TÉCNICO LISBOA

TODaaS

Transport On-Demand as a Service

João Pedro de Melo Pinheiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. Alberto Manuel Ramos da Cunha
Members of the Committee: Prof. Ernesto José Marques Morgado

June 2019

Acknowledgments

I would like to thank Professor Alberto Cunha for the support and guidance. Also, to my family, friends, colleagues, and beloved Timmy, who were always there when times were rough, my best thanks. Finally, thanks to Enlightenment.AI [1], especially to Miguel Marques, and Manuel Levi, whom experience added valuable ideas and insights.

To each and every one of you – Thank you, and best regards.

[1] Enlightenment.AI, Online: https://enlightenment.ai/, Last accessed: 05 April 2019

Abstract

The increasingly competitive market of mobility services in cities, together with European regulations are motivating Public Transport providers to offer Demand-Responsive Transport Services. In this work we describe this kind of mobility solution, comparing it with other known solutions. Then, we make a literature review on similar projects, and algorithmic approaches to accomplish a service like this. Finally, we present a software solution for a demand-responsive public transport service, with its requirements validations, and a small study on its routing algorithm, making use of some literature examples.

Keywords

Demand-Responsive Transport; Mobility; Software; VRP; DARP; Genetic Algorithms;

Contents

1 Introduction **1**

1.1 Service Description . . . 3
1.2 Service Flow . . . 4
1.3 Motivation . . . 4
1.3.1 Demand-responsiveness Advantages . . . 5
1.3.2 European Directive . . . 5
1.4 Scope . . . 6
1.5 Objectives . . . 7

2 State of Art **9**

2.1 Mobility Solutions . . . 11
2.1.1 Types of Mobility Solutions . . . 11
2.1.2 Electronic Hailing . . . 12
2.1.3 Public Transport . . . 12
2.1.4 Demand-Responsive Transport . . . 13
2.2 On-Demand Services . . . 14
2.2.1 Kutsuplus . . . 14
2.2.2 MOIA . . . 15
2.2.3 Via . . . 15
2.2.4 Discussion . . . 16
2.3 Vehicle Routing Problem . . . 16
2.3.1 VRP Variants . . . 17
2.3.2 Solving the VRP . . . 18
2.3.2.A Exact Methods . . . 19
2.3.2.B Heuristics . . . 19
2.3.2.C 2-Phase Algorithm . . . 19
2.3.2.D Constructive Methods . . . 20
2.3.2.E Metaheuristics . . . 20

2.3.2.F Discussion . . . 21
2.4 Genetic Algorithms . . . 22
2.4.1 Methodology . . . 22
2.4.2 Population . . . 22
2.4.3 Selection . . . 23
2.4.4 Mutation . . . 24
2.4.5 Crossover . . . 24
2.4.5.A Simple, Bad Crossover Operators . . . 24
2.4.5.B PMX Crossover Operator . . . 25
2.4.6 Local Search . . . 25
3 Solution Proposal **27**
3.1 Storing Requests . . . 30
3.2 Routes Calculation . . . 31
3.2.1 Reducing Trip Stops . . . 31
3.2.2 Inputs . . . 32
3.2.3 Outputs . . . 33
3.2.4 Algorithm . . . 33
3.3 Requests Life-cycle . . . 33
3.4 Finalizing the Request . . . 34
4 Technologic Arquitecture **35**
4.1 Resources . . . 38
4.1.1 Providers and Areas . . . 38
4.1.1.A Calculating intersections on an Ellipsoid . . . 39
4.1.2 Vehicles . . . 40
4.1.3 Drivers . . . 40
4.1.4 Availabilities . . . 40
4.1.5 Trackings . . . 41
4.1.6 API Endpoints . . . 41
4.2 Maps . . . 41
4.3 Algorithm . . . 42
4.3.1 Representation and Genetic Operators . . . 44
4.4 Requests . . . 45
4.4.1 Requests . . . 45
4.4.1.A Validation . . . 45
4.4.1.B Operations . . . 47

4.4.2 Solutions . . . 47
4.4.2.A Solvers, Job, and Solution generation . . . 47
4.4.3 Request States . . . 48
4.4.4 Ratings . . . 48
4.4.5 API Endpoints . . . 49
4.5 Users, Authentication and Authorization . . . 49
4.5.1 Users . . . 49
4.5.2 Authentication/Authorization . . . 50
5 Evaluation **51**
5.1 Validations: By Module and Integrated . . . 53
5.1.1 Maps . . . 53
5.1.2 Requests . . . 54
5.1.2.A POST /requests . . . 54
5.1.2.B GET /requests . . . 55
5.1.2.C PUT /requests/1 . . . 55
5.1.2.D PATCH /requests/1 . . . 55
5.1.2.E GET /requests/1 . . . 55
5.1.2.F DELETE /requests/1 . . . 56
5.1.2.G GET /requests/1/preview . . . 56
5.1.2.H POST /requests/1/rate . . . 56
5.1.2.I GET /requests/provider-area/2 . . . 56
5.1.3 Resources . . . 56
5.1.3.A Providers and Areas . . . 57
A – POST /providers . . . 57
B – GET /providers . . . 58
C – PUT /providers/1 . . . 58
D – GET /providers/1 . . . 58
E – DELETE /providers/1 . . . 58
5.1.3.B Vehicles . . . 58
A – POST /vehicles . . . 58
B – GET /vehicles . . . 58
C – PUT /vehicles/1 . . . 59
D – GET /vehicles/1 . . . 59
E – DELETE /vehicles/1 . . . 59
F – GET /vehicles/provider/2 . . . 59

G – DELETE /vehicles/provider/2 . . . 59
5.1.3.C Drivers . . . 59
5.1.3.D Availabilities . . . 59
A – POST /availabilities . . . 59
B – PUT /availabilities . . . 60
C – GET /availabilities . . . 60
D – DELETE /availabilities . . . 60
5.1.3.E Trackings . . . 60
A – POST /trackings . . . 61
B – PUT /trackings . . . 61
C – GET /trackings . . . 61
D – DELETE /trackings . . . 61
5.1.4 Algorithm . . . 61
5.1.5 Integration . . . 62
5.2 Service Behaviour . . . 63
5.2.1 DARP vs. OVRP . . . 63
5.2.2 Lisbon Metro vs. DRT Service . . . 65
5.3 Discussion . . . 66
6 Conclusion **67**
6.1 Conclusions . . . 69
6.2 System Limitations and Future Work . . . 70
6.2.1 New Features . . . 70
6.2.2 Algorithm Improvements . . . 71
6.2.3 New Studies . . . 71
References **73**
A Tan et al. Heuristic Methods Comparison **77**

List of Figures

2.1 A TSP solution representation, compared with a VRP solution. 17

3.1 Structure Diagram of the Solution Modules. 29
3.2 Flow of storing a request, and calculating routes. 31
3.3 Diagram of a Request Life-Cycle. Thicker borders indicate Final States. 33

4.1 The Proposed Solution Technological Architecture. 37
4.2 The Resources Module Entities Schema. 39
4.3 The Requests Module Entities Schema. 46

List of Tables

2.1 Mobility Solutions examples by type of Use and type Access. 12
2.2 Comparison between the various mobility solutions. 16

5.1 Requests API Endpoints. Note: the numbers represent identifiers of the entities 54
5.2 Resources API Endpoints. Note: the numbers represent identifiers of the entities 57
5.3 Open Vehicle Routing Problem (OVRP) Test Instances Performance 64
5.4 Dial-a-Ride Problem (DARP) Test Instances Performance 64
5.5 Lisbon Metro Test Instances Performance 65

A.1 Comparison between the results obtained by Tan et al. against historical best. NV: Number of Vehicles. TD: Total Distance. 2-INT: 2-interchange (Taillard's Algorithm). SA: Simulated Annealing. TS: Tabu Search. GA: Genetic Algorithm. 78

List of Algorithms

1 DARP Solver 43
2 Genetic Algorithm 44

Acronyms

API	Application Programming Interface
BKS	Best Known Solution
CVRP	Capacitated Vehicle Routing Problem
DARP	Dial-a-Ride Problem
DNA	Deoxyribonucleic Acid
DRT	Demand-Responsive Transport
DTO	Data transfer objects
GPDP	General Pickup and Delivery Problem
HFFVRP	Heterogeneous Fixed Fleet Vehicle Routing Problem
HSL	Helsinki Regional Transport Authority
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JWT	JSON Web Tokens
LSMVH-DARP	Large Scale Multi-Vehicle Heterogeneous Dial-a-Ride Problem
LSHFFOVRPTWSPD	Large Scale Heterogeneous Fixed Fleet Open VRP with Time Windows and Simultaneous Pickup and Delivery
LSVRP	Large Scale Vehicle Routing Problem
MVC	Model-view-controller
NEO	Network and Emerging Optimization

OVRP	Open Vehicle Routing Problem
PMX	Partially-Matched Crossover
REST	Representational State Transfer
TSP	Traveling Salesman Problem
VRP	Vehicle Routing Problem
VRPPD	Vehicle Routing Problem with Pickup and Delivery
VRPSPD	Vehicle Routing Problem with Simultaneous Pickup and Delivery
VRPTW	Vehicle Routing Problem with Time Windows

1

Introduction

Contents

1.1 Service Description 3
1.2 Service Flow 4
1.3 Motivation 4
1.4 Scope 6
1.5 Objectives 7

The present report refers to a project to build a software solution for public transport providers, in order to provide a demand-responsive public transportation service.

This service aims to connect public transport providers, who input and manage their information; to passengers, who submit trip requests.

The main purpose is to allow already operating providers to offer this new type of service, to face the increasingly competitive mobility market, which has a fair variety of solutions.

These solutions range from the cheap, yet inflexible regular public transportation services (regular bus, subway and tram, etc.), to the expensive and flexible driver services (as taxis or electronic hailing platforms, like Uber). Some of these driver services are trying to offer shared rides in order to lower the price - carpooling solutions. Despite having similar advantages to demand-responsive transport, carpooling is not the same. While in carpooling passengers are matched with drivers beforehand, a demand-responsive public transport must offer the possibility for passengers to request a trip with immediate effect. This requires sophisticated algorithms to optimize matches for faster and shorter routes - optimized to meet both the passengers and the provider preferences. Also, public transport services are linked to public authorities, which ensure service quality, while driver services are not, being the driver the main responsible for the quality of the service.

The idea of developing such a software service came from an initial contact with a company that produces software to public transport providers. From this contact, some of the service traits were defined, but due to many doubts on what should be the business model, we decided to give providers some flexibility and not define a fixed business model.

This allows the service to adapt to different Demand-Responsive Transport (DRT) transport services (which might have different business approaches), and also adapt to different scenarios, like city centers or longer range bus services; but always in the passengers market (the resulting system, and its nomenclature are not adapted to freight services).

1.1 Service Description

Providers have a fleet of vehicles, operating in a defined geographic area. The vehicles are used to fulfill requests within this operating area. Passengers make trip requests from a pickup point to a delivery point, being also able to specify the time window they want to depart, and the time window they want to arrive.

They should then be able to check the status of their trip: the attributed vehicle and driver, the vehicle route, the estimated time to reach the departure point, and the estimated time to reach the arrival point. Other information might also be available, the average rating of the driver's trips, or the vehicle capacity and occupation.

From the provider point of view, the driver can also consult any new requests attributed to them, or changes in the current route.

When presented with a route proposal (a solution), a passenger must then choose either to **Confirm** the request, or to **Cancel** it. If they confirm, the vehicle will then serve the request.

New requests (from new passengers) might occur as vehicles are serving previous ones, being the routes adapted to these new requests. A passenger cannot submit simultaneous requests, as if they submitted one, it must get served or cancelled before submitting another.

1.2 Service Flow

Incoming trip requests are immediately persisted. Then, an algorithm will distribute these requests the best way possible to available vehicles, producing their routes.

The algorithm should minimize passengers' waiting times, and routes total traveled distances. The goal is to raise the service attractiveness in the passenger point of view, and reduce trip costs for the provider. These costs depend directly on the trip distance and time of travel, and on the number of vehicles used.

A passenger trip starts when the attributed vehicle arrives at the requested departure point, and finishes when the vehicle arrives at the requested arrival point. Despite completing one passenger trip, the vehicle keeps getting and serving requests, and route is updated on the go. Yet, new requests can only be added to ongoing routes if they don't impact the other requests (breaking their expected departure or arrival time windows).

Similarly to regular transport services, stops allow both entries and exits of passengers.

Providers are also able to minimize travel times by specifying a tolerance distance, for which close enough request points will be joint in a single stop. Alternatively, a provider may specify a collection of fixed stop points, which are the only departure and arrival points possible. It is not possible to specify both at the same time.

If a vehicle finished serving its route, and is not attributed any new requests, there should be a default action predetermined by the provider (i.e. returning to a depot).

A passenger should always make the desired trip in a single vehicle route, with no connections.

1.3 Motivation

A service like this will cover the lack of transport services in certain periods of the day, or even in certain locations (less crowded locations away from city centers). This way, it might try to replace some regular transport services.

On another side, new European legislation changed how concessions should be made inside the European Union, and their regularity. This makes the transport providers' business more competitive and open to new providers. This way, more flexible services might raise their popularity to win concessions.

1.3.1 Demand-responsiveness Advantages

On one side, passengers benefit from demand-responsive public transport, as a new possibility of service in areas or at times previously nonexistent in public transport. This offers them more flexibility, at a fair monetary price (as described in Chapter 2). By running in an electronic platform, this service also saves them the time and work of planning the trip.

On other side, public transport providers can benefit with a better management of their resources, as demand-responsive services allow them to adapt routes and vehicle usage instantly to current demand, instead of having to study passengers demand (which is a difficult and time consuming task) *a posteriori*. As such, they might minimize operation costs.

For both sides, demand-responsive public transport makes possible to create courses that are not available in regular public transport.

On yet another side, public authorities raise their popularity by offering more diverse mobility solutions, attracting passengers to use them, with population well-being and environmental improvements in mind.

1.3.2 European Directive

So far, concession contracts between public authorities and public transport providers were long-lasting, and not focused in quality, as only a small number of providers were competing for concessions. In some cases, a provider is the only one competing for concessions.

The Directive 2014/23/EU of the European Parliament and of the Council of 26 February 2014 on the award of concession contracts [2] aims to solve these problems, by making award concession contracts more regular and monitored, aiming to:

- increase market competitiveness in the sector
- improve services quality
- allow innovative models and smaller companies to penetrate the market

It is believed that accomplishing these aims will enrich economic activity, and increase public satisfaction.

Monitoring these concessions will evaluate their quality of service, within the period of activity. The evaluation metrics used could be, for example, frequency of service, schedule fulfillment, or vehicles' condition, as examples.

Concessions monitoring may be performed in different ways:

- feedback and opinions submission by passengers
- tracking systems that detect vehicles' position in real-time
- inspections and examinations made by public space management entities

This directive applies to existent regular transport services. So, in order to raise their popularity (and therefore win concessions), providers might want to make use of a DRT service as a possible strategy.

Here enters a service like this, a software solution for these transport services providers.

1.4 Scope

This work consists in the development of a software solution to accomplish a service like we have described.

Due to its academic nature, the work will only regard the back-end part of the solution. User interfaces, ticketing and payments will not be addressed. They are complex by themselves, and only a professional team with more man-hour resources could accomplish such a work with these many features and requirements.

The main parts of the work are:

- developing a software service to receive requests
- developing an optimization algorithm module to process them
- developing test and simulation environments for the algorithm
- testing all these modules, both separately and joint

Some of the algorithm inputs will be produced by an external integration with a maps Application Programming Interface (API), that will take care of the real-world distance and travel times calculations.

Some extra features will be added for passenger to submit their feedback on the trip (rating), and consult the current route (path and information on the vehicle and driver).

1.5 Objectives

The main objectives to accomplish are:

1. Research and Analyze previous and relevant work done in demand-responsive road transport services
2. Develop and test the modules to accomplish the service features
3. Evaluate the performance of the service against current methods
4. Use the service for small studies in the public transport field

2

State of Art

Contents

2.1 Mobility Solutions 11
2.2 On-Demand Services 14
2.3 Vehicle Routing Problem 16
2.4 Genetic Algorithms 22

This work is ultimately a project, fact that emphasizes the need of two types of analysis: one in a conceptual side, and one in a technical side. For the conceptual parts, sections 2.1 and 2.2 describe the main concepts and how they were applied in other projects. Section 2.3 describes the algorithmic techniques and approaches to the problem, and section 2.4 described the studied applications and properties of genetic algorithms, like operators, representations, and evaluation functions.

2.1 Mobility Solutions

The Geography of Transport Systems, by Jean-Paul Rodrigue, Claude Comtois, and Brian Slack, states the purpose of Transport as ”to overcome space” [3], being the movement of both people, goods, and information a crucial part of human societies. They also define ”Transport as a Derived Demand” [3], as mobility needs mostly come from other. Mobility needs cannot exist alone.

But transport has costs. Overcoming space is subject to the physical distance between origin and destination, as well as many other barriers, like the physical geography of the path, weather conditions, logistic tasks and costs, etc. These define the **Logistical Distance** of a trip.

The best way to minimize costs is to minimize the physical distance between demand fulfillment providers and people. This, aligned with other factors, concentrates most people in cities. In 2016, 54% of total World population lived in cities, according to The World Bank[1] [2].

Urban mobility demands are, therefore, the most frequent. With the growing numbers of people, comes the growing of urban mobility demands. As changing infrastructure or the space shape, to reduce congestion and distances, is a very expensive investment, the need of mobility solutions rises as an alternative solution to problems like traffic, pollution, excessive budgets on transport services, or the lack of quality of those.

2.1.1 Types of Mobility Solutions

Following UITP's[3] Policy Brief on Public Transport at the Heart of Urban Mobility Solution[4], which, amongst other recommendations, divides modern mobility solutions into being of *Individual* or *Collective* use, and *Public* or *Private* access, where *Access* relates to who can request the service. *Public* mobility solutions are provided by companies and public space management entities to the general public, while *Private* mobility solutions are provided only to people authorized by the provider . *Use* relates to the number of people that use the service or vehicle in a single trip.

Types are pictured in Table 2.1.

[1] The World Bank, Online: http://www.worldbank.org/. Last accessed: 1 May 2019
[2] World Bank Open Data, Online: https://data.worldbank.org/indicator/SP.URB.TOTL.IN.ZS. Last accessed: 1 May 2019
[3] Union Internationale des Transports Publics, Online: http://www.uitp.org/. Last accessed: 1 May 2019
[4] Public Transport at the Heart of Urban Mobility Solution, April 2016

Table 2.1: Mobility Solutions examples by type of Use and type Access.

	Collective Use	Individual Use
Public Access	Public Transport, Demand-Responsive Transport	Bike-Sharing, Car-Sharing, Taxi, Car Rental, Bike Rental, E-hailing Services
Private Access	Chartered services, (like School or Company Bus), Car-Pooling (e.g. BlaBlaCar[5]), Ride-Sharing	Own Bicycle, Own Car, Pedestrian

Some of the E-Hailing services also provide Public Accessible-Collectively Used solutions, like uberPOOL[6] as a service provided by Uber[7].

It is also possible to separate mobility solutions into two different groups: regular or on-demand, being regular solutions the one which have predefined schedules and on-demand the ones which only occur when requested.

In this subsection, the types of mobility solutions that are relevant to this work will be described in detail.

2.1.2 Electronic Hailing

Hailing is the commonly known act to signal a Taxi to request a trip.

Electronic Hailing (or E-Hailing) is the act of requesting a transport service via an electronic device. Some examples of E-Hailing solutions include Uber, Lyft[8], Taxify[9], or Cabify[10].

The need for electronic hailing came from the ability of users setting the starting and arrival points beforehand, knowing roughly the time the trip will take, how much will they wait until picked up, and the final cost of the trip. This information allow a better quality of service, by removing most of uncertainty. The costs are not much of a concern, as prices are not much different from regular taxis'.

Most platforms also include feedback systems, where users' can evaluate the trip, the driver, the vehicle conditions, amongst others, being this another factor of better quality, as providers can improve it faster and accurately know the satisfaction level of each trip.

2.1.3 Public Transport

Public transport is collective transport of passengers, using travel systems available to the general public, falling into the collective use-public access mobility solution type.

In this kind of service, trips runs mostly in fixed routes with fixed embark or disembark points, and following predefined schedules.

[6] uberPOOL, Online: https://www.uber.com/pt-PT/ride/uberpool/. Last accessed: 1 May 2019
[7] Uber Technologies Inc., Online: https://www.uber.com/. Last accessed: 1 May 2019
[8] Lyft, Inc., Online: https://www.lyft.com/. Last accessed: 1 May 2019
[9] Taxify, Online: https://taxify.eu/. Last accessed: 1 May 2019
[10] Cabify, Onlyne: https://www.cabify.com/. Last accessed: 1 May 2019

Providers of public transport can either be privately or publicly held companies, which in both cases aim to provide a cheap and fast mobility solution to their users.

Public transport is also divided in modes, like road, rail, tram, or light rail transit, depending on the infrastructure and technological solution it runs on.

UITP's Statistics Brief of Urban Public Transport in the 21st Century[11] contains a modal distribution of all public transport journeys in 2015, being road transport responsible for 63% of trips, metro for 16%, suburban rail for other 16%, and tram and LRT (light rail transit) for the remaining 5%.

2.1.4 Demand-Responsive Transport

DRT is a subgroup of Public Transport, where routes and schedules are flexible to users' needs.

This kind of transport is commonly used in areas with lower demand, which do not justify a regular service. In higher demand areas, it is an emergent solution to provide a more flexible service at a fairly low cost to users (less than driver services, like electronic hailing or taxis), while reducing operative costs for transport service providers as well.

Demand-responsive transport services differ from the conventional ones in many aspects:

- Regular Bus Routes: DRT has flexible routes and schedules
- Shuttle Bus: DRT may not have fixed Departure or Arrival points
- Paratransit: DRT is available to the general public, while paratransit is available only to certain people (e.g. elders, people with disabilities)
- Regular Taxis: DRT are booked in advance, Taxis usually work in an ad-hoc basis
- Taxis and other driver services: DRT is a collective transport service, being the usual number of passengers higher than in driver services
- Carpooling: carpooling service providers cannot guarantee safety nor quality to passengers, as the driver is the only responsible for the trip. DRT is a public transport service, where providers are responsible for passengers' safety
- Car-Sharing: car-sharing involves the rental of a car for a single trip, where users can change course at any point. DRT does not allow changes in the route after the trip starts

This type of services are often portrayed as a midterm between the flexibility of E-Hailing solutions, and the low cost of Public Transport.

[11] Urban Public Transport in the 21st Century, October 2017

2.2 On-Demand Services

There are already some projects for DRT platforms, and even some up and running services. In this section those will be described and discussed.

2.2.1 Kutsuplus

One of these is Kutsuplus [13], a pilot project ran in Helsinki, Finland, led by a partnership between Split Finland Ltd. (earlier Ajelo Ltd.), and the Helsinki Regional Transport Authority (HSL)[12], from 2012 to 2015.

In Finland, Kutsuplus passengers could use a mobile application to request trips between so-called virtual stops - specific points designated for passengers' pickup and delivery.

As stated in the project final report, the main purpose was to calculate routes based on passengers requests, "in such a way that customers heading approximately in the same direction are directed to the same vehicle". The main challenge is then roads traffic, by its unpredictable nature. Kutsuplus solution for this problem was a system designed to quickly adapt to changes in traffic, redirecting the vehicles to avoid congestion.

In a passenger's point of view, the quality of a service is often dependent of its punctuality (the estimated times of pickup and delivery being accurate). In their solution, Kutsuplus tried to minimize error in time estimations by not picking up passengers from custom locations - fixed virtual stops only. This allows a major reduction in the possible paths for algorithms to optimize.

The project was a great success, as measured by customers' feedback, and statistics over time. In 4 years, over thirty-thousand people registered as passengers, generating almost nine-hundred thousand euros (900.000€) in ticket revenues.

Estimates from the final report also point to major savings in inefficient use of time, in the form of unnecessary public transport operative costs. This could redistribute the local authorities budget.

The downside of such a project is the initial financial investment. The goals of budget savings and environmental impact are only achieved when the service gets up-scaled to the hundreds of operating vehicles, being only profitable at that scale. As such, projects like this should not be started unless local authorities are backed up with a strong investment (often from the private sector). Alternatively, it also possible that providers might want to reassign some of their vehicles to on-demand service, instead of regular service.

This strategy is much more competitive, as they are already established and don't require such a big investment on the fleet.

[12] HSL, Online: https://www.hsl.fi/en. Last accessed: 1 May 2019

2.2.2 MOIA

Split Finland was recently acquired by MOIA[13], a company from the Volkswagen Group. But before acquisition, they ran another DRT solution in Washington D.C..

Split replicated the model used in Helsinki (fixed stops, demand-responsive ride hailing service), this time competing with other hailing services, like Uber and Lyft. Both Uber and Lyft had carpooling services (shared rides), which pose the main concurrence to a service like a DRT.

Split also intended to compete with existent public transport solutions, like the local metro, and the regular bus services.

Such a competitive environment led to a saturated market in the hailing business. At first, Split made good expansions, by adding three neighborhoods to their operative area after five months of activity. But then, big discounts made by Uber and Lyft turned passengers to cheaper solutions. DRT's premise was to be a cheaper solution than E-Hailing, with more flexibility than regular public transport. Not being able to fulfill their premise, Split discontinued their service.

But all this experience and technical expertise would be a huge waste. Here comes MOIA. By buying Split Finland, MOIA could take the technical expertise and pioneer algorithms to a new business model of shared rides.

MOIA based their business model in comfort of the ride, with a innovative vehicle design, aiming to add more personal space to each of the six seats. The vehicles are electric-powered, as another innovation towards environmental sustainability.

The vision of the company is to provide ride services that will reduce the number of private cars in cities, reducing traffic, accident rates and pollution. All of this, while providing a flexible mobility solution.

2.2.3 Via

Via[14] is a company that offers the technology for shared rides services to transport providers. They partner with these providers by licensing their platform - a software solution, just like the one we built in this work.

Providers can obtain the technology to provide on-demand ride services, and Via backs them up with consultancy services. A partnership that aims to transform public transit.

The idea is to provide on-demand shared rides, nearly at the price of a bus. The way the service works is in everything similar to E-Hailing, introducing the licensing part as a possible decisive factor. Providers can access the technology without having to build it from scratch.

This factor might work as a way to more providers being able to compete.

[13] MOIA, Online: https://www.moia.io/. Last accessed: 10 May 2019
[14] Via, Online: https://platform.ridewithvia.com/. Last accessed: 10 May 2019

Table 2.2: Comparison between the various mobility solutions.

Solution	Price	Flexibility	Safety Responsible	Vehicle	Traffic Reduction
Public Transport	Low	Low	Public Authorities	Bus	High
Taxi	High	High	Driver	Car	Low
Carpooling	Medium	Medium-High	Driver	Car	Medium
Kutsuplus	Medium	Medium-High	Public Authorities	Mini-bus	Medium-High
MOIA	Low	Medium-High	MOIA	Custom Van	Medium-High
Via	Provider-Dependant	Medium-High	Provider	Provider-Dependant	Medium-High

2.2.4 Discussion

After looking into the different types of mobility solutions and services providers can offer, it is now possible to analyze how the different projects can inspire this work:

- **Kutsuplus** was the pioneer project of a previously-booked, demand-responsive public transport service. From it, we take the lessons of (1) building a service to quickly adapt to changes in course, and (2) the need of scaling it, as a smaller service is not profitable, nor can properly reduce traffic in a city.
- **MOIA** is the result of **Kutsuplus** and Split's Washington D.C. experience of DRT, that focuses in providing a high-comfort, sustainable service to the public, by specially designing a vehicle for the purpose, and solving the scaling need by financially supported by a big corporation.
- **Via** provides software services to transport providers, helping them to innovate onto more flexible solutions. This work aims to build the back-end software for a similar purpose.

The various mobility solutions and projects' properties are illustrated in Table 2.2.

2.3 Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a combinatorial optimization problem, in which a fleet of vehicles is given the task of fulfilling customers' transport requests, and the goal is to minimize the total cost of performing such a task. It is the problem representation chosen for this work.

The VRP was put forward by Dantzig and Ramser [4], back in 1959, as an algorithmic approach applied to petrol distribution.

It is a generalization of the widely-known Traveling Salesman Problem (TSP)[15], first studied in 1930, by Karl Menger, with multiple vehicles. There is previous literature from the 1800s, namely by the mathematicians William Rowan Hamilton and Thomas Kirkman, where the problem is defined, but no algorithmic solutions are proposed.

[15]TSP, Online: https://en.wikipedia.org/wiki/Travelling_salesman_problem. Last accessed: 11 April 2019

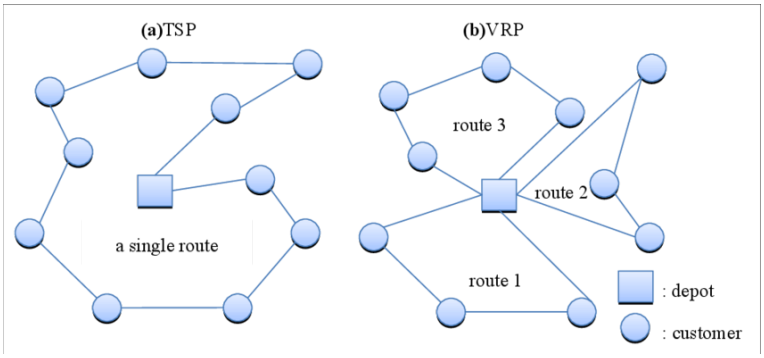

Figure 2.1: A TSP solution representation, compared with a VRP solution.

Both problems start with a graph, containing all vertices (customers) to be visited, the initial position(s) of the vehicle(s) (depot(s)), and the edges, which indicate travel costs between vertices. The goal is to obtain the route which minimizes the total cost of travel.

Obtaining solutions for the classical TSP is trivial, since a solution is an ordered list of vertices, where both the first and last elements are the depot of the vehicle. Any combination of the remaining dots is a solution.

Similarly, obtaining solutions for the VRP is also trivial, since a solution is a set of trips, each containing the vehicle that will perform that trip, and the same ordered list of vertices to visit in the trip.

These problems are very similar, but differ in the perspective of the optimal solution for a VRP being the one that minimizes total cost for all vehicle trips, which are assigned only part of the total vertices to be visited. An optimal solution for a VRP might not make use of the whole fleet of vehicles.

2.3.1 VRP Variants

The VRP is a very rich problem, since it has multiple applications in real-world cases. Each case has its own assumptions and features, producing variants of the problem that work slightly differently. Despite there is a large variety of variants, only the ones that apply to this work will be visited in detail.

Since working with real world transport, it is known that vehicles have a limited capacity. Therefore, they cannot carry more passengers (in this case) that allowed. This puts the problem with new constraints: now each vehicle has a defined maximum capacity, which must be taken into consideration while the vehicle travels. The fleet can be heterogeneous. As such, different vehicles might have different capacity. For this case, the Capacitated Vehicle Routing Problem (CVRP) [6] applies, as well as the Heterogeneous Fixed Fleet Vehicle Routing Problem (HFFVRP) [5]. As we are dealing with public transport providers, most of them own their fleets. As such, we will consider fixed fleets, making this an HFFVRP [7].

Passenger transport also implies that each passenger has a defined pickup and delivery points.

This means that, in order to fulfill the passengers order, the vehicle must visit its pickup point before its delivery point. This new constraint, besides reducing the universe of possible solutions, introduces the Vehicle Routing Problem with Pickup and Delivery (VRPPD) [8] variant. In this work's particular case, we also might have the case of condensing both pickup and delivery points (departures and arrivals) into a single stop point. It makes this a simultaneous pickup and delivery case, which is a Vehicle Routing Problem with Simultaneous Pickup and Delivery (VRPSPD) [9].

Then, the vehicles are transporting passengers around, not having the need of a central depot to load. Loading and unloading corresponds to picking up and delivering passengers on-the-go. It is also interesting to keep the vehicle actively transporting passengers without stopping (unless for technical motives, like refueling, maintenance, driver mandatory breaks or shift changes, etc.). As such, solutions do not require the vehicle to come back to a specific point. This makes our problem an OVRP [1], and our solutions do not require the last vertex. The first vertex must still be the current position of the vehicle, though. When finished a trip, it is intended to pick new requests, and only go to a default depot if none are made.

Our customer might also want to schedule a trip to start or finish at determined times, making the problem bound to time windows. As time plays a crucial part of the optimization in this kind of service, the Vehicle Routing Problem with Time Windows (VRPTW) [10] is also applicable.

The combination of simultaneous pickup and delivery with the absence of central depots and time windows makes our problem a General Pickup and Delivery Problem (GPDP), in its special case of a Dynamic Multi-Vehicle DARP [11], as it is passengers being transported.

Finally, we might need to scale this service to fulfill thousands of requests in the same area. The previous variants of the DARP often present good solutions for hundreds of customers. As this work requires a larger scale, it visits the Large Scale Vehicle Routing Problem (LSVRP) [12], by stepping from the hundreds to the thousands.

Combining all restrictions of the problem this work aims to study and solve, we get a Large Scale Heterogeneous Fixed Fleet Open VRP with Time Windows and Simultaneous Pickup and Delivery (LSHFFOVRPTWSPD), or alternatively, a Large Scale Multi-Vehicle Heterogeneous Dial-a-Ride Problem (LSMVH-DARP).

To note that many restrictions make the problem harder to solve, as variations in a solution have a higher probability of breaking constraints.

2.3.2 Solving the VRP

Finding a feasible solution is trivial in the VRP, it just requires any solution that matches all the constraints of the problem being solved. The challenge is to find the optimal one, if possible. Otherwise, find the best possible.

Definition 2.3.1. Feasible Solution A feasible solution is one that satisfies all constraints of the problem it solves.

The VRP is considered an NP-Hard problem[16]. The Network and Emerging Optimization (NEO) Research Group[17], from the University of Málaga, Spain; made a great compilation [14] of information about this problem.

This compilation involves many articles with experimental results in many different techniques used to solve it:

- **Exact Methods**: Branch-and-Cut and Branch-and-Bound approaches
- **Heuristics**: 2-Phase Algorithms and Constructive Methods
- **Metaheuristics**: Ant Algorithms, Simulated Annealing, Deterministic Annealing, Genetic Algorithms, and Tabu Search

2.3.2.A Exact Methods

A method employing a classical search that explores all possible solutions. This kind of systematic search method finds optimal solutions, but takes an enormous amount of time to compute them. Examples are the Branch and Cut [16] method, and the Branch and Bound [15] method. These approaches divide the problem into subproblems, which are smaller and easier to solve, until a level of trivial decision is attained (branching). Then, by combining the solutions to subproblems we get to the final (optimal) solution.

2.3.2.B Heuristics

These methods are very similar to **Exact Methods**, with the difference of using heuristic functions that allow pruning of alternatives that are worse than the best solution found so far. This pruning allows a reduction in the solution space.

Heuristic methods can be separated into two different types: **2-Phase Algorithms**, in which there is a clustering phase of vertices into feasible routes, and a second phase of route construction; and **Constructive Methods**, which try to gradually build a feasible solution, while keeping and eye on costs. **Constructive Methods** do not contain an improvement phase.

2.3.2.C 2-Phase Algorithm

There are two types of **2-Phase Algorithms**: (1) Cluster-First, Route Second, and (2) Route-First, Cluster-Second.

[16]NP-Hardness, Online: https://en.wikipedia.org/wiki/NP-hardness. Last accessed: 1 May 2019
[17]NEO, Online: http://neo.lcc.uma.es/vrp/. Last accessed: 1 May 2019

The clustering phase involves assigning each vertex of the graph to a vehicle. It can be done by any clustering algorithm, as far as the centroids do not change (vehicles have fixed positions, and only the attributions of vertices might change).

If clustering is done first, then solving the VRP is reduced to solving a classic TSP for each cluster (single vehicle trip that passes through all points). The routing phase is solving the TSP for this case, which might contain feedback loops. Examples of this approach are described by Fisher and Jakumar [17] (Generalized Assignment Heuristic), in the Petal Algorithm [18], the Sweep Algorithm [20], or the Taillard's Algorithm [19].

If we choose to do routing-first, the routing phase consists in solving a giant TSP, followed by decomposing this route into feasible vehicle routes. This approach was first mentioned by John E. Beasley [21], but there is not much work done around it. The method itself does not seem very effective, as it disregards some constraints in the routing phase.

2.3.2.D Constructive Methods

Constructive methods are heuristic methods that gradually build a feasible solution, while keeping solution cost under control.

An example of this approach is the Savings Algorithm [22], with its variant, the Matching Based Savings Algorithm [23]; which consists of starting with atomic vehicle routes (from the depot to each vertex and back to the depot), and start merging the ones that produce the highest savings. A saving is the difference between the cost of making the routes separately and combined (i.e., the saved distance).

Another example is the Multi-Route Improvement Algorithm [24], which tries to improve a solution by performing exchanges within, or between routes.

2.3.2.E Metaheuristics

These methods are a subset of **Heuristics**, that have an extrapolation to physical world examples (a metaphor).

They distinguish themselves from the remaining methods, by having mechanisms designed to avoid being stuck local minimums of the solution space (which is common in these complex problems, with huge solution spaces).

Some metaphor examples are ant colonies (**Ant Algorithms** [25]), solid mechanics (**Annealing** [26]), or genetic evolution (**Genetic Algorithms** [27]).

Ant Algorithms have two main phases - construction and trail update, where the construction phase builds initial feasible solutions (if an ant route gets unfeasible, it is told to return to the depot), leaving a pheromone behind in each step (vertex or edge, depending on the implementation). This pheromone will indicate the quality and desirability of the taken trail to following ants (the quality of the route, and

the probability of taking that route again, respectively). Then, trail updates occur when a better route is found starting on the same vertex or edge.

Annealing uses a mechanical analogy, where solids are heathen up until they can be malleable, and then slowly cooled down as they gain shape. The temperature here is a threshold, which allows to escape local minimums, by accepting worse solutions than the current best.

Genetic Algorithms use an analogy of population training for fitness, mirroring mechanisms of selection, recombination, and mutation.

The selection phase consist of randomly choosing two parent individuals from the population for mating purposes. The probability of selecting a population member is generally proportional to its fitness in order to emphasize genetic quality while maintaining genetic diversity. Here, fitness refers to a measure of profit, utility or goodness to be maximized while exploring the solution space.

The recombination or reproduction process makes use of genes of selected parents to produce offspring (children solutions) that will form the next generation.

A mutation consists of randomly modifying some genes of a single individual at a time to further explore the solution space and preserve, genetic diversity. The occurrence of mutation is generally associated with a low probability.

While selection and recombination processes try to reach minimum possible cost, mutation inserts a small probability of a random swap, in order to avoid local minimums again.

There is yet another metaheuristic technique: **Tabu Search** [28]. It explores the solution space based on the current solution's neighborhood (the set of solutions that very similar to the current one, according to an implementation-defined set of rules), in a process called Local Search. To avoid getting stuck in local minimums, the search allows worse solutions, as far as they are not repetitions of previously found solutions. These are marked as Tabu (not to be revisited for a while) until the tabu time is over.

Most metaheuristic methods have similar approaches, ones better adapted to some variants of the VRP, others to other variants.

2.3.2.F Discussion

It is widely known in the literature that **Exact Methods** are easily outrun by heuristic and metaheuristic methods for bigger problems.

Annexed to this report are the experimental results obtained in the study of heuristic methods in [29] (Table A.1).

To note that these results were obtained for the VRPTW, which differs from the problem being solved in this work.

The point of these results is to show that, in general, Tabu Search and Genetic Algorithms are the best performing techniques for this kind of problem, and are the ones that will get more attention in this

work for building the optimization algorithm.

2.4 Genetic Algorithms

Following the discussion on the various solution methods, we chose Genetic Algorithms to serve as the requests processing algorithm. They offer a bigger variety of possibilities and adaptations to explore, in order to achieve a good performance.

This section shows some of the methods and operators related to the produced genetic algorithms used in this work.

2.4.1 Methodology

Genetic Algorithms start by initializing a population of individuals - generated solutions of the problem. This generation can be random or seeded. Random generated solutions allow a better distribution all over the solution space, while Seeded generation allows the population to be more focused in an area more likely to produce optimal solutions.

Then, the population is evaluated, producing a fitness measure - a number that represents an individual's quality to solve the problem. After that, genetic operators - selection, mutation and/or crossover. The application of these operators will produce a new population - the next generation.

This process of evaluation and operators application is then repeated until the algorithm meets its termination conditions. These can be resources allocation limits (time or memory/space), maximum number of generations, or maximum consecutive unimproved generations. Other termination conditions might exist, but were not considered for this kind of problem due to its nature.

2.4.2 Population

For each specific problem, we should choose an appropriate representation for our Deoxyribonucleic Acid (DNA). To do this, we should keep in mind that the whole metaphor for Genetic Algorithms is based on the fact that DNA is a sequence of genes, and that changes on this sequence produce either fitter individuals, or deficient ones.

TSP problems are often represented as a sequence of numbers [30], being each the index of the point/city to visit, or request to fulfill. In the DARP this is a bit more tricky, since we have pickup and delivery points. In order to accomplish that, it is possible to represent the points, but expanding the solution space to unfeasible solutions, in the case a delivery point comes before a pickup point for the same request. Gintaras Vaira [31] suggests methods to overcome this kind of problems, through penalties

in fitness, treating the problem as multi-objective, repairing solutions, or by preserving feasibility in the genetic operators.

Adding multiple vehicles to the mix, there are many routes, whose order does not matter. Only the order of points to visit within a route matters. So, we can opt for a multitude of representations, being the most common a binary matrix [34], or a sequence of pairs point-vehicle.

In the binary matrix, the columns represent each request to fulfill, and each line represents a vehicle of the fleet. So, each entry *i,j* is a Boolean value indicating whether that vehicle *i* satisfies request *j*, or point *j* if we choose to use the points in this representation. This way, we can assure DNA sequences always have the same size. Sequences might get really long for a larger number of vehicles, though.

The pairs representation also assures sequences to have the same size, as all requests/points are attributed to one vehicle, no matter how many vehicles are used.

In this work, the pairs representation with request indexes instead of points seems more appropriate, to guarantee both sequences size does not depend on the number of vehicles, and that all request stop points are in the same route. There can be repetitions of points across routes, for instance, in a case of two opposite requests, one from point A to point B, the other from B to A.

2.4.3 Selection

The selection operator is the one to choose which individuals should pass their genes into the next generation. This process guides the search towards higher fitness possibilities, assuming that fitter individuals are more likely to produce even better offspring than less fit individuals.

There are several methods to perform selection, being the simplest just choosing a portion of the top of the population, ordered by fitness. This might stick the search to local minimums, though.

Then, there are fitness proportionate methods, like the Roulette Wheel Selection [31], Tournament Selection [31], or Rank-Based Selection [32].

Fitness Proportionate Roulette Wheel Selection is made through creating a cumulative normalized fitness vector with the same size of our ordered population, then generating a random number *R* between 0 and 1, and then selecting the individual whose cumulative fitness value is higher than *R*.

The simplest Tournament Selection is made by selecting a random sample of our population, and then using the winner of each tournament for **Crossover**. The winner of each tournament can be assumed to be the individual with the highest fitness, as no other element can match them. In the case of a tie, choose randomly.

Tournament Selection can also be probabilistic, where the best individual of each tournament is selected with probability *p*, the second best with probability $p(1-p)$, the third with probability $p(1-p)^2$, and so on, such that element *k* is selected with probability $p(1-p)^{(k-1)}$.

Rank-based selection is also a roulette wheel selection (just like the Fitness Proportionate), where probabilities are calculated by:

- Ordering the population by fitness value
- Ranking the ordered population (from 1 to the size of the population)
- Choosing a Selective Pressure (SP) value, between 1 and 2
- Calculating probability $p_i = 2 - SP + 2(SP - 1)\frac{i-1}{pop\_size-1}$

2.4.4 Mutation

Mutation operators transform an individual into a slightly different one. It is a method that allows the algorithm not to be stuck in local minimums, as it can be applied to any individual, regardless of its fitness value. Mutation probabilities may also decrease with higher fitness (individual less susceptible to environmental influence).

From the many mutation possibilities in [31], we only looked into two simple ones: Gene Reinsertion [33], and Gene Swap [33]. Gene Reinsertion consists of randomly selecting a gene, and then randomly selecting a position on the DNA to insert it again. Gene Swap consists of randomly selecting two genes, and swapping them.

In this work we might look into both methods, as there is no strong proof of any being far superior to the other.

2.4.5 Crossover

These operators consist of combining two individuals (parents) into a newly generated one - the child. Many crossover methods exist, but only some of them are good enough for this kind of problem. As it is very sensitive to changes in attributions (request-vehicle pairs), crossover operators should be carefully picked up not to produce deficient solutions, or solutions that are far worse than the parents.

2.4.5.A Simple, Bad Crossover Operators

The simplest crossover operators are K-Point Crossovers, in the simplest case for $k = 1$. One-Point Crossover consists in choosing one index of the DNA at random to be a crossover point. Then, from the beginning until our crossover point, the genes passed to the child come from the first parent. After the crossover point until the end, the genes come from the second parent.

For higher values of *k*, *k* point are chosen as toggle points, and, when iterating through the genes to pass them to the child, each time we hit a crossover point, we toggle the parent we get them from, just like explained for One-Point Crossover.

These methods are not very good as Crossover Operators for VR problems, as they are very destructive. They generate deficient and worse solutions more easily than they can generate better, or even feasible children. There can also occur the case of repeating requests or points across routes, making the child solution also deficient.

2.4.5.B PMX Crossover Operator

This crossover method is a more tricky one, yet it seems to be much better than the previous ones in comparison. It makes sure that most child solutions are not deficient, yet it can happen if we choose the points representation (delivery point before its pickup within the same request).

Partially-Matched Crossover (PMX) consists of selecting a start index, and an end index, to represent a swath. Then, we will pick the first parent's swath, and copy it to the child directly, in order to preserve its genes. After this, we will search the genes present in the second parent swath that are not in the first parent swath, and check for the sites to place them.

As described in [35], this process preserves the elements, avoiding repeated or missing genes, fact that allows this method to automatically prune the search space on all deficient possible children with repeated or missing genes, speeding up the search process.

In this work, we will look into this crossover method, despite there are many others [35].

2.4.6 Local Search

In order to reduce the search space, it is suggested [36] to distribute requests by the fleet of vehicles, and then perform a local search to optimize each route as a TSP. This way, the solution space is reduced to only the possible combinations of request attributions to the vehicles, and order does not matter anymore for the global search representation, being only relevant in the local searches.

3

Solution Proposal

Contents

3.1 Storing Requests 30
3.2 Routes Calculation 31
3.3 Requests Life-cycle 33
3.4 Finalizing the Request 34

Let's now present the solution proposal. The main service architecture will be presented, in order to accomplish a software solution for a DRT service.

The service aims to connect passengers who need to get from A to B, to providers that can perform such a trip. The main service features are: (a) providing automatic routes calculation for incoming passenger requests (for providers), and (b) submitting a passenger request, and checking its status (for passengers).

The conceptual solution architecture is proposed as follows:

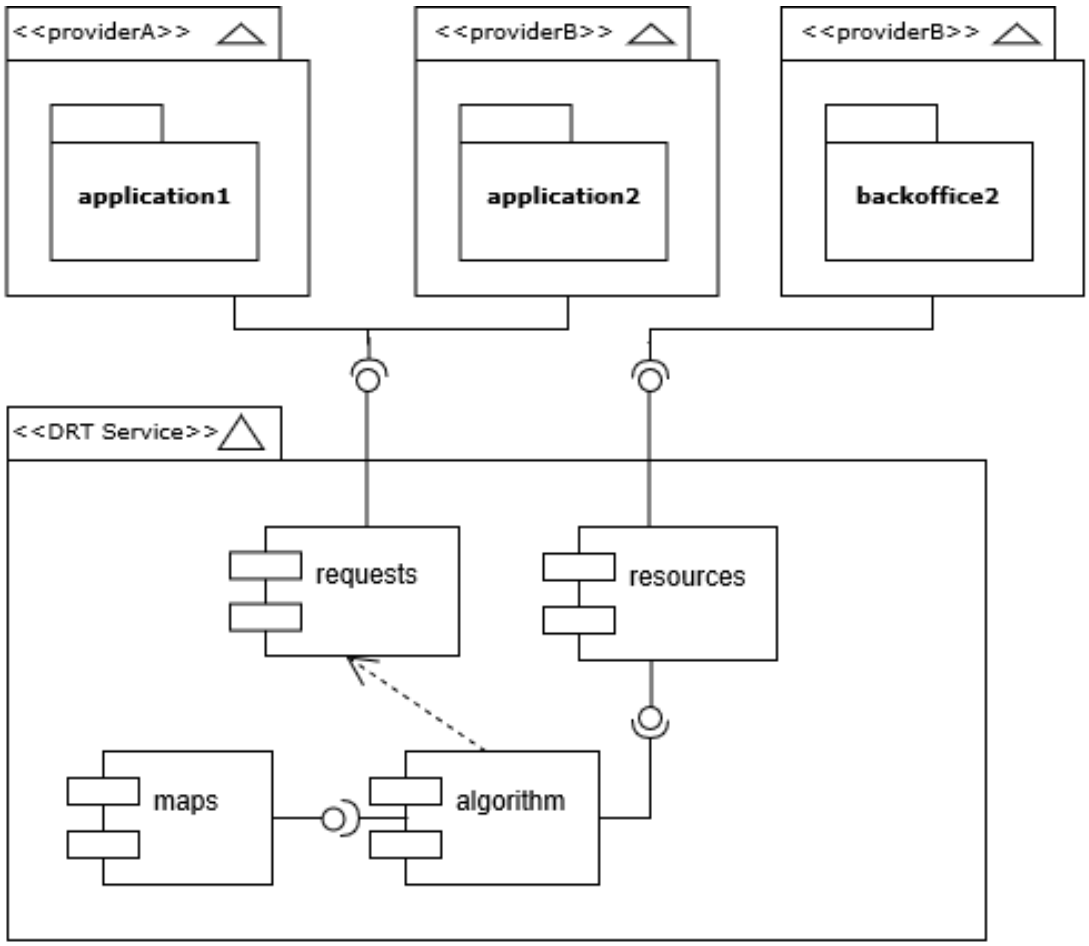

Figure 3.1: Structure Diagram of the Solution Modules.

Before a provider can start fulfilling requests, they should use the shown **Resources** module to input and update their data, such as the area(s) of activity, the fleet, the drivers, and the schedule/availability of these. To do this, the provider should use their back-office interface (shown by **backoffice2** for **Provider B**).

A provider can have multiple, non-overlapping activity areas, and multiple providers can be active in the same area, meaning areas can overlap, as far as they are from different providers.

Additionally, providers should inform the resources module of the vehicles' position regularly (either from their back-office, or from an external system that calls our **Resources** module). This information will be relevant for the algorithm.

Then, it is ready to process requests. Request processing is divided into three steps:

1. Storing requests
2. Calculating best routes
3. Finalizing requests and responding

The service might get requests from different providers, in a variety of ways and sources (such as web applications or mobile applications). To make it available to all kinds of sources, an API will be built. This API will contain a main endpoint to receive trip requests, and store them. This description is illustrated in Figure 3.1, by different applications (**Application A** and **Application B**) calling our **Requests** module.

Then, it is not desirable to send one vehicle per each request. As such, a processing algorithm will process all current requests and calculate the best routes for them. If a passenger cancels a request in the meantime, it is removed from the original pool and archived. If the request gets confirmed, resources are dispatched to fulfill it. The assigned vehicle is then informed of the attribution.

Finally, when the passenger gets picked up and delivered, the trip ends and the request is completed successfully.

The request storage and route calculation steps are illustrated in Figure 3.2.

In Figure 3.2, the **Passenger** actor is represented in Figure 3.1 by **application2**, as well as the interaction between respectively the **Requests** module and the **Algorithm** module.

Within this process, the **Algorithm** module makes use of **Maps** and **Resources** modules internally.

3.1 Storing Requests

Request storage is intended to be a fast and simple process. As such, the endpoint should only get requests, and persist them in the desired format to a queue. No calculations nor processing are necessary at this point. Yet, the request should be validated for any incoherence.

Each request of service must contain (among other information) these main components:

- departure point - latitude and longitude coordinates
- arrival point - latitude and longitude coordinates
- number of passengers

Passenger requests' points can also include altitude, but this is not used in the maps module for distances calculations. The identification of the provider area must also be there.

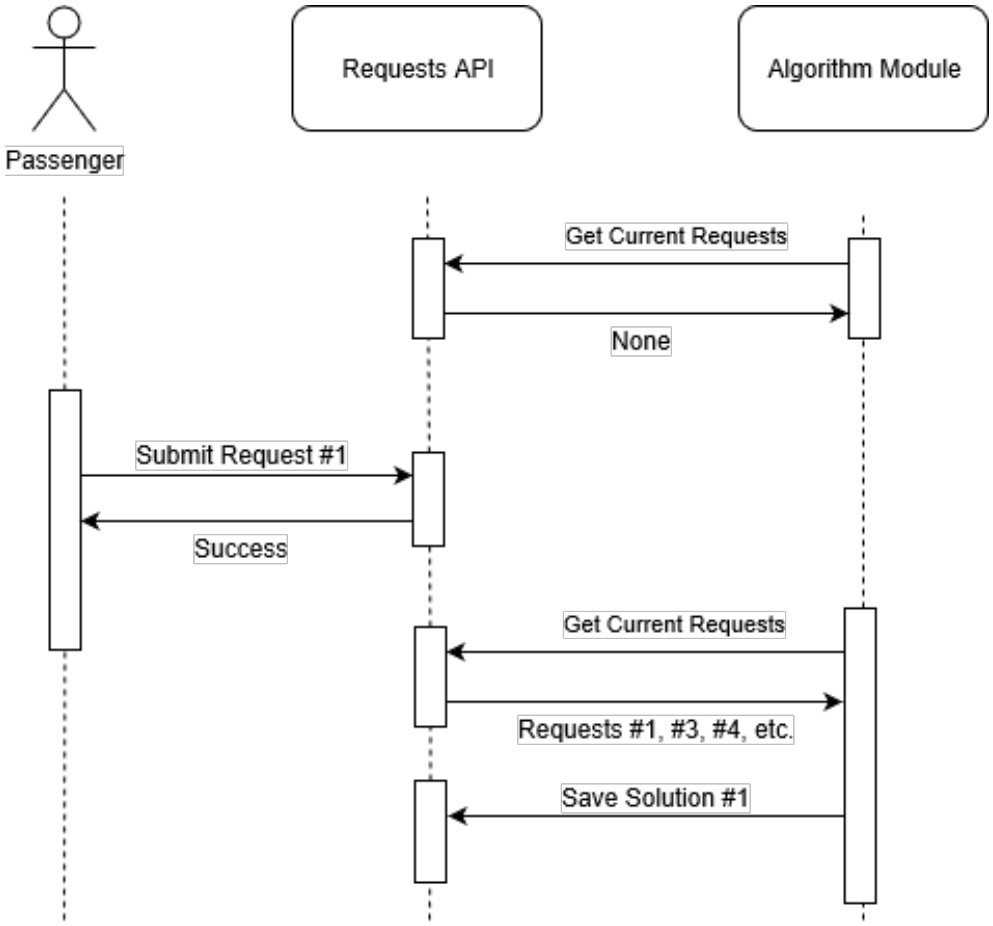

Figure 3.2: Flow of storing a request, and calculating routes.

3.2 Routes Calculation

Calculating the best combination of routes to satisfy passengers requests is a very important part of this work. Perhaps the most challenging, as it is an Optimization Problem[1].

This step can be divided in two main parts: (a) extracting information from the **Resources** and **Maps** modules (consulting maps, traffic information, fleet availability and tracking, etc.), and (b) providing an optimized solution to the current requests, given these inputs.

As such, a pool of worker tasks gets a portion of the requests, defined by provider and activity area. Then, each worker will perform **Route Calculation** for these requests.

3.2.1 Reducing Trip Stops

The first thing that should be done when calculating the routes is reducing the number of stops, by condensing close points in the aggregate (which might either be departure or arrival points) into a single

[1] Online: https://en.wikipedia.org/wiki/Optimization_problem. Last accessed: 1 May 2019

point. A threshold should define a maximum radial distance between the points and a common centroid. Providers should specify a threshold to be used by each worker.

The goal is to reduce the number of distance calculations between points that are close enough (reduce the size of the problem), while serving more passengers in a single trip. This same trip also becomes faster (less time to conclude), as making less stops reduces travel and waiting times.

Some providers might want to specify that departure and arrival points are fixed (like bus stops), for which this step of reducing is not necessary, and requests' points can only be chosen from these fixed stops.

3.2.2 Inputs

Knowing that each provider has a fleet operating in a geographic area, and requests to fulfill, it is necessary to have sources of information that tell us:

- how to determine the paths between points, and how travel times might be affected (e.g. traffic congestion, traffic lights, known blockages, etc.) - **maps**
- the capacity, availability and position of the vehicle fleet - **resources**
- the requests to fulfill, which already include departure and arrival points, the number of passengers, and the tolerance options, or fixed stops - **requests**

Each of these information sources will be abstracted in its own module, in order to better distribute responsibilities across the solution.

Requests will consist in the described service for receiving, storing, and updating status of requests.

Maps will condensate close points (or take the fixed stops), with given geographical and real time information; and produce a graph - the chosen world representation. In this graph, a vertex is either the current position of a vehicle, or a stop point. Each stop point will contain its pickups and deliveries. An edge represents the distances between points, containing physical distances, and estimated time distances.

Additionally to data sources, the algorithm will require the genetic operators to be passed as inputs, in order to easily modify and switch them,without changing the main algorithm.

Some of these inputs are **internal** - to be created along with this work, in a back-office API. Others are **external** - data or services from third-parties.

Resources, **options**, and **requests** are **internal** inputs, while **maps** is an external source, due to its complexity to implement internally, and requirements of accuracy.

3.2.3 Outputs

The algorithm is meant to produce a solution set. Each item of the solution is composed by:

- the optimized (ordered) route, containing a distance matrix (of time and length)
- the vehicle/driver pair that will perform the route

There may be vehicles not used in the proposed solution. As such, the solution set might contain less elements than the number of available vehicles in the fleet.

3.2.4 Algorithm

Given a graph (provided by the maps module, combined with the vehicles' positions of the resources module), a fleet (given by the resources module), and an evaluation function, this module is responsible to find the best solution possible to the current requests.

As said before, in this work, we chose to implement a genetic algorithm to optimize routes. As such, we will also need to provide the genetic operators that we want to use.

3.3 Requests Life-cycle

This way, requests are divided into six states: **Unprocessed**, **Confirmed**, **Started**, **Finished**, **Canceled**, and **Abandoned**, which diagram is shown in Figure 3.3.

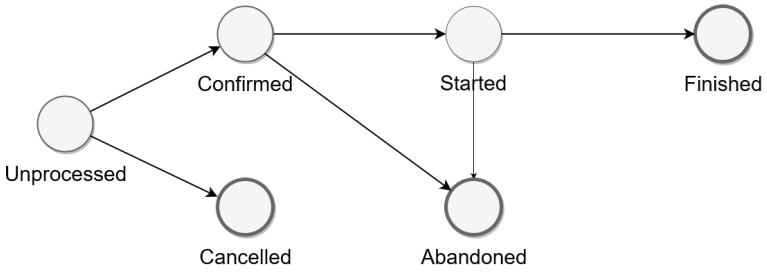

Figure 3.3: Diagram of a Request Life-Cycle. Thicker borders indicate Final States.

A new request is stored as **Unprocessed**, remaining in this state until solution is given to the passenger. Then, they decide either to confirm or cancel the request, transitioning to the **Confirmed**, or **Canceled** state respectively.

If a request is canceled, it is removed from the queue and archived. **Canceled** is a final state. If it gets confirmed, the process continues. If the customer does not confirm or cancel, the request gets automatically canceled after a specified timeout.

When the assigned vehicle reaches a **Confirmed** request departure point, the request should pass to **Started**, not being considered anymore for solution calculations (the passenger is already in the vehicle, and no connections are allowed).

Finally, when the vehicle reaches the request arrival point, the request passes to **Finished**, being now fulfilled. This is also a final state, and the most desired one.

If, by any means, a passenger does not respect the departure or the arrival points (within the stipulated time-line), their request should be marked as **Abandoned**. This is also a final state.

Providers might want to associate a reason for abandonment, which should be provided by the driver to the service. A passenger may also mark their own request as **Abandoned** voluntarily, providing a reason as well.

Each time a request changes its state, an audit should be created with the state change, and date of transition. The audits data might be useful for various legal or analysis reasons.

3.4 Finalizing the Request

Independently from which final state was reached, the customer can rate the service, placing the rating evaluation. This will measure the satisfaction level, which will be crucial to the evaluation and evolution of the service.

4

Technologic Arquitecture

Contents

4.1 Resources 38
4.2 Maps 41
4.3 Algorithm 42
4.4 Requests 45
4.5 Users, Authentication and Authorization 49

Now that we have gone through the main architecture, let's jump into the implementation in detail. The **Requests** and the **Resources** modules are made to receive requests from outside. As such, they are both Representational State Transfer (REST) Web API modules.

They are separated as a matter of purpose. **Resources** is responsible for the providers' information management, and **Requests** for the passengers'. And, for information security matters, as well as for a better API scope definition, these pieces of information are stored separately.

The **Maps** module is only a connector to an external maps API, where we translate input and output information between this external API and our own service. As such, this is implemented as a code library module.

Finally, the **Algorithm** module is just an algorithm, with specifications on the problem representation, constraints, and goals. It also contains connectors to receive the inputs from both **Requests** and **Resources** modules, yet it does not depend on them. This algorithm runs inside a scheduled job, which runs alongside with the **Requests** API Web server. This choice was made since both will require the same treatment of escalation (both depend on the number of requests received to process, and entities managed are the same).

This job runs one worker thread with an instance of the same algorithm per each provider area.

The described architecture is built as follows:

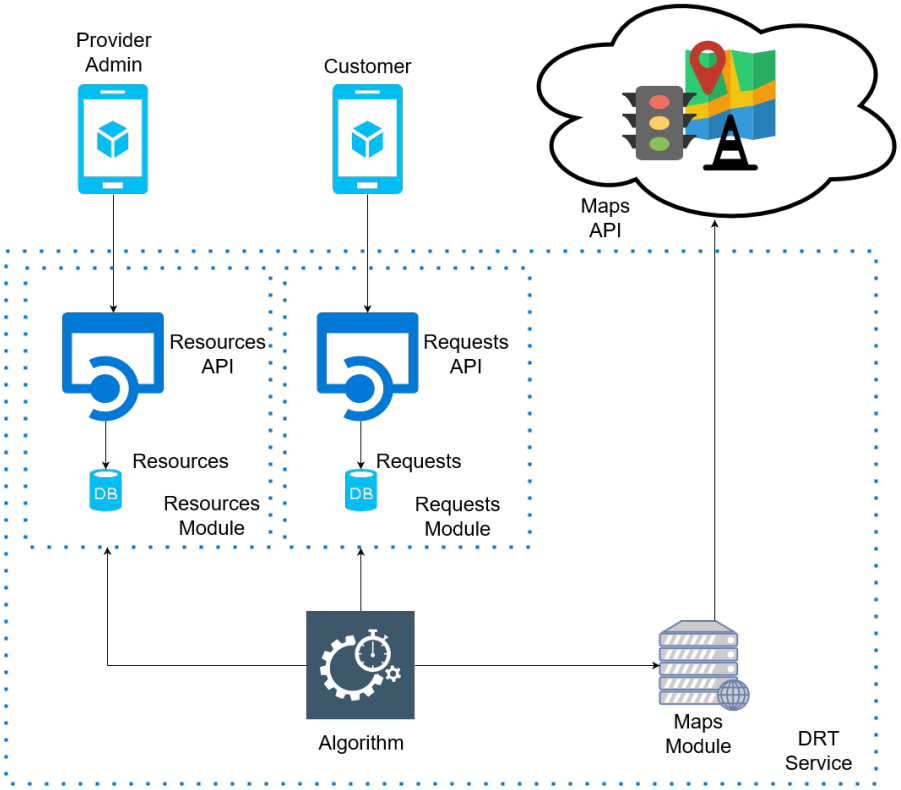

Figure 4.1: The Proposed Solution Technological Architecture.

All the modules were implemented using the C# language, and the .NET Core Framework[1]. The two Web API modules make use of SQL Server[2] databases. Some of the modules make use of other libraries, which will be described in their own sections.

Known the architecture, the modules descriptions follow. The next sections will show each module in detail, with all their functionality.

4.1 Resources

As we have defined above, the **Resources** API is a Web API, made to store and manage providers' information. This information is divided into five models:

1. Providers and Areas
2. Vehicles
3. Drivers
4. Availabilities
5. Trackings

The API is implemented using the Model-view-controller (MVC) design pattern[3], using C#

Each model operations defined in its own controller. Views are not really pages the user will see (as this work's scope does not include user interfaces), but rather Data transfer objects (DTO), or collections of those.

The entities are structured in the following schema in figure 4.2.

4.1.1 Providers and Areas

A provider is defined by a unique identifier, its name, and the areas of activity. An area contains its own identifier, another identifier for the provider it belongs to, a number that indicates the mean service time (the average time a vehicle takes on each stop), and a collection of edges that compose the area's polygon (they must define a closed polygonal line, without intersections).

Optionally, provider areas can define a tolerance distance, being the distance for which the algorithm will consider points are too close to make multiple stops, and therefore condense them into a single stop. It is also possible, yet optional, to define exception areas inside the area, or define a collection of

[1].NET, Online: https://en.wikipedia.org/wiki/.NET_Core. Last accessed: 1 May 2019
[2]SQL Server, Online: https://www.microsoft.com/en-us/sql-server. Last accessed: 1 May 2019
[3]Online: https://en.wikipedia.org/wiki/Model–view–controller. Last accessed: 1 May 2019

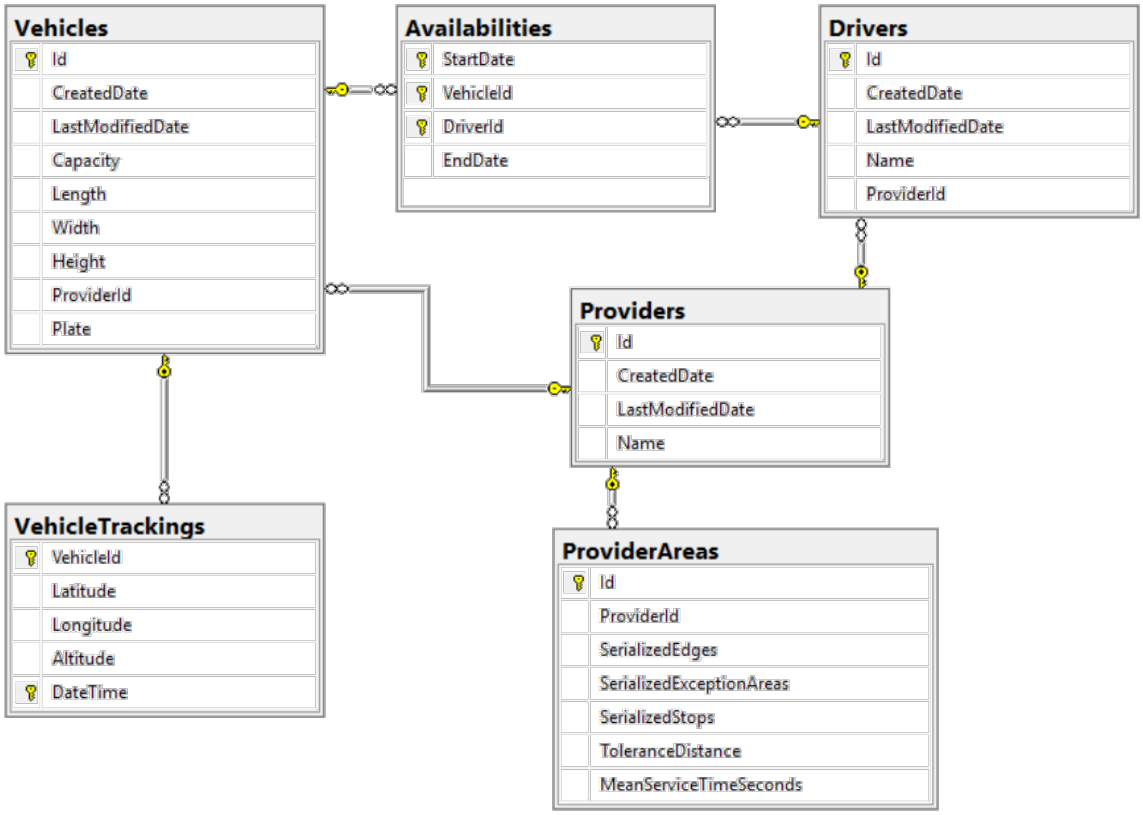

Figure 4.2: The Resources Module Entities Schema.

fixed stops, the only possible stop points for this area. It is only possible to define either fixed stops, or a tolerance distance, not both.

Exception areas are areas too, to represent enclaves[4]. As such they must be defined inside their container area (this is validated using the ray casting algorithm[5]). They can also contain their own exception areas.

Let the level of a provider area be 0. Its exception area will be level 1, and its exception areas exception areas level 2, and so on. At an odd level, it is not possible to define tolerance distance nor fixed stops (it does not make sense in the context of the service), being only possible at an even level. In this work we allow providers to define exclaves inside enclaves, and trips to cross an enclave to reach an internal exclave (an exclave inside an enclave, inside an area).

4.1.1.A Calculating intersections on an Ellipsoid

Commonly, computer systems use spherical representations of the earth to calculate distances between points more easily.

[4]Online: https://en.wikipedia.org/wiki/Enclave_and_exclave. Last accessed: 1 May 2019
[5]Online: https://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm. Last accessed: 1 May 2019

This is known to introduce a small error, though, specially for larger distances, according to Geographic Information Systems Stack Exchange[6]. In order to reduce distance calculation and/or line intersection errors to a minimum, we want to represent the Earth as an ellipsoid of revolution (or a slightly oblate spheroid)[7], defined by $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$, where (a = b).

Due to the impossibility of determining simple closed geodesics, except for the equator and the meridians, we cannot use great-circle formulas to easily calculate distances, nor great-circle intersections. In this work, we explored the Vincenty's formulae [37].

To calculate intersections, we used a hybrid method, that calculates planar intersection (a flat projection of the space, using vector dot product and matrix determinants) for edges shorter than ten kilometers (10 Km). For larger ones, we used a method to solve Vincenty's inverse problem[8]. This way, we guarantee a very high precision on intersections calculation, and also guarantee a fair balance between high precision and computing resources usage.

These intersections will allow to determine if a point is inside a polygon (Ray Casting Algorithm), or if two polygons overlap (used in areas validation).

To note that these validations are only used in stages where the system does not require high performance in terms of time.

4.1.2 Vehicles

A vehicle is defined by a unique identifier, another identifier of the provider it belongs to, its license plate, capacity, and measures (length, width and height).

4.1.3 Drivers

A driver has a unique identifier, another identifier of the provider it belongs to, and its name.

4.1.4 Availabilities

An availability is defined by a driver identifier, a vehicle identifier, a start date and time, and an end date and time.

The start date must be before the end date. It is possible to define multiple associations with the same driver and vehicle, but they cannot overlap, as one vehicle can only be driven by one driver at a time, and a driver can only drive one vehicle at a time. Drivers can have multiple availabilities on the same day, being the times not available representative of their breaks. There are no limits for an

[6]Geographic Information Systems, Online: https://gis.stackexchange.com/questions/25494/how-accurate-is-approximating-the-earth-as-a-sphere. Last accessed: 1 May 2019

[7]Wikipedia, Online: https://en.wikipedia.org/wiki/Spheroid. Last accessed: 1 May 2019

[8]Online: https://en.wikipedia.org/wiki/Vincenty%27s_formulae#Inverse_problem. Last accessed: 1 May 2019

availability duration, or to a maximum number of daily available hours for drivers and vehicles. Each provider might have to obey existent laws respective to these data, which might differ by county, city, or range of activity (long-range trips might require more than a day, amongst other exceptional scenarios).

4.1.5 Trackings

A tracking is defined by the vehicle identifier, the position of this vehicle (latitude, longitude, and optionally altitude), and the date and time for which the vehicle was in this position. Trackings can repeat position at different times for the same vehicle, but the same vehicle cannot have two trackings at the same exact time (regardless of their position).

4.1.6 API Endpoints

The API has endpoints to Create, Update, Delete, and Retrieve (both individually and in filtered collections) models. Creations are made through the Hypertext Transfer Protocol (HTTP) POST verb, Updates through the PUT verb, Deletions use the DELETE verb, and retrievals through the GET verb.

The endpoints listing and requirements are shown in Chapter 5, in Section 5.1.3, where we show the tests and validations done.

4.2 Maps

This module is very small. A code library that contains mainly connectors to two different maps API: Openrouteservice[9] and HERE[10].

Both connectors have the same functionality: When called, they request the respective maps API for a matrix of routes, in order to obtain the shortest paths between points. The returned matrix has the travel distances and times. Each connector transforms its inputs (the stop points) into an API request, uses an HTTP client to send it and get the response, and finally parses the response into a known internal representation (a map).

This step is done after the point condensation. As such, each point represents a stop.

At first, the Openrouteservice was used in this work, since it is free to use and crowd-sourced. But, with a deeper investigation on this type of API, we concluded that HERE would provide an equally free tier of usage, with an extra: HERE includes traffic congestion and other external factors in travel times, while Openrouteservice is static over time (travel times only depend on speed limits and distance).

[9] Openrouteservice, Online: https://openrouteservice.org/dev/#/api-docs/matrix/. Last accessed: 1 May 2019

[10] HERE, Online: https://developer.here.com/documentation/routing/topics/request-matrix-of-routes.html. Last accessed: 1 May 2019

HERE also allows splitting the matrix into smaller matrices (being the limits on 15 start points, and 100 destination points[11]).

Knowing these differences, we chose HERE over Openrouteservice.

4.3 Algorithm

This module contains the routes optimization algorithm for a DARP. As such, it takes the inputs (requests, a map, and fleet of vehicles), and tries to fit them in the best way possible, following these constraints:

- all requests must be satisfied exactly once (no requests can be left, and it can only be served by one vehicle)
- all requests time windows should be respected (pickup time should be between a request minimum and maximum pickup time, and the delivery time should be between the request minimum and maximum delivery time)
- the number of passengers inside a vehicle cannot ever exceed its capacity
- every request pickup must be before its delivery (already done in time windows, by definition, but still a constraint)
- all vehicles' availability should be respected (a vehicle has a time window of activity, and should not serve requests out of it)
- the distance traveled and time of travel should be the shortest possible

To accomplish such an algorithm, we chose to use a Genetic Algorithm, because the performance of genetic algorithms is good for this kind of problems, and there is much investigation already done about them in the literature to help building an acceptable one. As this is an initial version of the service, the algorithm doesn't have to perform as good as in the literature. It just needs to be acceptable for most cases.

This DARP solver algorithm is defined in Algorithm 1 (pseudocode).

The algorithm starts by validating if there are any requests to serve. If not, simply return an empty solution. After this, we use the **Resources** module to get the fleet. If this retrieval is not successful, we return an empty solution, and log an error.

Thirdly, we determine the required stops either by retrieving the provider area fixed stops (if existent), or by selecting the distinct points from all requests, and condensing them making use of the provider area tolerance distance.

[11] Online: https://developer.here.com/documentation/routing/topics/resource-calculate-matrix.html. Last accessed: 1 May 2019

Algorithm 1 DARP Solver

```
procedure SOLVE(providerArea, requests)
    if requests.isEmpty then
        return ∅
    end if
    providerId ← providerArea.providerId
    fleet ← GetFleet(providerId, providerArea, requests)
    if fleet.isEmpty then
        return ∅
    end if
    stops ← GetStops(providerId, providerArea, requests)
    depots ← fleet.lastTrackings
    points ← concat(stops, depots).distinct()
    map ← GetMap(points)
    if !map.isValid then
        return ∅
    end if
    d ← providerArea.toleranceDistance
    mt ← providerArea.meanServiceTime
    solution ← FindSolution(requests, fleet, stops, map, d, mt)
    return solution
end procedure
procedure GETSTOPS(providerId, providerArea, requests)
    if providerArea.hasFixedStops then
        return providerArea.GetFixedStops()
    end if
    points ← requests.points
    d ← providerArea.toleranceDistance
    stops ← CondensePoints(points, d)
    return stops
end procedure
```

The condensation step is done using Hierachical Clustering [12] (with complete linkage distance between clusters) on points, with the tolerance distance as the maximum cluster size equal to the tolerance distance.

The process continues by using the **Maps** service to create a map of the distances between points all points (the required stops and the vehicles' current positions).

If we are unavailable to retrieve a valid map from its module, again we log an error, and return an empty solution. Otherwise, a **Genetic Algorithm** is ran to optimize the routes. This Genetic Algorithm is shown in Algorithm 2.

Algorithm 2 Genetic Algorithm

procedure FINDSOLUTION($requests, fleet, stops, map, d, mt$)
  $solutionSpaceSize \leftarrow count(fleet)^{count(requests)}$
  $population \leftarrow InitialPopulation(populationSize)$
  $currentBest \leftarrow population.Best(FitnessOperator)$
  $degenerations \leftarrow 0$
  while $degenerations < maxDegenerations \wedge time < maxTime$ **do**
    $population \leftarrow population.RemoveWorst()$
    while $population.size < populationSize \wedge solutionSpaceSize > populationSize$ **do**
      $parent1 \leftarrow SelectionOperator(population)$
      $parent2 \leftarrow SelectionOperator(population)$
      $child1 \leftarrow CrossoverOperator(parent1, parent2)$
      $child2 \leftarrow CrossoverOperator(parent2, parent1)$
      $population \leftarrow population \cup \{child1, child2\}$
    end while
    $population \leftarrow MutationOperator(population)$
    $generationBest \leftarrow population.Best(FitnessOperator)$
    if $generationBest > currentBest$ **then**
      $currentBest \leftarrow generationBest$
      $degenerations \leftarrow 0$
    else
      $degenerations \leftarrow degenerations + 1$
    end if
  end while
  return $currentBest$
end procedure

Finally, the best solution found is returned.

4.3.1 Representation and Genetic Operators

As mentioned in Section 2.4, there are plenty of possible representations, as well as genetic operators. The chosen representation is a set of pairs vehicle-request, where order is taken care by solving a local TSP for each route. We chose this representation to narrow the solution space the most - to only contain the possible associations between any subset of the vehicles, to all of the requests.

[12] Online: https://en.wikipedia.org/wiki/Hierarchical_clustering. Last accessed: 1 May 2019

The chosen fitness operator is the following function:

$$x_{fitness} = x_{quality} \times \frac{1}{x_{totaldistance} + 1}$$

Where $x_{quality}$ is a number between 0 (lowest quality) and 1 (best quality), calculated based on the constraints x violates, and how much they are violated (i.e. time windows can be violated by one minute, or by hours, applying different discounts to the overall quality).

This function is a simple one (heavy calculations may prejudice the whole algorithm, without great benefit of optimization), that takes all constraints into account, and optimizes routes for the desired goal.

The selection operator is a Roulette Based Selection method, with Rank-Based probabilities. As for the crossover operator, the PMX crossover was chosen. Lastly, we chose a single-gene swap mutation method.

All representation, fitness function and genetic operators choices were based on the literature reviews made, and are the ones to be reviewed for algorithmic improvements in the future (see Section 6.2).

4.4 Requests

This is the Web API passengers use to create trip requests, update them, check their status, what are the current solutions proposed to accomplish the trip, and finally rate it.

As the name indicates, this module only has two entities: Requests and Solutions, which are structured in the following schema in figure 4.3.

4.4.1 Requests

This is the central entity of the whole project, alongside with solutions. A request has a unique identifier, another identifier for the provider area where it should be served, yet another identifier for the user (the passenger), the points of the trip, and four time stamps: minimum and maximum departure dates, and minimum and maximum arrival dates (which will make the time windows for this request). Additionally, a number of passengers can be specified (a positive integer).

4.4.1.A Validation

In order to submit a valid request, some conditions must be met:

- the number of passengers is mandatory, and must be a integer, greater or equal to one
- all four dates must be provided with valid date and time values (ISO 8601 compliant)

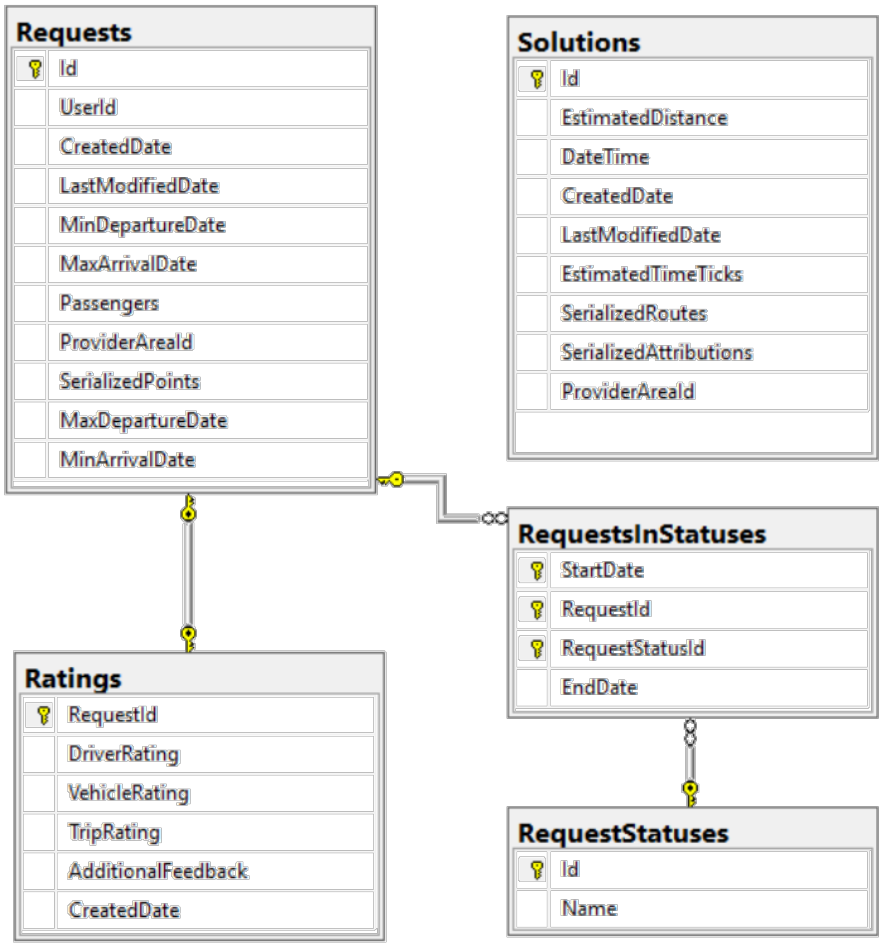

Figure 4.3: The Requests Module Entities Schema.

- at least two points must be provided, with valid geographic coordinates (latitude in interval]-90,90], longitude in interval]-180, 180])
- all points must be different (cannot coincide)
- the user that is submitting the request must be registered, logged into the service, and have a valid provider identifier associated to them
- there cannot be any other requests ongoing from this same user (users cannot make simultaneous requests)
- a valid provider area identifier must be provided, for the user's provider only
- all of the request point must be inside the chosen provider area
- if the provider area has defined fixed stops, all points must coincide exactly with any of these stops

- if not, all of the request points must be farther apart than the provider area tolerance distance, otherwise they would get condensed

4.4.1.B Operations

Apart from submitting a request (done in route POST /requests), some other operations are possible.

A user can retrieve all their requests, retrieve one specific request, get the route preview for one specific request, and update its state.

The state of a request can only be updated if it conforms to the possible transitions. These are described in section 3.3.

When a request is in the **Finished** state, it is also possible to rate it. Each request can only be rated once.

If the user is an administrator, they can also retrieve all requests from a specific provider area.

4.4.2 Solutions

This entity is the answer given by the processing algorithm to a set of requests. A solution has an identifier, another identifier for the provider area it is being calculated in, a time stamp for which it was calculated, a set of routes, a set of request-vehicle attributions, the estimated total distance, and the estimated total time.

Each route is a key-value pair, where the key is the vehicle that will perform the route, and the value is an ordered list of pairs, each containing a point and an expected time stamp. These compose the expectation for passengers to know when they will be picked up, and delivered.

Each attribution is a key-value pair too, where the key is the same vehicle, and the value is a set of requests that will be fulfilled by that vehicle.

4.4.2.A Solvers, Job, and Solution generation

Solutions are produced by a solver, that runs inside a job. This job runs alongside with the Web API, as they need to meet the same performance goals, as the number of requests grow.

This job runs one instance of a solver per each provider area. This separation is due to requests from different provider areas cannot be fulfilled by the same fleet of vehicles. It just wouldn't make sense to mix them, as areas of the same provider are probably far away from each other, and providers won't share their requests with their competition.

These solvers are ran periodically, with a configurable amount of time between runs. A solver is an implementation of the algorithm module, that will do the optimization work.

At the end of a run, each solver returns and saves a solution in the database.

These solutions allow a passenger to check the current route preview (which can get updated if a new solution is calculated meanwhile), based on the last solution calculated in their provider area for that request.

An improvement that was started, but could not be finished in this work, is a second solver that runs when a passenger requests a new trip, and tries to fit it into the last solution calculated without it. This is described in Section 6.2

4.4.3 Request States

As described in section 3.3, requests have states, which can be final or not.

The initial state is **Unprocessed**, then it can go to either **Confirmed**, and continue the process, or **Cancelled** (final state). From **Confirmed**, a request can get **Started** (when the passenger is picked up), and continue the process, or **Abandoned** (if the passenger does not show up at the indicated time and place). **Abandoned** is also a final state. Finally, a **Started** request can go to **Abandoned** (if the passenger quits the trip before getting to the destination), or to **Finished** (when the passenger trip is complete).

Figure 3.3 shows this behaviour.

Requests get associated with states, by a **RequestsInStatuses** entry. These contain an identifier of the request, another identifier of the request state, a start date, and an optional end date (when not specified, this indicates the current state). RequestsInStatuses entries cannot overlap for the same request, and every time a request changes its state, the previous association's end date is updated with the time stamp of the change. A new entry is then added for the new association, with the same time stamp as start date.

4.4.4 Ratings

A rating represents the user evaluation of a request. As described before, a user can rate a request, by providing the request unique identifier, the ratings values, and optionally an additional feedback text.

The ratings values are the evaluations of:

- the driver
- the vehicle
- the trip

The driver evaluation should take care of the relation with the passengers, the quality of driving, the presentation of the driver, among other aspects a passenger might find important.

The vehicle evaluation is related to its conditions and comfort.

Finally, the trip evaluation measures the overall experience, especially if the waiting times were acceptable, the routes previews were accurate, and other things the passenger values.

The additional feedback is an important way for both the service and the platform to improve. Compliments might also be interesting! A rating cannot be updated once submitted.

4.4.5 API Endpoints

Just like the **Resources** API, this API has endpoints to Create, Update, and Retrieve (both individually and in filtered collections) models. Creations are made through the HTTP POST verb, Updates through the PUT verb, and retrievals through the GET verb.

There are no deletions, as it is not possible to delete a request once it is made. It is only possible to cancel, or abandon it.

The endpoints listing and requirements are shown in Chapter 5, in Section 5.1.2, where we show the tests and validations done.

4.5 Users, Authentication and Authorization

Security play a major role in most systems. As such, we have given a little attention to it, and added a basic level of security to both of the API modules.

All of the API modules run under the Hypertext Transfer Protocol Secure (HTTPS) protocol (instead of HTTP). This way, all messages are encrypted, being almost impossible to decipher them in useful time.

Users can also change their password, an operation they should perform regularly.

Apart from this, some other operations are to be made available. These are defined in section 6.2.

4.5.1 Users

There are two types of users: **Passengers** and **Administrators**. A passenger is the public-use account type, that allows a user to use the Requests API. An administrator is a back-office management account, that can manage all providers' information, from the area to the vehicle tracking (Resources API user).

A passenger can register in the service just by providing an email, user name and password. And this is done through the Requests API.

To register as an Administrator, a user must register through the Resources API, and provide not only an email, user name and password, but also a known authorization token.

4.5.2 Authentication/Authorization

Both of the API modules are built to accept JSON Web Tokens (JWT) as the way of authorizing requests. These tokens can contain all information regarding a user's role and permissions (encrypted), being a very flexible way to do authorization management.

A user gets a token by logging into the system, and the session is active until the token expires. It is possible to ask for another token before the first expires, being the second one active, and the old one is automatically expired by the system.

As previously said, a token has a due time, when it expires and becomes useless.

5

Evaluation

Contents

5.1 Validations: By Module and Integrated 53

5.2 Service Behaviour 63

5.3 Discussion 66

Evaluating the implemented solution is an important part to know where there is room for improvement.

We have done some validations and testing for our software solution, both separately by module, and together as a whole system. As this project is mainly a software solution for a DRT service, the focus of the evaluation will be on the fulfilled requirements and features.

Additionally, we have used the service for small studies, where we tried to observe the behaviour of such a service in different contexts. These studies show a poor performance for real-world application of the service, reason why there is much learning and many suggestions for future work on it.

5.1 Validations: By Module and Integrated

As we have divided our solution into modules, so did we for the methods to validate them. These methods range from simple API calls, to more complex request submissions to assert different validation rules. Additionally, we have tested some edge cases for all modules.

The **Maps** module is the easiest to evaluate, as it is the smallest. We have ran some tests making use of both services (Openrouteservice and HERE), with different parameters, to check their behaviour.

As for the API modules (**Requests** and **Resources**), we have used Postman [1] to make web requests, and manually test all the endpoints with different input data, and observe the API responses.

We have used the algorithm with some simple cases to validate its behaviour.

Finally, we have made some integration tests on the whole service, to verify its functionality.

5.1.1 Maps

In order to test this module, we had to test if it would make the requests to the respective maps API, and parse the responses correctly.

Openrouteservice was simple to test: we simply provided a pair of points, and confirmed it would return a 1x1 matrix back. After this, we tried more request with n points, each returning a n-1 x n-1 matrix.

The HERE service was similar, with another feature to be tested: the matrix split. To accomplish this, our maps connector had to calculate the sub-matrices and make the proper requests. To do this, the connector was provided two configuration values: (1) the maximum origins limit per-request, and (2) the maximum destinations limit per-request. Different values were tried, with both odd and even numbers bigger than 1, and finally with 1 in either and both of the limits. All tests returned the correct matrices, being the matrix splitting and joining mechanism correct.

[1] Online: https://www.getpostman.com/. Last accessed: 1 May 2019

The evaluation of this module shall reside on its ability to handle failures, and on its correctness. To note that the connector should not handle exception cases by itself (like connection issues), but rather throw exceptions, to be handled on the caller (the code that uses the connector, in this case, the **Algorithm** module).

5.1.2 Requests

As an API module, the **Requests** module was tested using Postman. From this application, one is able to create and maintain request specifications and collections. It is also possible to specify small scripts to validate the responses gotten from the API to be tested.

In this case, the endpoints to be tested are listed in Table 5.1.

Table 5.1: Requests API Endpoints. Note: the numbers represent identifiers of the entities

Endpoint	Verbs	Observations
/requests	POST, GET	Create and retrieve requests
/requests/1	PUT, PATCH, GET, DELETE	PUT always returns HTTP Code 405 (Method not allowed) PATCH can only change request state
/requests/1/preview	GET	Route preview of a request
/requests/1/rate	POST	Rate a request
/requests/provider-area/2	GET	Only accessible to Administrator Users 2 is the provider area identifier

All endpoints require proper authentication. All unauthenticated requests will get an HTTP 401 code (Unauthorized), and if a User tries to access any resource that does not belong to them, they will get an HTTP 404 (Not Found), as a security measure for hackers not to know which resources exist.

5.1.2.A POST /requests

To test this endpoint, we have used different requests, in order to hit the different validations:

1. invalid time windows (both date formats and precedence)
2. minimum departure time window in the past
3. number of passengers lower than 1
4. less than two points (none, or just one)
5. points with invalid geographic coordinates (longitude lower than -180, or higher than 180, latitude lower than -90, or higher than 90, negative altitude)
6. exactly coincident points
7. points out of the provider area, or in one of its exception areas

8. if the provider area has fixed stops, points that are not a stop

9. if the provider area has a tolerance distance, points closer than this distance

All of these requests get an HTTP 400 (Bad Request). Aside from validations, we tried to submit a request for a user that already has an ongoing request. For this case, we get an HTTP 409 (Conflict). For the success case, the user gets an HTTP 201 (Created).

5.1.2.B GET /requests

This endpoint only has a couple possible tests:

1. getting requests as a Passenger User

2. getting requests as an Administrator User

The common expected results are a collection of visible requests to that user, with its associated states history, and ratings (if rated). The only difference is on visibility. Administrator Users can see all requests from all providers, while Passenger Users can only see their own requests.

5.1.2.C PUT /requests/1

Only one simple test: try to hit the endpoint with different values, and it will always respond with HTTP 405 code (Method not allowed), as it is not possible to replace the content of a request, only **Cancel** it, and submit another.

5.1.2.D PATCH /requests/1

This is a very simple endpoint: it only changes the state if a request. As such, the only tests to be made here are: as a Passenger User, try to access a request that does not exist (HTTP 404 Not Found), or that is owned by another Passenger (HTTP 404 Not Found); and try to change a request to an invalid state (HTTP 400 Bad Request), or to a state to which this request cannot transit (e.g. from Unprocessed to Finished). The expected response is again an HTTP 400 (Bad Request). If one submits a valid request with a valid state change, they get an HTTP 204 (No Content).

5.1.2.E GET /requests/1

Get a request by its id. If the request is visible to the current user, return an HTTP 200 (OK) with the details of the request. Otherwise, they get an HTTP 404 (Not Found).

5.1.2.F DELETE /requests/1

Similarly to previous the endpoint, delete a request by its id. If the request is visible to the current user, return an HTTP 200 (OK) with the details of the request. Otherwise, they get an HTTP 404 (Not Found).

5.1.2.G GET /requests/1/preview

This endpoint retrieves the current route preview for the given request. The possible tests are only on request visibility (HTTP 404 Not Found if not visible or nonexistent), and on whether a preview is available or not. If the algorithm had enough time to produce a preview, the response is an HTTP 200 (OK), with the route preview, and the information on the driver and the vehicle. If a preview is not yet available, the response is an HTTP 202 (Accepted), with a time estimate for the preview to be available.

5.1.2.H POST /requests/1/rate

After a finishing a trip, a passenger can use this endpoint to rate it. Again, visibility tests were made (HTTP 404 Not Found if not visible, or nonexistent). Also tried to rate an already rated request, giving an HTTP 409 (Conflict). Additionally, tried to rate a request that was not in the **Finished** state, also giving an HTTP 409 (Conflict). Finally, the success case (rating an unrated finished request) results in an HTTP 201 (Created).

5.1.2.I GET /requests/provider-area/2

Finally, this endpoint is only accessible for Administrator Users, to retrieve all requests from a given provider area. The only tests made were on authorization: tried to make the request as a Passenger User, resulting in an HTTP 403 (Forbidden); and with an Administrator User, resulting in an HTTP 200 (OK) with the correct collection of requests.

5.1.3 Resources

This is also an API module, reason why we also used Postman to test this module's endpoints. The list is given in Table 5.2.

Just like in the **Resources** module, all endpoints require proper authentication, this time from an Administrator User, who can access and manage all providers' information. All unauthenticated requests will get an HTTP 401 code (Unauthorized).

The following sections illustrate the tests made for all endpoints of each group.

Table 5.2: Resources API Endpoints. Note: the numbers represent identifiers of the entities

Group	Endpoint	Verbs	Observations
Providers	/providers	POST, GET	
	/providers/1	PUT, GET, DELETE	
Vehicles	/vehicles	POST, GET, DELETE	
	/vehicles/1	PUT, GET, DELETE	
	/vehicles/provider/2	GET, DELETE	2 is the provider identifier
Drivers	/drivers	POST, GET, DELETE	
	/drivers/1	PUT, GET, DELETE	
	/drivers/provider/2	GET, DELETE	2 is the provider identifier
Availabilities	/availabilities	POST, PUT, GET, DELETE	
Trackings	/trackings	POST, PUT, GET, DELETE	

5.1.3.A Providers and Areas

A – POST /providers This is perhaps the most complex endpoint of the whole project. In order to correctly submit a provider, many validations have to be made, especially in its areas of activity. The tests done were:

1. a provider with no areas
2. a provider without a name
3. a provider with an area with invalid geographic coordinates (same validations as in requests points)
4. a provider with an area that is not a polygon (open polygonal line, or closed polygonal line with intersections)
5. a provider with an area with an exception area that is not a polygon (open polygonal line, or closed polygonal line with intersections)
6. a provider with overlapping areas
7. a provider with two equal areas
8. a provider with an area, with an exception area that overlaps or is outside the main area
9. a provider with an area, with two overlapping exception areas
10. a provider with an area with an exception area that has fixed stops, with one stop inside an exception area, or one stop outside of the main area
11. a provider with an area with an exception area with a level 2 exception area (an enclave in an exclave), with all fixed stops inside this level 2 enclave, and one inside the exclave (level 1)

All these tests result in an HTTP 400 code (Bad Request). Some success cases are:

1. a provider with a name, and a valid area with no exception areas and with a valid tolerance distance (positive number or zero)

2. a provider with a name, and a valid area with valid exception areas, with a valid level 2 exception area, and the main area has some fixed stops, and the level 2 enclave has other fixed stops

We do not limit the exception areas maximum level of depth. The only limit is that fixed stops should be in an even level (odd levels are areas that are not part of the main area). As such, we could end up with an arbitrarily high number of possible cases. For all of them, the API responds with an HTTP 201 code (Created).

B – GET /providers For this endpoint, no validations other than authorization are made. It always returns HTTP 200 with a collection of all providers with their areas to authorized Administrator Users.

C – PUT /providers/1 This is a very similar endpoint to POST /providers. The same validations are done, with the only addition of verifying that the provider previously exists. If it doesn't, an HTTP 404 (Not Found) is given. Otherwise, an HTTP 204 (No Content) is returned.

D – GET /providers/1 Gets a provider by its identifier. If existent, returns an HTTP 200 (OK), with the provider details. Otherwise, gives an HTTP 404 (Not Found).

E – DELETE /providers/1 Similarly to the previous endpoint, deletes a provider by its id. If existent, returns an HTTP 200 (OK), with the deleted provider details. Otherwise, gives an HTTP 404 (Not Found).

5.1.3.B Vehicles

A – POST /vehicles In order to test vehicle creation, the following tests were made:

1. empty license plate
2. negative capacity
3. any the measures (width, height, length) negative
4. nonexistent provider identifier

For all of these cases, the API returns an HTTP 400 code (Bad Request). For the success case, an HTTP 201 (Created) is given.

B – GET /vehicles Gets all vehicles from all providers. Always returns an HTTP 200 (Ok), with a collection of vehicles, to authorized users.

C – PUT /vehicles/1 Replaces an existent vehicle. This endpoint has the same validations as POST /vehicles, with the only addition of validating the vehicle existence (returns HTTP 404 Not Found if nonexistent, HTTP 204 No Content if existent).

D – GET /vehicles/1 Gets a vehicle by its identifier. The only validation here is existence of the vehicle.

E – DELETE /vehicles/1 Similarly to GET /vehicles/1, this endpoint verifies the vehicle existence, deletes a vehicle if existent, returning an HTTP 200 (OK); and returns an HTTP 404 (Not Found) if nonexistent.

F – GET /vehicles/provider/2 This endpoint gets all vehicles by provider identifier, and always returns an HTTP 200 (OK) with a collection of vehicles to authorized users.

G – DELETE /vehicles/provider/2 Similarly to GET /vehicles/provider/2, deletes all vehicles of a given provider. If the provider does not exist, or has no vehicles, an HTTP 404 (Not found) is given. Otherwise, and HTTP 200 (OK) is returned, with a collection of the deleted vehicles.

5.1.3.C Drivers

This group is very similar to the Vehicles group. The only differences reside in the model (in this case, the driver) validations.

The validated conditions are:

- a driver must have a (non-empty) name
- a driver must have a valid provider identifier (existent)

All remaining tests and endpoints are equal to Vehicles'.

5.1.3.D Availabilities

A – POST /availabilities This endpoint is used to create an availability. The tests done on validations are the following:

1. an availability with the start date equal or after the end date
2. an availability with a nonexistent vehicle identifier
3. an availability with a nonexistent driver identifier

4. an availability that conflicts with other availabilities of the chosen vehicle or of the chosen driver
5. an availability that picks a vehicle from one provider, and a driver from another provider

All mentioned cases, except the last one, give an HTTP 400 (Bad Request), while the last gives an HTTP 409 (Conflict). It is considered a conflict when there is an availabilty for the same vehicle with a different driver on the chosen time interval, or when there is an availability with the same driver and another vehicle. In the case the availability we want to save ”conflicts” with another, with the same vehicle and driver, we consider it as an extension. For this case, we pick the union of the availability with all of its extensions. There are two possible success scenarios: (1) the mentioned extensions case with no conflicts, and (2) an availability with no conflicts nor extensions. These cases give an HTTP 201 (Created), with the final availability details.

B – PUT /availabilities Replaces an availability. This endpoint is all in all similar to the POST /availabilities, with the difference of the extensions case. For this case, we remove all existent extensions, and replace them with the give availability. The success cases return an HTTP 204 code (No Content). An extra validation checks if there are no extensions at all. For this case, an HTTP 404 (Not Found) is given.

C – GET /availabilities Returns all availabilities (an HTTP 200 OK), filtered by the input criteria:

- provider identifier
- vehicle identifier
- driver identifier
- time interval (which is inclusive if partially matched)

All criteria are optional. The only validations are: (1) the time interval start must be before its end; and (2) if provided, the provider, driver, and vehicle identifiers must exist. An HTTP 400 (Bad Request) is returned when any of these validations fail. Otherwise, an HTTP 200 (OK) is given with a collection of the filtered availabilities.

D – DELETE /availabilities This endpoint follows the same filter logic as GET /availabilities. Returns an HTTP 200 (OK) if the filter has any availabilities, and an HTTP 404 (Not Found) otherwise.

5.1.3.E Trackings

Trackings have a compound key, made by the vehicle identifier and a time stamp. These are the only thing that distinguish vehicle trackings.

A – POST /trackings Creates a vehicle tracking point. The only validation done here is if there is any existent tracking for this vehicle at the same exact time (to a precision of a tick), or if the geographic coordinates are invalid (similarly to the previous endpoints with coordinates validation). For these cases, an HTTP 400 (Bad Request) is given. Otherwise, it gives an HTTP 201 (Created).

B – PUT /trackings This endpoint is all in all similar to the POST /trackings, with the existence validation reversed: if nonexistent, return an HTTP 404 (Not Found), and an HTTP 204 (No Content) for the success case. The coordinates validation is the same.

C – GET /trackings Similarily to GET /availabilities, this endpoint gets all vehicle trackings, filtered by the input criteria:

- provider identifier
- vehicle identifier
- time interval

All criteria are optional. The only validations are: (1) the time interval start must be before its end; and (2) if provided, the provider and vehicle identifiers must exist. An HTTP 400 (Bad Request) is returned when any of these validations fail. Otherwise, an HTTP 200 (OK) is given with a collection of the filtered trackings.

D – DELETE /trackings This endpoint has the same filtering behaviour and GET /trackings. If the filter returns no tracking, an HTTP 404 (Not Found) is given. Otherwise, gives an HTTP 200 (OK) with a collection of the filtered trackings.

5.1.4 Algorithm

The algorithm module had more attention to testing, as we have produced a battery of automated tests for it. This battery is divided into two main groups, being the first the edge and simplest cases, and the second the performance tests.

In the first group, we have tested the following cases:

1. null or empty fleet
2. null or empty requests
3. null or invalid map
4. single request, single vehicle case

5. single request, two vehicles case

The first 3 cases result in an empty solution, as the algorithm does not have the necessary inputs to run. The fourth case results in a single route attributed to the only vehicle available, with only the request points (two, in this case). Lastly, the two vehicles case results in a single route, attributed to the closest vehicle to the request's points (same request as in the previous case, with same points and same time windows).

As for the second group, we have added some well known literature test cases, like the Cordeau and Laporte [38] DARP instances, and OVRP instances from Christofides's 1979 book [39] and Fisher [15].

These were used for two purposes: validating the correctness of the algorithm, and studying its behaviour for improvements and learning. In all cases, despite not giving optimal solutions, the algorithm does correctly the optimization work.

In section 5.2, we describe the behavioural studies in more detail.

5.1.5 Integration

After verifying all modules separately, we have ran a few tests to verify the whole system. To do this, we started both **Requests** (which already includes the algorithm job) and **Resources** services, and shot requests at both of them.

First, we have used **Resources** API (with the success cases described in section 5.1.3) to create the providers' required information for operation in our DRT service. Then, we submitted a trip request for one of these providers, and observed that, in its next run, the algorithm started to calculate routes for it, and produced a correct solution (Image 3.2 in Chapter 3 illustrates this behaviour). After this, we have **Cancelled** the request. The algorithm then correctly recognized there were no ongoing requests, and done nothing (returned an empty solution). We followed with another similar request to the same provider, and immediately asked for its route preview. The API responded with a correct time estimation, based on the time left for the algorithm to finish running. After this time, we retried the preview request, and got the correct answer with the correct route. We then tried to rate this request, but it was not finished yet. The API responded with an HTTP 409 code (Conflict). We then updated the request all the way to the **Finished** state, and rated it correctly. Lastly, we verified that when we retrieve this request, all states history and the rating are shown.

A couple more validations were made, like submitting requests for different providers, observing that the job processes them in separate worker tasks, and that the solution were correctly determined and persisted.

5.2 Service Behaviour

In order to study the algorithm behaviour, we have used the mentioned literature cases, under different constraints. We have also done a small study, comparing some trips on the Lisbon Metro, with the suggested routes by the algorithm.

For both experiments, the algorithm's solutions feasibility is the $x_{quality}$ variable mentioned in Section 4.3.1.

5.2.1 DARP vs. OVRP

Simply by changing a problem's constraints, the algorithms can make some assumptions. In this case, A DARP has, at least, a pair of points per request, while in a classic OVRP, a request and a point are one and the same.

To observe our service's behaviour on both problems, we ran it, using the mentioned Cordeau and Laporte instances for the DARP, and using the Christofides and Fisher instances for the OVRP.

And, despite giving poor solutions when compared with the literature best known solutions (with gaps ranging from the 30-60% on the DARP to 3000% on the OVRP), we were able to observe that our algorithm is suitable to run for both. The results are shown in tables 5.4 and 5.3. To note that it is possible that new researches might have discovered a new Best Known Solution (BKS) for these instances. But for this work, we are not so much focused on algorithmic performance, we only wanted a fair measure to learn more about road transportation science, and better design an architecture for such a service.

These tables columns are defined as follows:

- **Total Distance** - the total traveled distance by all vehicles (in Kilometers)
- **BKS** - the total distance of the BKS
- **#Vehicles** - the number of vehicles used (our solution)
- **#Requests** - the problem size (number of requests)
- **CPU Time** - the time given to the algorithm to run (based on the time the original authors took to reach the BKS). The chosen format is mm:ss,ms
- **Feasibility** - the reached solution quality, based on the constraints defined in Section 4.3.1
- **Gap** - given by $gap = \frac{TotalDistance}{BKS} \times 100$

To note that we have ran each instance 5 times, and all values, except the **BKS** column are averages of all runs.

Table 5.3: OVRP Test Instances Performance

Instance	Total Distance (Km)	BKS (1979)	#Vehicles Used	#Requests	CPU Time (m:s)	Feasibility	Gap
C01	1533,680289	408,50	31	50	00:03,5	96%	375%
C02	2311,967241	567,14	50	75	00:11,1	78%	408%
C03	3224,482216	622,13	65	100	00:21,2	96%	518%
C04	4725,552062	733,13	96	150	00:32,5	95%	645%
C05	6174,800981	879,37	124	200	00:55,5	84%	702%
C11	7689,307490	682,12	10	120	00:16,2	100%	1127%
C12	3741,015977	534,24	62	100	00:26,2	83%	700%
C13	7612,083477	896,50	75	120	00:29,2	100%	849%
C14	3743,236469	591,87	65	100	00:15,2	89%	632%
F11	5520,079864	175,00	42	71	00:06,1	100%	3154%
F12	6094,395107	769,66	85	134	00:23,3	74%	792%

Table 5.4: DARP Test Instances Performance

Instance	Total Distance (Km)	BKS (2003)	#Vehicles Used	#Requests	CPU Time (m:s)	Feasibility	Gap
R01a	311,6165	190,02	3	24	00:36,0	92%	61%
R02a	551,2076	302,08	5	48	00:14,2	88%	55%
R03a	1096,8120	532,08	7	72	00:20,3	87%	49%
R04a	1269,7030	572,78	9	96	00:34,0	86%	45%
R05a	1398,9350	636,97	11	120	00:41,5	86%	46%
R06a	1893,3630	801,40	13	144	00:40,5	85%	42%
R07a	500,2368	291,71	4	36	00:15,5	89%	58%
R08a	1031,4650	494,89	6	72	00:26,9	86%	48%
R09a	1523,2400	672,44	8	108	00:35,0	84%	44%
R10a	1993,6260	878,76	10	144	00:54,0	82%	44%
R01b	303,5191	164,46	3	24	00:26,4	94%	54%
R02b	564,9256	296,06	5	48	00:20,2	92%	52%
R03b	1090,1370	493,30	7	72	00:17,7	89%	45%
R04b	1278,8590	535,90	9	96	00:31,6	90%	42%
R05b	1419,6150	589,74	11	120	00:38,0	89%	42%
R06b	1869,5200	743,60	13	144	00:45,3	88%	40%
R07b	498,4887	248,21	4	36	00:11,7	93%	50%
R08b	1034,9540	462,69	6	72	00:24,6	88%	45%
R09b	1509,2430	601,96	8	108	00:34,4	87%	40%
R10b	1976,4850	798,63	10	144	00:59,2	85%	40%

The OVRP instances, having less constraints, have a higher probability of reaching feasible solutions inside the solution space, making it easier for our algorithm to reach one in less time. Yet, our algorithm is more suited to find better solutions for the DARP than it is for the OVRP (based on gap values).

Some of the DARP instances are well-known to have no 100% feasible solutions (without breaking any constraints), and are very good instances to study an algorithm performance.

These observations support the reviewed literature, as the more constraints we add to a problem, the harder it gets to find feasible, or even optimal solutions.

From here, in future work, one of the main challenges will be to improve the algorithmic performance, especially for the DARP cases, as they will be the most similar (among the studied ones) to real-world passenger scenarios.

Yet, the current is enough to observe the topological differences of solutions spaces for different problem constraints.

5.2.2 Lisbon Metro vs. DRT Service

To test a more real-world scenario, we have used a small subset of data with metro stations entries an exits, simulating ground-truth trips made by passengers.

The subset used for this study has 237 trips (requests), from 00:30h to 01:00h of one measured day.

Without any BKS for reference, we tested how the number of vehicles impacted the given solutions. We created 3 test instances then: LX01 (237 requests, 5 vehicles), LX02 (same 237 requests, 10 vehicles), and LX03 (same 237 requests, 20 vehicles).

Table 5.5 shows the results gotten, having its columns defined as follows:

- **Total Distance** - the total traveled distance by all vehicles (in Kilometers)
- **#Vehicles** - the number of vehicles used (our solution)
- **#Requests** - the problem size (number of requests)
- **CPU Time** - the time given to the algorithm to run (based on the time the original authors took to reach the BKS). The chosen format is mm:ss,ms
- **Feasibility** - the reached solution quality, based on the constraints defined in Section 4.3.1
- **Travel Time** - the sum of all vehicles route time (being each route time its end time minus its start time)

Table 5.5: Lisbon Metro Test Instances Performance

Instance	Total Distance (Km)	#Vehicles Used	#Requests	CPU Time (m:s)	Feasibility	Travel Time (Days)
LX01	2768351	5	237	05:00,0	77%	4,628819
LX02	2706737	10	237	04:23,2	72%	4,565458
LX03	2657615	20	237	01:39,7	70%	4,545259

We can observe a slightly constant total traveled distance, and a decrease in CPU times. More vehicles seem to make the algorithm reach the same solutions faster. Despite the feasibility percentage decreases with more vehicles, we do not believe this decrease is representative, as values are close enough. Also, total travel times are also more or less constant, leading to the observation that more vehicles can deliver the same amount of passengers faster. For **LX01**, the average route lasts about 22 hours, while for **LX03** the average route lasts 5 hours.

These observations support the idea that for higher demand environments, like city centers, the service should get properly scaled in order to be appealing, as more vehicles can deliver passengers faster.

This idea is also supported by the previous experiment results (in Section 5.2.1), as the number of requests has a positive correlation with the number of vehicles used.

More vehicles represent a higher operational cost, though. It is necessary to find the smallest number of vehicles that can deliver all passengers, respecting all time and capacity constraints.

The 237 passengers were easily transported in half an hour by the Lisbon metro, while our service suggests much higher delivery times. It is difficult to compare the delivery times of a regular metro with a DRT bus service, yet some more studies with a better algorithm might bring very interesting conclusions.

5.3 Discussion

In this section, we will go through the results gotten from the validations and small experiments made, and draft some of the findings we have made. Mostly, we will present learning achievements, which should be followed with a deeper work.

From the validations section, it is observable that a lot of the work done in this work is centered in offering providers a broad range of solutions, and almost all of the features desired for such a service. To also be noted the level of detail on entities validations and definition.

From the behavioral studies section, we mainly observe learning on our side, that supports the investigated authors' work. This service might be very useful for academic research in the transportation science field, and the architecture is built to easily adapt to different kinds of business (either by using a different navigation system, or by using a different algorithm for a different business model, with different constraints).

New studies could be done on the applicability of such a service to replace the Lisbon night buses, which work in fixed routes, just like daytime regular service. Such a study would have to be backed with information on the current usage of these buses. For now, there is only the observation that it is frequent that these buses are empty in many periods of the route, and a DRT service could optimize their occupation.

6

Conclusion

Contents

6.1 Conclusions 69

6.2 System Limitations and Future Work 70

This chapter is to resume all the work done, the learning involved, the conclusions reached, and how this work should continue improving in the future.

Firstly, we will present our conclusions, based on the project development, the investigations, and the experimental results obtained. And lastly, we will present a set of improvements to be done on this project.

6.1 Conclusions

We think a very flexible architecture was achieved, making it possible in the future to adapt the service to different providers, or even to new types of service. This is a good asset for such a project, where the business model is not tightly defined.

The results obtained in Chapter 5, when comparing our service to the Lisbon Metro, lead us to the conclusion that such a service should focus their activity in less accessible zones, or in less crowded periods of the day. This way, operation costs of a regular service would benefit from the savings, and be appealing enough for passengers.

Yet, this is not a final conclusion. More studies, with a better algorithmic solution could improve a lot the results, and is a very interesting work to be done in the future.

The problem constraints study has the main conclusion that different providers should have different algorithms, specially tuned for their business logic.

Another possible conclusion is that the geography of the provider areas also has an impact on which should be the best business model for a provider. This conclusion comes from the observation of the impact of time windows tightness on the probability to generate good solutions (R1a to R10a have tighter time windows, and lower probability of resolution, while R1b to R10b have more relaxed time windows).

Finally, despite this work not being centered in algorithmic performance of the service, it is important for more accurate studies in the transportation science field.

In summary, the work developed covers all the proposed features, with a higher level of architecture flexibility and detail than was initially intended. Most local providers, known from the first contact made with a company, would not even require that much quality of features in this work. The project needs future contributions, yet I am personally very satisfied with the learning and code base achieved. So much, that there is the intention to work in this field in the future.

The idea of adapting this architecture to delivery drones has emerged. This type of service is not public transport, is only package transport. Yet, if we consider that drones pick packages from one point (pickup), and deliver them to another point, that, and that these pickups and deliveries have time windows, the problem constraints are exactly the same.

6.2 System Limitations and Future Work

As mentioned before, this is a very flexible architecture, that could benefit with some more diversity in some of the modules, in order to adapt them to the different business models that might be required.

The only system limitation found resides in the algorithm module. In a scenario where a request is in the **Started** state, and algorithm runs again, both the request pickup and delivery points are considered, instead of just the delivery. This might lead to wrong results in a real-world scenario, for providers that accept new requests, while the bus is already on route. The algorithm will consider both points, possibly being stuck on some repeated points already visited.

The following sections will describe some other interesting improvements to be made, divided into **New Features** - improvements on the service features or on the architecture, **Algorithm Improvements** - on performance and correctness, and finally **New Studies** - proposed studies in the transportation field.

6.2.1 New Features

There are some interesting improvements that can be done to the implementation, in order offer even better features, at an almost professional level.

One of these improvements would be a worker task, that runs periodically, and cancels requests that are **Unprocessed** for too long. The time limit could be defined per provider area. This feature would remove probably forgotten requests, cleaning up for the algorithm to run only on interesting requests.

The next phases of this project should now also include user interfaces, in order to allow tests with real users.

Another thought feature is to provide an endpoint that, given a vehicle and a time interval, compares the executed route (from the vehicle trackings) to the algorithm preview. This would help faster improvements on the algorithm, being a tool to evaluate its quality.

It is also interesting the idea of, when storing a new request, running a lighter algorithm that takes the current vehicle routes, and tries to fit the new request in any of them, without breaking any constraints. If successful, the heavy algorithm does not have to run so often, raising the service adaptability to new requests, and users can preview their trips faster.

In the security field, there are also some room for improvement, with two-factor authorization of operations (especially in the administration back-office), possibility to register or login from external accounts (like Facebook, Microsoft or Google accounts), and a ”Forgot my credentials” endpoint, that sends an email or an SMS to the user with a temporary token to reset their password.

6.2.2 Algorithm Improvements

The main improvement to be made is the algorithm performance, and also the most difficult to achieve.

From the investigations done on Genetic Algorithms, it is suggested that a better crossover operator would raise the probability of generating feasible solutions, making the algorithm evolution process faster, and richer. An idea is to use the Best-Cost Route Crossover (BCRC), proposed by Ombuki et. al [40].

Other ideas include the mentioned Tabu Search, and a Gravitational Search Algorithm, as proposed by Hosseinabadi et. al [41]. These new methods could be interesting for comparative studies, and possibly be combined for different providers, if futures studies conclude that none of them outperforms another for all situations.

The purpose of all these suggestions is to improve the algorithm module, making it suitable for more demanding tests, and explore different approaches for different providers.

6.2.3 New Studies

As mentioned in Section 5.3, some new studies are suggested.

The first is in order to assert the possibility of using a service like this to replace the Lisbon night buses (or at least some of them). The goal would be to maximize vehicles' usage, and minimize traveled distances. If possible, would be interesting that the service could also reduce passengers' waiting times.

Another interesting study would be to use a more broad portion of the Lisbon metro data set, and check if the determined vehicle routes converge to the metro lines courses. This would assert the quality of their distribution on the city. There is a popular idea that their planning is not so good, and such a study could help with the discussion.

Yet another proposed study is to use Lisboa Aberta data sets [1], a municipal open data initiative, to generate real-world test scenarios, using the pedestrian densities (normalized) as probabilities of getting a request for that location. The pedestrian densities are visible in the Potential Pedestrian Map (Mapa de Potencial Pedonal) [2].

Finally, we proposed a business idea: applying a discount to **Started** requests' passengers every time a route changes with impacts on their time windows. This idea would require a profitability study, and attractiveness inquire from the public.

[1] Online: http://lisboaaberta.cm-lisboa.pt/index.php/pt/dados/conjuntos-de-dados. Last accessed: 2 May 2019

[2] Online: http://planoepap.maps.arcgis.com/apps/webappviewer/index.html?id=069c4784a209443ca98ede37e9fb5867. Last accessed: 2 May 2019

Bibliography

[1] Li, F., Golden, B., Wasil, E.: The open vehicle routing problem: Algorithms, large-scale test problems, and computational results. Computers & Operations Research **34**(10), 2918–2930, (2007)

[2] Directive 2014/23/EU of the European Parliament and of the Council of 26 February 2014 on the award of concession contracts (OJ L 94, 26.2.2014, p. 1–64)

[3] Rodrigue, J.P., Comtois, C., Slack, B.: The Geography of Transport Systems. 3rd edn. Routledge, New York (2013)

[4] Dantzig, G. B., Ramser, J. H.: The Truck Dispatching Problem. Management Science **6**(1), 80–91 (1959)

[5] Baldacci R., Battarra M., Vigo D.: Routing a Heterogeneous Fleet of Vehicles. In: Golden B., Raghavan S., Wasil E. (eds) The Vehicle Routing Problem: Latest Advances and New Challenges. Operations Research/Computer Science Interfaces, vol 43, pp. 3-27 Springer, Boston, MA (2008)

[6] Toth, P., Vigo, D.: Models, relaxations and exact approaches for the capacitated vehicle routing problem. Discrete Applied Mathematics **123**(1–3), 487–512 (2002)

[7] Yousefikhoshbakht, M., Didehvar, F., Rahmati, F.: A Mixed Integer Programming Formulation for the Heterogeneous Fixed Fleet Open Vehicle Routing Problem. Journal of Optimization in Industrial Engineering **8**(18), 37–46 (2015)

[8] Desaulniers, G., Desrosiers, J., Erdmann, A., Solomon, M. M., Soumis, F.: VRP with Pickup and Delivery. In: Desaulniers, G., Desrosiers, J., Erdmann, A., Solomon, M. M., Soumis, F.: The Vehicle Routing Problem. Society of Industrial and Applied Mathematics, Philadelphia, PA, pp. 225–242 (2002)

[9] Fan, J.: The Vehicle Routing Problem with Simultaneous Pickup and Delivery Based on Customer Satisfaction. Procedia Engineering, **15**, 5284–5289 (2011)

[10] El-Sherbeny, N. A.: Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods. Journal of King Saud University - Science **22**(3), 123–131 (2010)

[11] Savelsbergh, M. W. P., Sol, M.: The General Pickup and Delivery Problem. Transportation Science **29**(1), 17–29 (1995)

[12] Huang, M., Hu, X.: Large scale vehicle routing problem: An overview of algorithms and an intelligent procedure. International Journal of Innovative Computing, Information and Control **8**(8), 5809–5819 (2012)

[13] Rissanen, K.: Kutsuplus -– Final Report. Helsinki Regional Transport Authority, Helsinki (2016)

[14] Vehicle Routing Problem — NEO Research Group, http://neo.lcc.uma.es/vrp/. Last accessed 10 May 2019

[15] Fisher, M. L.: Optimal Solution of Vehicle Routing Problems Using Minimum K-trees. Operations Research **42**(4), 626–642 (1994)

[16] Araque G., J. R., Kudva, G., Morin, T. L., Pekny, J. F.: A branch-and-cut algorithm for vehicle routing problems. Annals of Operations Research **50**(1), 37-–59 (1994).

[17] Fisher, M. L., Jaikumar, R.: A Generalized Assignment Heuristic for Vehicle Routing. Networks **11**(2), 109–124 (1981)

[18] Ryan, D. M., Hjorring, C.: Extensions of the Petal Method for Vehicle Routeing. Journal of the Operational Research Society **44**(3), 289–296 (1993)

[19] Taillard, É.: Parallel iterative search methods for vehicle routing problems. Networks **23**(8), 661–673 (1993)

[20] Renaud, J., Boctor, F. F.: A Sweep-Based Algorithm for the Fleet Size and Mix Vehicle Routing Problem. European Journal of Operational Research **140**(3), 618–628 (2002)

[21] Beasley, J. E.: Route first–Cluster second methods for vehicle routing. Omega **11**(4), 403–408 (1983)

[22] Clarke, G., Wright, J. W.: Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. Operations Research **12**(4), 568–581 (1964)

[23] Altinkemer, K., Gavish B.: Parallel Savings Based Heuristic for the Delivery Problem. Operations Research **39**(3), 456–469 (1991)

[24] Kinderwater, G. A. P., Savelsbergh M. W. P.: Vehicle Routing: Handling Edge Exchanges. In: Aarts, E. H. L., Lenstra, J. K. (eds.) Local Search in Combinatorial Optimization, pp. 337–360 . Wiley, Chichester (1997)

[25] Bullnheimer, B., Hartl, R. F., Strauss C.: An improved Ant System algorithm for the Vehicle Routing Problem. Annals of Operations Research **89**(0), 319–328 (1999)

[26] Dueck, G.: New Optimization Heuristics: The Great Deluge Algorithm and the Record-To-Record Travel. Journal of Computational Physics **104**, 86–92 (1993)

[27] Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. 1st edn. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1989)

[28] Glover, F.: Future Paths for Integer Programming and Links to Artificial Intelligence. Computers and Operations Research **13**(5), 533–549 (1986)

[29] Tan, K. C., Lee, L. H., Zhu, Q. L., Ou, K.: Heuristic methods for vehicle routing problem with time windows. Artificial Intelligence in Engineering **15**(3), 281–295 (2001)

[30] P. Larrañaga, C. Kuijpers, R. H. Murga, I. Inza, S. Dizdarevic: Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. Artificial Intelligence Review **13**(2), 129–170 (1999)

[31] Vaira, G.: Genetic Algorithms for Vehicle Routing Problem, Ph.D. dissertation, Vilnius University (2014)

[32] Kumar, R.: Blending Roulette Wheel Selection & Rank Selection in Genetic Algorithms. International Journal of Machine Learning and Computing, **2**(4), 365–270 (2012)

[33] Potvin, J.-Y., Bengio, S.: The Vehicle Routing Problem with Time Windows—Part II: Genetic Search. INFORMS Journal on Computing **8**, 165–172 (1996)

[34] Jorgensen, R. M., Larsen, J., Bergvinsdottir, K. B.: Solving the dial-a-ride problem using genetic algorithms. Journal of the operational research society **58**(10), 1321-–1331 (2007)

[35] Umbarkar, A.J., Sheth, P.D.: Crossover Operators in Genetic Algorithms: A Review. ICTACT Journal on Soft Computing **6**(1), 1083–1092 (2015)

[36] Hamzaçebi, C.: Improving genetic algorithms' performance by local search for continuous function optimization. Applied Mathematics and Computation **196**(1), 309–317 (2008)

[37] Vincenty, T.: Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations. Survey Review. **23**(176), 88-–93 (1975)

[38] Cordeau, J.-F., G. Laporte.: A tabu search heuristic for the static multi-vehicle dial-a-ride problem. Transportation Research Part B: Methodological **37**(6), 579–594 (2003)

[39] Christofides, N., Mingozzi, A., Toth, P.: The vehicle routing problem. In: Combinatorial Optimization, pp. 315—338. Wiley, Chichester, UK (1979)

[40] Ombuki, B., Ross, B., Hanshar, F. T.: Multi-Objective Genetic Algorithms for Vehicle Routing Problem with Time Windows. Applied Intelligence **24**(1), 17–30 (2006)

[41] Hosseinabadi, A.A.R., Kardgar, M., Shojafar, M., Shamshirband, S., Abraham A.: Gravitational Search Algorithm to Solve Open Vehicle Routing Problem. In: Snášel, V., Abraham, A., Krömer, P., Pant, M., Muda, A. (eds.) Innovations in Bio-Inspired Computing and Applications. Advances in Intelligent Systems and Computing, vol. 424, pp. 93–103. Springer, Cham (2015).

A

Tan et al. Heuristic Methods Comparison

Table A.1: Comparison between the results obtained by Tan et al. against historical best. NV: Number of Vehicles. TD: Total Distance. 2-INT: 2-interchange (Taillard's Algorithm). SA: Simulated Annealing. TS: Tabu Search. GA: Genetic Algorithm.

Problem	Best Known		2-INT		SA		Tabu		GA	
	NV	TD	NV	TD	NV	TD	NV	TD	NV	TD
C101	10	829	10	828.937	10	828.937	10	828.937	10	828.937
C102	10	827	10	923.375	10	923.375	10	901.527	10	868.798
C103	10	828.06	10	994.87	10	994.87	10	954.718	11	939.456
C104	10	824.78	11	1130.85	11	1130.85	10	895.774	10	963.72
C105	10	829	10	828.937	10	828.937	10	828.937	10	828.937
C106	10	827	10	1052.07	10	1052.07	10	941.154	10	828.937
C107	10	829	10	875.62	10	867.234	10	828.937	10	828.937
C108	10	827	10	878.089	10	876.427	10	828.937	10	828.937
C109	10	829	10	1100.71	10	1124.44	10	828.937	10	828.937
C201	3	590	3	591.557	3	591.557	3	591.557	3	591.557
C202	3	590	4	801.281	4	787.856	4	745.99	4	683.864
C203	3	591.55	4	1225.1	4	1208.94	4	727.221	4	745.934
C204	3	590.6	3	661.213	3	642.691	3	590.599	3	604.998
C205	3	589	3	625.333	3	613.327	3	588.876	3	588.876
C206	3	588	3	704.162	3	694.25	3	588.493	3	588.493
C207	3	588	3	725.404	3	704.365	3	600.841	3	593.195
C208	3	588	3	902.597	3	888.685	3	645.206	3	590.873
R101	18	1608	20	1847.37	20	1847.37	20	1707.95	20	1676.86
R102	17	1434	19	1720.46	18	1544.82	16	1488.59	18	1558.59
R103	13	1207	17	1551.34	17	1482.39	15	1293.85	15	1311.81
R104	10	982.01	12	1184.38	12	1184.38	11	1057.02	12	1128.29
R105	14	1377.11	17	1671.94	17	1595.68	16	1431.56	17	1496.37
R106	12	1252.03	14	1439	14	1434.3	14	1331.5	14	1357.19
R107	11	1126.69	13	1390.55	12	1270.04	12	1174.89	13	1240.82
R108	10	968.59	12	1256.61	12	1186.34	11	1039.34	12	1091.69
R109	11	1205	14	1525.45	14	1515.38	14	1256.36	15	1300.29
R110	11	1080.36	13	1444.63	13	1430.35	13	1179	13	1315.56
R111	10	1104.83	13	1371.64	13	1370.21	13	1148	12	1202.31
R112	10	953.63	12	1215.27	12	1180.18	11	1088.32	12	1097.64
R201	4	1354	5	1791.42	5	1726.13	5	1437.49	8	1329.74
R202	3	1530.49	4	1610.02	4	1581.64	5	1272.6	7	1307.03
R203	3	1126	4	1475.63	4	1248.55	4	1081.04	6	1086.43
R204	2	914	3	1098.67	3	1088.06	3	895.867	6	956.384
R205	3	1128	4	1370.68	4	1344.2	4	1150.34	5	1131.18
R206	3	833	3	1341.22	3	1369.5	4	1103.22	5	1187.25
R207	3	904	3	1167.11	3	1153.65	3	1007.3	4	1016.63
R208	2	759.21	3	988.367	3	971.572	3	806.797	3	845.937
R209	2	855	4	1210.96	4	1206.58	4	1110.3	5	1097.42
R210	3	1052	4	1312.91	4	1238.14	4	1071.3	6	1136.54
R211	3	816	3	1232.48	3	1140.65	3	946.354	7	932.483
RC101	14	1669	17	1948.94	17	1940.57	16	1734.17	17	1728.3
RC102	12	1554.75	16	1803.95	16	1777.02	14	1562.62	17	1603.53
RC103	11	1110	14	1627.02	14	1620.35	13	1377.93	14	1519.83
RC104	10	1135.83	12	1408.5	12	1408.5	11	1259.28	12	1276.02
RC105	14	1602	17	1920.19	17	1809.78	16	1597.67	17	1688.77
RC106	11	1448.26	14	1656.29	14	1645.24	14	1476.15	14	1491.58
RC107	11	1230.54	15	1677.92	15	1653.65	13	1392.97	14	1462.3
RC108	10	1139.82	12	1377.93	13	1335.06	12	1264.5	12	1333.15
RC201	4	1249	5	2070.4	5	1891.9	5	1617.5	10	1565.67
RC202	4	1221	5	1970.66	5	1956.97	5	1429.04	10	1353.27
RC203	3	1203	4	1627.74	4	1522.68	4	1179.67	6	1189.06
RC204	3	879	4	1314.21	4	1290.89	4	939.678	4	989.943
RC205	4	1389	5	1923.9	5	1907	5	1487.49	9	1465.8
RC206	3	1213	4	1663.87	4	1645.17	4	1357.32	5	1388.13
RC207	3	1181	4	1539.85	4	1497.54	4	1295.9	6	1304.48
RC208	3	919	3	1490.27	3	1422.94	3	1040.47	6	1003.43