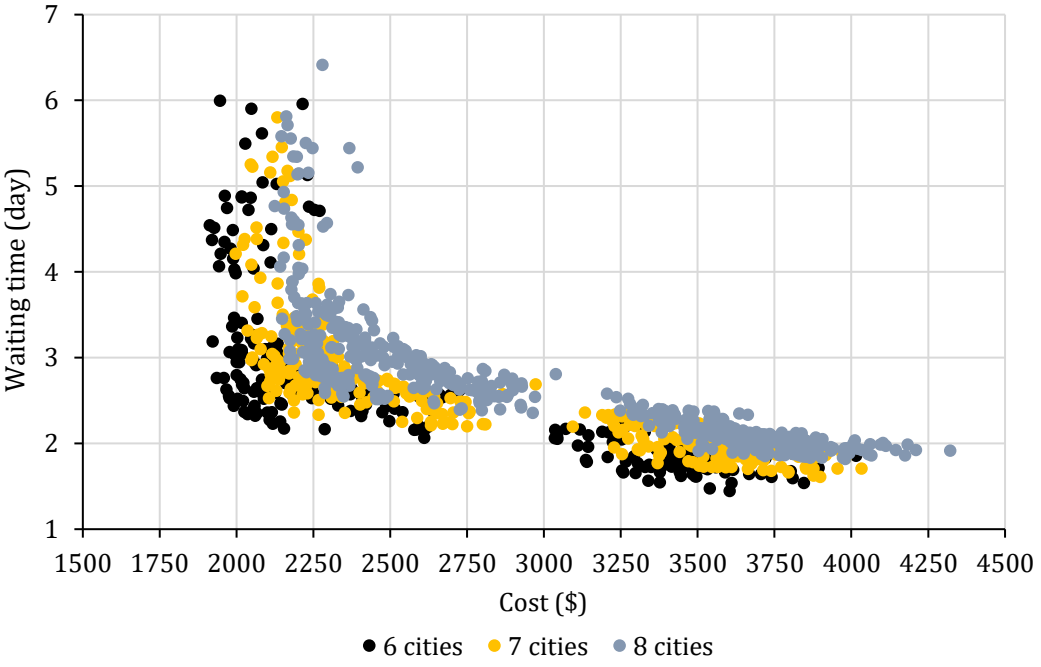

Figure 40 – Optimal Pareto fronts of all test cases comprising 3 dwelling days. There is a layer formation process due to different number of destinations.

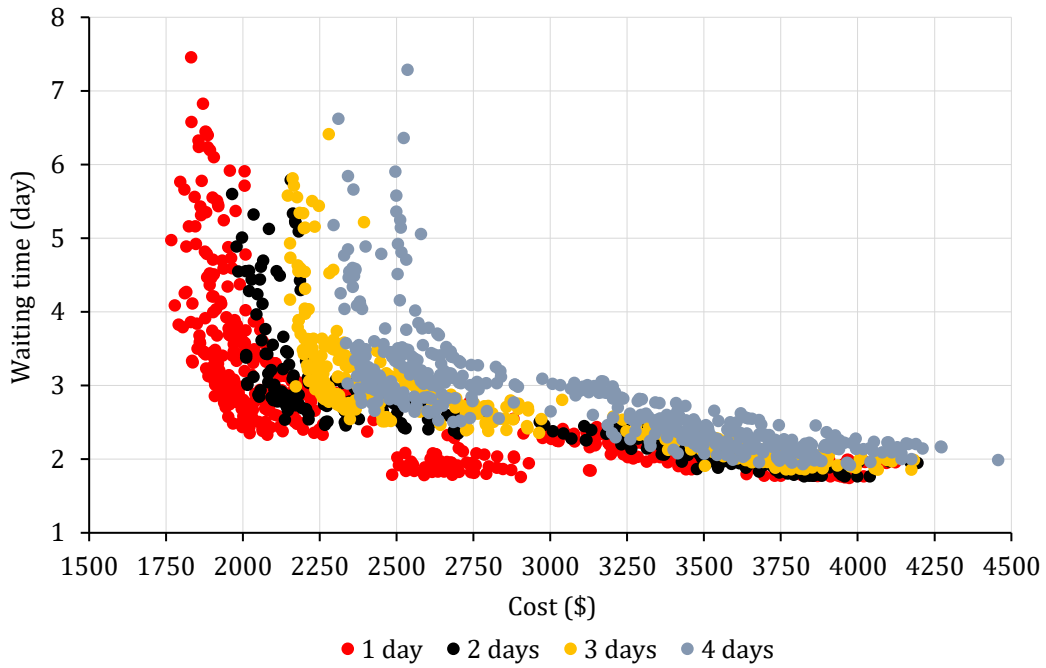

Figure 41 – Optimal Pareto fronts of all test cases comprising 8 destination cities. There is a layer formation process due to different amounts of dwelling times.

6.7.2 NSGA-II

Table 11 summarizes the results of the NSGA-II configurations we tested for each one of the 180 instances. The first column of results compiles the performance indicator values for each configuration based on the optimal results achieved by the brute force method. In turn, the second column shows the average size of the memory set, i.e., the number of different permutations each configuration was able to explore. In order to ease the analysis, the smallest ϵ-indicator values and the biggest memory set's sizes for each amount of cities are highlighted in boldface. Based on these values, the best configurations to calibrate the GA algorithm for 6, 7 and 8 destinations are (Mut.:0.1;Pop.:60), (Mut.:0.0;Pop.:48) and (Mut.:0.0;Pop.:36) respectively. Figures 42, 43 and 44 emphasize the performance differences of the configurations for each amount of cities, so that we can further evaluate the relation among the GA parameters. When a population of size 24 is considered, for example, there is a noticeable improvement in the average quality as the mutation rate gets higher. This relation confirms the idea that a small population may not have enough chromosomal material to produce high quality individuals over the generations. As the size of the populations increases, the method does not need to rely necessarily on mutation to produce new individuals since the mating process has a bigger pool of options to work on. In fact, the mutation process is even ineffective for our 7 and 8 cities instances.

Table 11 – Average results for the performance metric and for the size of the memory set considering all available configurations of Section 6.6.2.

		Additive ϵ-Indicator			**Size of memory set**		
Pop.	**Mut.**	**6 cites**	**7 cites**	**8 cites**	**6 cites**	**7 cites**	**8 cites**
24	0.0	0.04434	0.06755	0.08961	260.15	242.16	218.54
	0.1	0.03785	0.06399	0.08519	267.45	258.21	229.19
	0.2	0.03444	0.06290	0.08205	297.80	270.46	238.26
36	0.0	0.03733	0.06028	**0.08017**	315.74	285.29	250.70
	0.1	0.03083	0.05886	0.08492	325.13	295.53	257.75
	0.2	0.03088	0.05852	0.08426	348.80	301.23	262.22
48	0.0	0.03272	**0.05768**	0.08289	342.91	307.54	266.25
	0.1	0.02978	0.05859	0.08033	358.68	310.71	271.04
	0.2	0.02764	0.05962	0.08257	376.46	317.91	273.65
60	0.0	0.02906	0.05769	0.08514	360.72	320.25	276.66
	0.1	**0.02552**	0.06040	0.08312	378.85	324.13	283.43
	0.2	0.02696	0.05927	0.08658	**393.48**	**328.76**	**284.34**

With respect to the size of the memory set, no matter the amount of destinations, the configuration (Mut.:0.2;Pop.:60) was able to reach the biggest number of distinct solutions. However, although this configuration is able to increase solution variability by combining the biggest population size and the highest mutation rate, this variability does not necessarily translate into actual quality: none of the

best performance indicators were produced by the configuration (Mut.:0.2;Pop.:60). Nevertheless, in general, a heuristic approach is prone to find good approximations as far as it is able to properly explore higher number of different permutations. However, this number tends to decrease as the amount of destinations grows. With 6 cities for example, the best configuration was able to check in average 378.85 permutations, which corresponds to about 52.62% of the total number of existing permutations (6!=720). In contrast, the best configurations for 7 and 8 cities were able to evaluate in average 307.54 and 250.70 solutions each, only 6.10% and 0.62% of the total number of permutations for these numbers of destinations (7!=5,040 and 8!=40,320). The performance is also affected by an increase in the amount of destinations. For example, while the worst performance value of the 6 cities instances is 0.04434 (configuration (Mut.:0.0;Pop.:24)), the best value of the 7 cities instances is 0.05768 (configuration (Mut.:0.0;Pop.:48)). In both cases, this inversely proportional relation has to do with the complexity of the travel network: as pointed out in Section 5.4.1, the higher the number of cities the longer it takes to build a travel graph due to the increase in the number of nodes and arcs.

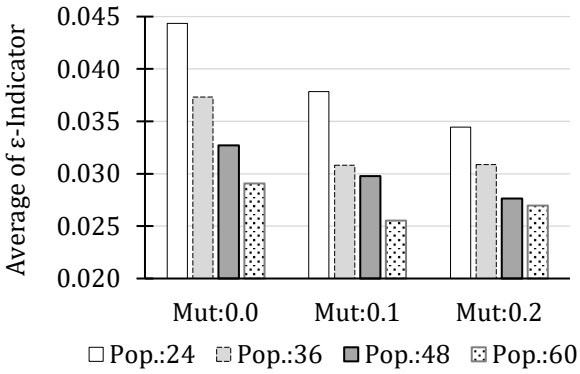

Figure 42 – Average of ϵ-Indicator considering the GA parameters for 6 cities instances. Best results are reached by configuration (Mut:0.1;Pop.:60).

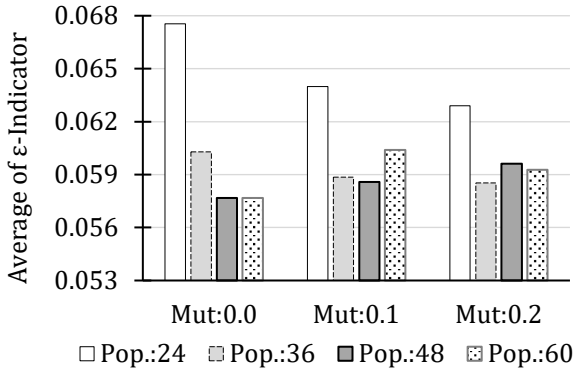

Figure 43 – Average of ϵ-Indicator considering the GA parameters for 7 cities instances. Best results are reached by configuration (Mut:0.0;Pop.:48).

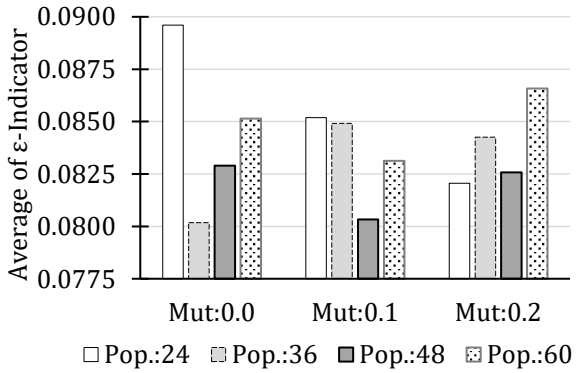

Figure 44 – Average of ϵ-Indicator considering the GA parameters for 8 cities instances. Best results are reached by configuration (Mut:0.0;Pop.:36).

7 Conclusions and future research

In this study we proposed a single-objective and a bi-objective approach for a realistic version of the Travel Planning Problem. Regardless of the approach, our main concern was to create relevant strategies that could be used in a real world travel planner application that often considers a vast amount of travel data, besides having a pressing execution time limitation. For each day of a 41 days travel window, we automatically collected from Google server all flight data involved in all trips concerning every possible combination of city pairs previously determined set of destination cities, and manually collected the accommodation costs of dwelling in a city for each day of a 41 days travel window. In contrast with popular travel planners that rely on the route provided by their customers to search for the best deals, our approach considers the case in which the sequence of visits is irrelevant: given a set of destinations and journey preferences, the traveler wants to find the permutation that provides the best itinerary. Our core strategy is based on heuristically generating promising sequences of visits, which are then transformed into time-dependent travel networks adapted to hold realistic travel data. For each network, there is a single optimal itinerary constructed out of the application of the classical Dijkstra shortest path algorithm. To deal with the delay related with the building of these networks, we employed a parallel strategy that was able to successfully speedup this solution construction phase. Additionally, we conduced an analysis on the impact of the journey preferences on the structure of the travel networks. In general, the larger is the set of destinations and the smaller is the period of visit, the more complex is a travel network, and the longer is its construction time.

Our first version of the problem was a single-objective approach in which the traveler's most important goal was to create a budget itinerary, i.e., minimize the travel's total cost. To better understand the specifications of this particular version, we created an integer programming formulation and to solve the problem to its optimality we used a parallel brute force algorithm. The preliminary results provided by this algorithm were important to determine a viable excerpt of our set of test cases. In fact, it is not necessary to use a heuristic approach for instances containing less than 6 cities, once they can be optimally determined in less than 1 minute, and it is impractical to get optimal values for over 9 cities, once it would take more than 30 hours. Consequently, we setup a benchmark involving 280 test cases containing from 6 to 9 cities. Our single-objective heuristic strategy to deal with this set of test cases was an ILS (*Iterated Local Search*) method. In this strategy we slightly change the objective function to not completely ignore the time component by applying monetary penalties on waitings. For each test case, the method started with a greedy solution, applied a parallel implementation of the 2-Opt local search

to explore promising solutions and a Double Bridge perturbation to escape from local optima. We tested two versions of the ILS method, the first was a sequential implementation in which we varied the number of iterations without improvement to finish the method, and the second was a parallel implementation in which we varied the number of perturbations performed at a time. In total there were 9 possible configurations and we executed all our test cases considering each one of them. Since we defined the solving method should run in less than 1min we eliminated all configurations that had larger execution times than this limitation. Among the remaining configurations we selected the best option for each travel length. For 6, 7, 8 and 9 destinations, the methods chosen were able to reach solutions in average only 0.3%, 1.1%, 2.7% and 4.1% divergent from the optimal values reached by the brute force method.

When cost is the only optimization component, the single way a traveler can engage on using its subjectiveness to choose a convenient route would be exhibiting a list of traveling options ordered by price. A tourist who wants also to optimize its time spending would have some difficulties on trying to find an itinerary that comprises an adequate balance among price and waiting. To enhance the decision making process, our second version of the problem also included the time component in the optimization equation. After introducing the concept of Pareto optimality, the base structure our bi-objective solution is built over, we explained how to separate a set of solutions into Pareto fronts. Based on the exact results of the single-objective version and in the impracticability of executing the bi-objective brute force algorithm for more than 8 cities, we shortened the set of test cases to comprise from 6 to 8 cities. As a result, we produced 180 optimal Pareto fronts to serve as a reference to our heuristic approach. Regardless the strategy chosen, exact or heuristic, each route was built in parallel over a time dependent network and had its shortest path determined regarding 3 objectives: cost, waiting time and a balanced value. Although we have chosen to apply a multi-objective strategy only on route determination, this threefold route evaluation was able to successfully fill the space of solutions. We applied the NSGA-II framework to reach a set of quality detailed travel itineraries considering the time limitation of 1 minute. The evolutionary approach was used to create and evolve a population of routes. Regarding the calibration of the genetic algorithm, there is not a single optimization setup able to improve all test cases: each number of destinations demands a particular configuration of parameters. Choosing the best setup for each case guarantees that an approximated solution is, in average, only 2.6%, 5.8% and 8.0% divergent from

the exact implementation for 6, 7, and 8 cities test cases respectively.

7.1 Future research

Besides decreasing the execution time of the process as whole and implementing new test cases, possibly with a bigger amount of cities and heterogeneous dwelling times, the future work on the travel planning problem may be focused on:

Enhancing the traveler's experience: Firstly, regarding the flights, the traveler could decide the earliest takeoff time and the latest landing time so that all possible solutions would only include convenient times. Furthermore, the current approach does not account for possible appointments the tourist may need to commit throughout the route. Take for example the case of a scholar who must travel to attend a congress overseas and wants to take the opportunity to visit cities nearby, or the case of a tourist who wants to visit several cities but also intends to go to a concert in a specific place, date and time. Both situations require an additional restriction to the model so that it can correctly schedule a visit to one or more cities considering the position of their events throughout the time window.

Relaxing the input data: As presented in Section 6.7.1, many solutions with different number of destinations and dwelling times occupy nearly the same area of the solution space. A more sophisticated travel planner would make use of this fortunate circumstance to create broader solutions for the problem by providing further options for the traveler. Similar to some on-line travel planners that allow customers to search for flights considering flexible dates, we envisage a system that would also consider a flexible number of destinations. Our current approach is already able to handle this flexibility in the dwelling time once we define minimum and maximum stay durations. This idea could be adapted when determining the set of destinations: the traveler would choose a set of cities that must be visited, and a set of bonus cities that can possibly be included in the route. The exact cost a customer is prone to pay to visit a bonus city will be part of the decision making process, once the planner would also include relaxed solutions to the final answer.

Including accommodations policies: Our current approach simply assign a price of stay for each date in a city, based on the cheapest accommodations found on-line. However, in some occasions the cheapest option is not available during all the period of stay so that the customer needs to decide if it is better to book only the unavailable days or the complete period in a different accommodation. Solutions showing both alternatives should be presented so that the traveler can once again decide what is the best option.

7.2 Publications

- BEIRIGO, B.; SANTOS, A. Application of NSGA-II framework to the travel planning problem using real-world travel data. In: IEEE. *2016 IEEE Congress on Evolutionary Computation (CEC)*. [S.l.], 2016.

- BEIRIGO, B.; SANTOS, A. A parallel heuristic for the travel planning problem. In: IEEE. *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*.[S.l.], 2015. p. 283-288.

References

ABELEDO, H. et al. The time dependent traveling salesman problem: polyhedra and algorithm. *Mathematical Programming Computation*, Springer, v. 5, n. 1, p. 27–55, 2013.

BELLMAN, R. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, ACM, v. 9, n. 1, p. 61–63, 1962.

BÉRUBÉ, J.-F.; POTVIN, J.-Y.; VAUCHER, J. Time-dependent shortest paths through a fixed sequence of nodes: application to a travel planning problem. *Computers & operations research*, Elsevier, v. 33, n. 6, p. 1838–1856, 2006.

BROWNLEE, J. *Clever algorithms: nature-inspired programming recipes*. [S.l.]: Jason Brownlee, 2011.

CAMACHO, D.; BORRAJO, D.; MOLINA, J. M. Intelligent travel planning: a multiagent planning system to solve web problems in the e-tourism domain. *Autonomous agents and multi-agent systems*, Springer, v. 4, n. 4, p. 387–392, 2001.

CHATTERJEE, S.; CARRERA, C.; LYNCH, L. A. Genetic algorithms and traveling salesman problems. *European journal of operational research*, Elsevier, v. 93, n. 3, p. 490–510, 1996.

CHEN, Z.; SHEN, H. T.; ZHOU, X. Discovering popular routes from trajectories. In: IEEE. *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. [S.l.], 2011. p. 900–911.

CHOUDHURY, M. D. et al. Automatic construction of travel itineraries using social breadcrumbs. In: ACM. *Proceedings of the 21st ACM conference on Hypertext and hypermedia*. [S.l.], 2010. p. 35–44.

CORMEN, T. H. *Introduction to algorithms*. [S.l.]: MIT press, 2009.

DANTZIG, G.; FULKERSON, R.; JOHNSON, S. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, INFORMS, v. 2, n. 4, p. 393–410, 1954.

DEB, K. et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 6, n. 2, p. 182–197, 2002.

DESROCHERS, M.; LENSTRA, J. K.; SAVELSBERGH, M. W. A classification scheme for vehicle routing and scheduling problems. *European Journal of Operational Research*, Elsevier, v. 46, n. 3, p. 322–332, 1990.

DIVSALAR, A. et al. A memetic algorithm for the orienteering problem with hotel selection. *European Journal of Operational Research*, Elsevier, v. 237, n. 1, p. 29–49, 2014.

DORIGO, M.; GAMBARDELLA, L. M. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, IEEE, v. 1, n. 1, p. 53–66, 1997.

ECMA. *The JSON Data Interchange Format*. 2013. [Online; accessed 10-July-2016]. Disponível em: ⟨http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf⟩.

ER, H. R.; ERDOGAN, N. Parallel genetic algorithm to solve traveling salesman problem on mapreduce framework using hadoop cluster. *arXiv preprint arXiv:1401.6267*, 2014.

GARCIA, A. et al. Integrating public transportation in personalised electronic tourist guides. *Computers & Operations Research*, Elsevier, v. 40, n. 3, p. 758–774, 2013.

GAVALAS, D. et al. A survey on algorithmic approaches for solving tourist trip design problems. *Journal of Heuristics*, Springer, v. 20, n. 3, p. 291–328, 2014.

GEERTS, W. *Top 100 city destinations ranking - Euromonitor International*. 2016. [Online; accessed 9-July-2016]. Disponível em: ⟨http://go.euromonitor.com/rs/805-KOK-719/images/TCD-presentation_FINAL.pdf⟩.

GENDREAU, M.; GHIANI, G.; GUERRIERO, E. Time-dependent routing problems: A review. *Computers & Operations Research*, Elsevier, v. 64, p. 189–197, 2015.

GENDREAU, M.; LAPORTE, G.; SEMET, F. A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks*, v. 32, n. 4, p. 263–273, 1998.

GOOGLE. *General FAQs*. 2016. [Online; accessed 7-July-2016]. Disponível em: ⟨https://developers.google.com/qpx-express/faq⟩.

GUNAWAN, A.; LAU, H. C.; VANSTEENWEGEN, P. Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, Elsevier, 2016.

GUTIN, G.; PUNNEN, A. P. *The traveling salesman problem and its variations*. [S.l.]: Springer Science & Business Media, 2006. v. 12.

HYDE, K. et al. New perspectives on vacation decision making. *International Journal of Culture, Tourism and Hospitality Research*, Emerald Group Publishing Limited, v. 5, n. 2, p. 103–111, 2011.

IATA. *About us*. 2016. [Online; accessed 9-July-2016]. Disponível em: ⟨http://www.iata.org/about/pages/index.aspx⟩.

KANG, S. et al. Gpu-based parallel genetic approach to large-scale travelling salesman problem. *The Journal of Supercomputing*, Springer, p. 1–16, 2016.

KRACHT, J.; WANG, Y. Examining the tourism distribution channel: evolution and transformation. *International Journal of Contemporary Hospitality Management*, Emerald Group publishing limited, v. 22, n. 5, p. 736–757, 2010.

LAPORTE, G. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, Elsevier, v. 59, n. 2, p. 231–247, 1992.

LEIFER, A. C.; ROSENWEIN, M. B. Strong linear programming relaxations for the orienteering problem. *European Journal of Operational Research*, Elsevier, v. 73, n. 3, p. 517–523, 1994.

LENSTRA, J. K. *Local search in combinatorial optimization*. [S.l.]: Princeton University Press, 1997.

LI, X.; ZHOU, J.; ZHAO, X. Travel itinerary problem. *Transportation Research Part B: Methodological*, Elsevier, v. 91, p. 332–343, 2016.

LIN, S.-W.; VINCENT, F. Y. A simulated annealing heuristic for the multiconstraint team orienteering problem with multiple time windows. *Applied Soft Computing*, Elsevier, v. 37, p. 632–642, 2015.

LITTLE, J. D. et al. An algorithm for the traveling salesman problem. *Operations research*, INFORMS, v. 11, n. 6, p. 972–989, 1963.

LU, X. et al. Photo2trip: generating travel routes from geo-tagged photos for trip planning. In: ACM. *Proceedings of the international conference on Multimedia*. [S.l.], 2010. p. 143–152.

LUST, T.; TEGHEM, J. The multiobjective traveling salesman problem: a survey and a new approach. In: *Advances in Multi-Objective Nature Inspired Computing*. [S.l.]: Springer, 2010. p. 119–141.

MEI, Y.; SALIM, F. D.; LI, X. Efficient meta-heuristics for the multi-objective time-dependent orienteering problem. *European Journal of Operational Research*, Elsevier, v. 254, n. 2, p. 443–457, 2016.

PEPPER, J. W.; GOLDEN, B. L.; WASIL, E. A. Solving the traveling salesman problem with annealing-based heuristics: a computational study. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, IEEE, v. 32, n. 1, p. 72–77, 2002.

PICARD, J.-C.; QUEYRANNE, M. The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. *Operations Research*, INFORMS, v. 26, n. 1, p. 86–110, 1978.

POGGI, M.; VIANA, H.; UCHOA, E. The team orienteering problem: Formulations and branch-cut and price. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *OASIcs-OpenAccess Series in Informatics*. [S.l.], 2010. v. 14.

RIGHINI, G.; SALANI, M. Dynamic programming for the orienteering problem with time windows. *Note del Polo-Ricerca*, Citeseer, v. 91, 2006.

SYLEJMANI, K.; DIKA, A. Solving touristic trip planning problem by using taboo search approach. *International Journal of Computer Science Issues (IJCSI)*, v. 8, n. 5, 2011.

TALBI, E.-G. *Metaheuristics: from design to implementation*. [S.l.]: John Wiley & Sons, 2009. v. 74.

TSILIGIRIDES, T. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, Springer, v. 35, n. 9, p. 797–809, 1984.

VANSTEENWEGEN, P.; OUDHEUSDEN, D. V. The mobile tourist guide: an or opportunity. *OR insight*, Springer, v. 20, n. 3, p. 21–27, 2007.

VANSTEENWEGEN, P. et al. Iterated local search for the team orienteering problem with time windows. *Computers & Operations Research*, Elsevier, v. 36, n. 12, p. 3281–3290, 2009.

VERHOEVEN, M.; AARTS, E. H.; SWINKELS, P. A parallel 2-opt algorithm for the traveling salesman problem. *Future Generation Computer Systems*, Elsevier, v. 11, n. 2, p. 175–182, 1995.

WIEL, R. J. V.; SAHINIDIS, N. V. Heuristic bounds and test problem generation for the time-dependent traveling salesman problem. *Transportation Science*, INFORMS, v. 29, n. 2, p. 167–183, 1995.

XIANG, Z. et al. Adapting to the internet: trends in travelers' use of the web for trip planning. *Journal of Travel Research*, Sage Publications, p. 0047287514522883, 2014.

ZALATAN, A. The determinants of planning time in vacation travel. *Tourism Management*, Elsevier, v. 17, n. 2, p. 123–131, 1996.

ZHENG, Y. et al. Mining interesting locations and travel sequences from gps trajectories. In: ACM. *Proceedings of the 18th international conference on World wide web*. [S.l.], 2009. p. 791–800.

ZHOU, A. et al. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, Elsevier, v. 1, n. 1, p. 32–49, 2011.

ZITZLER, E. et al. Performance assessment of multiobjective optimizers: an analysis and review. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 7, n. 2, p. 117–132, 2003.

A Google QPX Express API response

In this appendix, we present the main aspects of the JSON output related to Solution 1 of Figure 9. The complete JSON structure of this solution is presented in Figure 49. The `"data"` attribute stores all information needed to build the available solutions, such as airports (Figure 45), cities (Figure 46), aircrafts (Figure 47), taxes (Figure 48) and carriers (Figure 49).

A1

```
"kind": "qpxexpress#airportData",
"code": "GIG",
"city": "RIO",
"name": "Rio de Janeiro Galeao - A. C. Jobim International"
```

A2

```
"kind": "qpxexpress#airportData",
"code": "GRU",
"city": "SAO",
"name": "Sao Paulo Guarulhos International"
```

A3

```
"kind": "qpxexpress#airportData",
"code": "CMN",
"city": "CAS",
"name": "Casablanca Mohammed V"
```

A4

```
"kind": "qpxexpress#airportData",
"code": "AMS",
"city": "AMS",
"name": "Amsterdam Schiphol Airport"
```

Figure 45 – JSON objects A1, A2, A3 and A4 containing the names and cities of the airports that are part of Solution 1 of Figure 9.

C1

```
"kind": "qpxexpress#cityData",
"code": "RIO",
"name": "Rio de Janeiro"
```

C2

```
"kind": "qpxexpress#cityData",
"code": "Sao",
"name": "Sao Paulo"
```

C3

```
"kind": "qpxexpress#cityData",
"code": "CAS",
"name": "Casablanca"
```

C4

```
"kind": "qpxexpress#cityData",
"code": "AMS",
"name": "Amsterdam"
```

Figure 46 – JSON objects C1, C2, C3 and C4 containing the names of the cities involved in Solution 1 of Figure 9.

R1

```
"kind": "qpxexpress#aircraftData",
"code": "767",
"name": "Boeing 767"
```

R2

```
"kind": "qpxexpress#aircraftData",
"code": "738",
"name": "Boeing 737"
```

Figure 47 – JSON objects R1 and R2 containing the name of the aircrafts that are flying between the points of Solution 1 of Figure 9.

X1
```
"kind": "qpxexpress#taxData",
"id": "BR_003",
"name": "Brazil Embarkation
Fee"
```

X2
```
"kind": "qpxexpress#taxData",
"id": "MA_012",
"name": "Morocco Airport
Security Tax"
```

X3
```
"kind": "qpxexpress#taxData",
"id": "MA_001",
"name": "Morocco Passenger
Service Charge CMN"
```

Figure 48 – JSON objects X1, X2 and X3 containing the names of the taxes due for flying between an origin and a destination of Solution 1 of Figure 9.

```
{
  "kind": "qpxExpress#tripsSearch",
  "trips": {
  "kind": "qpxexpress#tripOptions",
  "requestId": "iLyPdyNV81N6mHHoF0OhH3",
  "data": {
    "kind": "qpxexpress#data",
    "airport": [{A1}, {A2}, {A3}, {A3}],
    "city": [{C1}, {C2}, {C3}, {C4}],
    "aircraft": [{R1}, {R1}],
    "tax": [{X1}, {X2}, {X3}],
    "carrier": [{
      "kind": "qpxexpress#carrierData",
      "code": "AT",
      "name": "Royal Air Maroc"
  }]
  },
  "tripOption": [{O1}]
  }
}
```

Figure 49 – Structure of a complete JSON solution for the request exhibited in Figure 8. We only show the objects involved in Solution 1. The airports A1, A2, A3 and A4 of attribute `"airport"` are showed in Figure 45; the cities C1, C2, C3 and C4 of attribute `"city"` are exhibited in Figure 46; the aircrafts R1 and R2 of attribute `aircraft` are showed in Figure 47; the taxes X1, X2 and X3 are showed in Figure 48. Finally, the attribute `"tripOption"` stores a list of trips available on QPX server, in this particular case only with trip O1 presented in Figure 50.

In turn, the `"tripOption"` attribute stores a list of priced itinerary solutions to the QPX Express query. In Figure 50 we show the content of a single trip option corresponding to the Solution 1. Initially it presents the total cost of the trip (`"saleTotal"` field) in the currency chosen at the travel's request and its unique identifier (`"id"` field). Next, there is a `"slice"` attribute which is a list of slices composing this trip. A slice consists in a traveler's intent to get between two points:

one-way journeys, for instance, are generally expressed using one slice and round-trips using two. Each slice comprises a duration time in minutes and also an array of segments. In Figure 50 for example, the travel slice lasts for 1045 minutes and has two segments, S1 and S2 (Figure 51).


```
"kind": "qpxexpress#tripOption",
"saleTotal": "USD418.33",
"id": "b4t4y5teOKPRSKYSYZdVu0001",
"slice": [{
  "kind": "qpxexpress#sliceInfo",
  "duration": 1045,
  "segment": [{S1}, {S2}]
}],
"pricing": [{
  "kind": "qpxexpress#pricingInfo",
  "fare": [{F1}],
  "segmentPricing": [{P1}, {P2}],
  "baseFareTotal": "USD363.00",
  "saleFareTotal": "USD363.00",
  "saleTaxTotal": "USD55.33",
  "saleTotal": "USD418.33",
  "passengers": {
    "kind": "qpxexpress#passengerCounts",
    "adultCount": 1
  },
  "tax": [{T1}, {T2}, {T3}],
  "fareCalculation": "RIO AT X/CAS AT AMS 363.00XA0W0BRA
                      NUC 363.00 END ROE 1.00  FARE  USD
                      363.00 XT 36.73BR 18.60MA",
  "latestTicketingTime": "2016-07-13T17:24-04:00",
  "ptc": "ADT"
}]                                                    01
```


Figure 50 – Formatted response stressing the most important features of a solution.

Figure 51 – Segments S1 and S2 that compose the trip option 1 showed in Figure 9 from Rio de Janeiro to Amsterdam. Segment S1 is composed by legs L1 and L2 (Figure 52) and segment S2 is composed by a unique leg L3 (Figure 52). There is a flight connection of 155 minutes between S1 and S2.

A segment is an operation of a flight with a single flight designator that can include any number of stops where passengers board and deplane the same aircraft, from origin to final destination. A flight designator includes an airline code, which has two letters or a number and a letter in combination, and a flight number of up to four digits. For example, the flight AT213 of segment S1 (Figure 51) from GIG to CMN has two legs, both operated by the aircraft 763: L1) (GIG,GRU); L2) (GRU, CMN).

A leg is the operation of an aircraft from one scheduled departure station to its next scheduled arrival station, i.e., a takeoff immediately followed by a landing. In Figure 52 we show the information related to the 3 legs that compose Solution 1: (GIG,GRU), (GRU,CMN) and (CMN,AMS)

```
"kind": "qpxexpress#legInfo",
"id": "LQsi67lpOTIu8z7G",
"aircraft": "763",
"arrivalTime": "2016-07-13T19:35-03:00",
"departureTime": "2016-07-13T18:25-03:00",
"origin": "GIG",
"destination": "GRU",
"originTerminal": "1",
"destinationTerminal": "2",
"duration": 70,
"mileage": 209,
"connectionDuration": 65
```
L1

```
"kind": "qpxexpress#legInfo",
"id": "L0mcsl6YNmWCCGuU",
"aircraft": "763",
"arrivalTime": "2016-07-14T09:50+01:00",
"departureTime": "2016-07-13T20:40-03:00",
"origin": "GRU",
"destination": "CMN",
"originTerminal": "2",
"destinationTerminal": "2",
"duration": 550,
"mileage": 4682,
"meal": "Breakfast"
```
L2

```
"kind": "qpxexpress#legInfo",
"id": "L5nhD8w2B-Gdv-HI",
"aircraft": "738",
"arrivalTime": "2016-07-14T16:50+02:00",
"departureTime": "2016-07-14T12:25+01:00",
"origin": "CMN",
"destination": "AMS",
"originTerminal": "2",
"duration": 205,
"mileage": 1445,
"meal": "Dinner"
```
L3

Figure 52 – Legs L1, L2 and L3 involved in the trip option 1 showed in Figure 9. Legs L1 and L2 are operated by the aircraft 763 and belong to the same travel segment S1 (Figure 51), besides having a 65 minutes flight connection. In turn, leg L3 is operated by the aircraft 738 and belong to the segment S2 (showed in Figure 51).

The `"pricing"` element of a trip option saves all data related to fares and taxes. The `"fare"` attribute consists in a list of fares used to price one or more segments. In this particular example, there is only the fare F1 showed in Figure 53.


```
"kind": "qpxexpress#fareInfo",
"id": "A95WknGJtU8cPZBoVG7wWX2s1EQf68x0gyR/y19JuiAM",
"carrier": "AT",
"origin": "RIO",
"destination": "AMS",
"basisCode": "XA0W0BR",
```
F1

Figure 53 – The fare object of the trip option presented in Figure 50. Besides a unique identifier, each fare has the 2-letter IATA airline designator of the carrier and the origin and destination codes to which the fare applies.

Next, there is a "segmentPricing" attribute that stores the list of pricing objects related to each segment. In Figure 54 we show the pricing objects P1 and P2 included in Solution 1. Although not presented in this particular example, the segment pricing object also hold information about the free baggage allowance covered by a fare as well as the details of this baggage such as weight and number of pieces. This detail is very important when exhibiting a final solution for a passenger once there may be additional costs related to luggage transportation.


```
"kind": "qpxexpress#segmentPricing",
"fareId": " A95WknGJtU8cPZBoVG7wWX2s1EQf68x0gyR/y19JuiAM",
"segmentId": "Gq70n0RQTsh42RxI"
```
P1

```
"kind": "qpxexpress#segmentPricing",
"fareId": " A95WknGJtU8cPZBoVG7wWX2s1EQf68x0gyR/y19JuiAM",
"segmentId": "GPVpE+bq-K3JZDp1"
```
P2

Figure 54 – Segment pricing objects P1 and P2 of the trip option presented in Figure 50. The segment ids of P1 and P2 refer to the segments S1 and S2 presented in Figure 51. In turn, both fare ids of P1 and P2 refer to the fare object F1 presented in Figure 53.

All taxes used to calculate the tax total per ticket are presented in Figure 55. Besides an identification code, each tax object provides information about what entity is charging the tax (government or carrier) and also the amount to be paid. A summary of all relevant items that go into the calculation of the total fare are presented in the field "fareCalculation" of a pricing object.


```
"kind": "qpxexpress#taxInfo",
"id": "BR_003",
"chargeType": "GOVERNMENT",
"code": "BR",
"country": "BR",
"salePrice": "USD36.73"
```
T1

```
"kind": "qpxexpress#taxInfo",
"id": "MA_012",
"chargeType": "GOVERNMENT",
"code": "MA",
"country": "MA",
"salePrice": "USD4.90"
```
T2

```
"kind": "qpxexpress#taxInfo",
"id": "MA_001",
"chargeType": "GOVERNMENT",
"code": "MA",
"country": "MA",
"salePrice": "USD13.70"
```
T3

Figure 55 – The taxes T1, T2 and T3 used to calculate the tax total per ticket (attribute "saleTaxTotal"). The names of each tax are presented in Figure 48.