TÉCNICO LISBOA

Dynamic Delivery Services for Smart Cities

Solving a Constrained Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment

Miguel de Sousa Esteves Martins

Thesis to obtain the Master of Science Degree in

Mechanical Engineering

Supervisors: Prof. João Miguel da Costa Sousa
Prof. Susana Margarida da Silva Vieira

Examination Committee

Chairperson: Prof. Paulo Jorge Coelho Ramalho Oliveira
Supervisor: Prof. João Miguel da Costa Sousa
Member of the Committee: Prof. Carlos Augusto Santos Silva

November 2017

Resumo

Nesta tese é proposto um modelo para serviço de estafetas capaz de lidar com o escalonamento de uma frota de veículos. Mais especificamente, é formulado um *Capacitated Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment*. Este problema surge de um problema do mundo real existente na companhia *Urban Dynamic Delivery*, que executa escalonamento e planeamento para serviços de estafeta. Este caso de estudo específico requer um intervalo de tempo curto e limitado para resolver o problema de otimização. Desta forma, em vez de se encontrar a solução óptima, independentemente do tempo computacional necessário, procura-se uma boa solução num pequeno intervalo de tempo. Um novo método emparelhando *Nearest Neighbourhood Search* com *Subtractive Clustering* é proposto para melhorar as soluções iniciais e acelerar a convergência do algoritmo de otimização. Além disso, um método híbrido combinando *Ant Colony Optimization* com *Local Search* é proposto para modelar este problema. Primeiro, um modelo foi desenvolvido para resolver um dado caso estático. Depois, é proposta uma estrutura para coordenar quaisquer mudanças dinâmicas nas entradas ao longo do tempo. A validação é feita usando referências de 100 clientes para o problema de direcionamento de veículos para recolhas e entregas. Visto que não existem referências para a abordagem dinâmica, estes foram gerados com base nas referências estáticas. O algoritmo de otimização proposto para resolver o *Capacitated Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment* é aplicado ao caso de estudo, que contém pedidos reais de clientes. Foi deduzido um modelo funcional para o problema abordado, que necessita apenas de considerar as distâncias reais entre nós, i. e., as distâncias para viajar pelas ruas entre duas moradas, e como direcionar uma frota heterogénea de veículos, visto que o modelo proposto só tem em conta uma frota homogénea. A proposta abordagem de *clustering* melhora os resultados. Além disso, o proposto algoritmo de otimização apresenta bons resultados para o ambiente dinâmico e é adequado a ser aplicado no contexto do mundo real.

Palavras-chave: *problema de recolhas e entregas, optimização de colónia de formigas, pesquisa local, janelas de tempo, pedidos dinâmicos*

Abstract

In this thesis, it is proposed a model for courier services capable of handling the routing of a fleet of vehicles. More specifically, a Capacitated Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment is formulated. This problem emerged from a real-world problem existent in the company *Urban Dynamic Delivery*, which performs scheduling and planning for courier services. This specific case study requires a limited short time for solving the optimization problem. Thus, a good solution must be found in a small time window, instead of finding the optimal solution independently of the required computational time. A new method pairing Nearest Neighbourhood Search with Subtractive Clustering is proposed to improve initial solutions and to accelerate the convergence of the optimization algorithm. Further, a hybrid method combining Ant Colony Optimization with Local Search is proposed to model this problem. First, a model was developed to solve a given static instance. Then, a framework to coordinate any dynamic changes on the inputs over time is proposed. Validation is made using static benchmarks of 100 customers for the pickup delivery vehicle routing problem. Since no benchmarks exist for the dynamic approach, these were generated based on the static benchmarks. The proposed optimization algorithm to solve Constrained Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment is applied to the case study, which contains real customer requests. A functional model for the tackled problem was derived, that only needs to consider the real distance costs between nodes, i.e. the costs for travelling through the roads between two locations, and how to direct a heterogeneous fleet of vehicles, since our model only accounts for a homogeneous fleet. The proposed clustering approach improves the results. Further, the proposed optimization algorithm presents good results for the dynamic environment and is suitable to be applied to the real-world scenario.

Keywords: pickup delivery problem, ant colony optimization, local search, time windows, dynamic requests

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

According to the United Nations [1], it is clear that big cities are bound to grow both in size and number. The population growth of each city follows the same trend, as can be seen on Table 1.1. It is estimated that in the last year more than half of the world's population lived in urban settlements, number which is expected to grow up to 60% by 2030. The share of population residing in cities is also projected to increase in all regions, with one in every three people predicted to live in cities with at least half a million inhabitants.

To accommodate the ever-growing number of people in cities, there must be a continuous effort to expand them in size, infrastructures and services. Unfortunately, this also means we will be increasing the number of vehicles travelling in cities, overburdening the already problematic vehicle saturation of urban settlements. Inner city logistics become increasingly worrisome up to a point where the sheer volume of traffic is so high any improvement in efficiency becomes of extreme value, no matter how small.

In recent years we have seen big advances on science and technology that granted access to an incredible amount of data about what surrounds us, in realtime. Such is the case for meter-accurate GPS tracking or on-demand traffic information. Combining the improving computational power available every day, new strategies unthinkable a few years back can be remastered and updated to solve the growing complexity of routing a fleet of vehicles thought a series of locations at the desired times.

In a realtime route planning, the environment is dynamic. This means both traffic status and target request list change over time, so a dynamic problem cannot simply be tackled as a static instance. A balance must be done between a good solution and a fast response in order to consistently achieve a reasonable solution facing constantly changing conditions, both for the customer and for the service provider.

Table 1.1: City population and average annual rate of change for years 2000, 2016 and projection for 2030

		City population (thousands)			Average annual rate of change(percentage)	
Country	City	2000	2016	2030	2000-2016	2016-2030
Brazil	São Paulo	17 014	21 297	23 444	1.4	0.7
China	Shanghai	13 959	24 484	30 751	3.5	1.6
France	Paris	9 737	10 925	11 803	0.7	0.6
Germany	Berlim	3 384	3 578	3 658	0.3	0.2
India	Delhi	15 732	26 454	36 060	3.2	2.2
Portugal	Lisbon	2 672	2 902	3 192	0.5	0.7
Portugal	Porto	1 254	1 304	1 443	0.2	0.7
United Kingdom	London	8 613	10 434	11 467	1.2	0.7

1.2 Objectives and Contributions

The objective of this thesis is to develop a model for courier services based on the general guidelines provided by the company *Urban Dynamic Delivery*. It deals with moving goods between two locations by efficiently handling the routing of a vehicle fleet. It is constructed to handle both changing traffic conditions and insertion of new customer requests in realtime.

A new model was created to model a routing problem for handling pickups and deliveries. It is formulated to respect most common constraints required for practical purposes, such as obeying target delivery times and having limited load capacity. Ultimately, our focus will be to consistently get a good solution under a tight time interval instead of achieving the optimal result, which could take a long period of time. For validation, the developed strategy is tested against publicly available benchmarks of 100 customers for static problems [2]. Besides this final value comparison, our generated routes are also validated as feasible according to a program directly granted from the entity behind the page providing the benchmarks. Concepts were implemented and tested in MATLAB.

To limit the complexity of the problem, which is already complicated with all the applied constraints, we will steer away from multi-objective optimization by hierarchically ordering the goals. Not doing so would also encumber the problem formulation, leading to slower computational times. The main focus will be on minimizing lateness to serve all current requests, then on reducing the number of vehicles and finally minimizing the total travelled distance. Waiting times will also have a negative impact on the objective function. The weights given to each objective are an input to the model for greater flexibility.

Besides defining a new mathematical formulation applicable to the modelled problem, it is showed that pairing the ant colony metaheuristic with local neighbourhood search is a valid approach. Also, a novelty concept for initial solution construction is proposed based on nearest neighbourhood search and subtractive clustering

1.3 Thesis Outline

A short presentation on this thesis' remaining contents is done in this section, as a way to summarize each chapter. Firstly, a brief introduction on the vehicle routing problem state of the art for all most relevant concepts used throughout this document is made on Chapter 2.

At Chapter 3 the problem at hand is described as concisely and clearly as possible. A mathematical formulation of the problem is presented, together with a listing of all variables and assumptions. In the rest of the section every relevant module or step of the solution employed is explained in detail, both as a whole and as a constraint-by-constraint analysis. To further explain the concepts and strategies implemented tables, figures and schematics from algorithms and computational simulations are presented.

Chapter 4 contains all the results generated both for model validation and demonstration proposes, together with relevant comments.

Finally, on Chapter 5 we summarize the results obtained on the last section and further elaborate on the results interpretation. As an ending note, we evaluate what objectives were fulfilled, how competitive we find the model and what would be the next steps to further improve this work.

Chapter 2

Vehicle Routing Problems

2.1 Most common types of Vehicle Routing Problems

2.1.1 Introduction

The Vehicle Routing Problem (VRP) category is a combinatorial optimization and integer programming set of problems. It deals on how to direct a fleet to serve interest points in the most profitable way. It has many uses in industry.

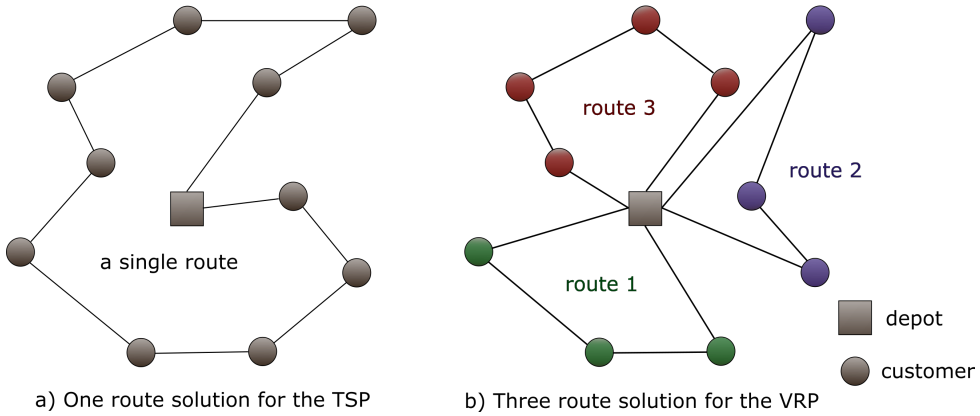
The main hurdle when solving a combinatorial problem is the sheer number of solutions that compose the feasible solution set, from where the ultimate goal is to extract the optimal solution. Many of these problems are in fact NP-Hard, i.e., there is no guarantee the optimal result can be reached in polynomial computation time. As formally defined in [3], an *optimization problem* is a set of instances, where an instance is a pair (S, f), where S is the domain of feasible solutions, f is the cost function and $f : S \rightarrow \mathbb{R}$ maps a positive cost value for each of the solutions $s \in S$. The goal is to find a feasible solution of minimal cost value $s \in S$ for which $f(s) \leq f(y)\ \forall\ y \in S$. The mentioned solution is known as the *global optimal solution*, s_{global}. Throughout this thesis the expression *fitness* will be used to denote the objective function value of the relevant solution.

This type of problems have their origin at the Travelling Salesman Problem (TSP), coined in 1948 by Merril M. Flood [4]. Here, the goal is to make a single salesman find the Hamiltonian cycle with the best cost, i.e., a round-trip through each and all the cities only once. Since then, the community has dedicated a large effort on creating, testing and comparing different approaches to the problem.

From this initial concept a broader generalization was made in 1959, the Vehicle Routing Problem (VRP), credited to Dantzig and Ramser [5] with their original ”Truck Dispatching Problem”. The focus of the VRP shifts to finding the optimal set of routes for a fleet of vehicles to serve a given set of customers. The new concept expands the TSP by introducing the notion of *delivery*. We are no longer interested only on the path between each city and now need to take into account what needs to be done at each location, namely that specified deliveries of certain goods must be made at every interest point. An example comparison between the two problems can be visualized in Figure 2.1 for the same set of points.

From this starting point, over the years extra constraints were added to represent and solve diverse

Figure 2.1: TSP and VRP example for the same set of points



a) One route solution for the TSP

b) Three route solution for the VRP

scenarios. As a consequent, many variants appeared which augmented the range of problems where the VRP can be applied. Many of the additional VRP constraints can be expanded into their own categories [6] [7], such as:

- **Capacitated VRP (CVRP)**: each customer has a specific demand for an amount of goods, which are characterized by a certain volume or weight. All goods originate from a defined location called the *depot*. Vehicles have finite capacity, meaning they can only take a certain volume and/or weight at a time.
- **Time Windows**: the VRP with Time Windows is used when each interest point needs to be visited between a specified time interval. A distinction can be made for *hard time windows*, where it is mandatory that each node is serviced within the defined time window limits or the solution is deemed infeasible, and *soft time windows*, where interest points can be visited outside the designated time window. For the later case, visiting any point outside the respective time window incurs a penalty that scales with distance to stated time objective. Any of the time window limits can be any of the two types.
- **VRP with Pickup and Delivery (PDVRP)**: instead of only delivering from depot to interest points, one might find itself to be in a scenario where there is a need to *pickup* an order from a specific location, besides the depot, and deliver it to another. A *request* is then characterized by the pair pickup-delivery. A precedence constraint must be added to the model assuring that for each pair, the pickup is visited before the delivery. Yet, multiple requests can be serviced at once, as a delivery does not have to immediately follow the matching request.
- **Heterogeneous fleet**: initial problem formulations account for an homogeneous fleet of vehicles with equal specifications, such as maximum capacity, travel speed, operation costs, etc. In the circumstance of needing to work with different vehicles the formulation has to change. Where before we only considered a certain fixed value for each variable, we now have to match every route to the specifications of the matching vehicle.

As we have seen lately [8], the research community's interest on more complex optimization problems

has been revamped. Making use of the expanding computational power available, VRPs have been further studied and the new devised strategies are able to handle convoluted and constrained problems faster, up to the point where we can solve some cases in real time. However, more complete algorithms alone might not be enough for the job. In many cases there is a need to combine developed strategies with extra decision support systems to solve the problems at hand. This is especially true when dealing with dynamic environments to make real time deployment viable. This has lead to new approaches targeted at handling dynamism and uncertainty to ultimately create effective tools for practical purposes.

All concepts briefly detailed above demand further explanation for better clarity, which will be presented over the next sections in detail as much as needed to prepare the reader to fully understand this thesis. Where they are meaningful, recent developments are mentioned and literature examples are commented. For an extensive and detailed list on vehicle routing problem variants and assumptions over the recent years the reader is redirected to [6].

2.1.2 Capacity constrained VRP

The capacity constrained vehicle routing problem (CVRP) is the closest formulation to the original "Truck Dispatching Problem" [5] and a common concept to add on all vehicle routing problem variations [6]. As mentioned above, this ties each interest point to a particular demand of goods to be delivered, originating from the depot.

Vehicles involved also have finite available space to transport goods. A problem can also be formulated where the vehicle capacity is unitary, and instead each vehicle can only take one request at a time, independently of what request it is. An example of such formulation is found in the "Swapping problem" [9].

Even though this is the oldest constraint of the VRP, new strategies are still being devised with the capacity constraint as main focus of the problem formulation. Examples can be seen using a Tabu Search algorithm as a complement to the routing problem [10] or a fuzzy logic approach [11].

2.1.3 Time Windows

A *time window* can be defined as an opening of limited duration during which something can be accomplished. It has a defined start and end time within the working time span[1] of the problem. A distinction can be made about the time window limits, namely if they are *hard* or *soft*.

When dealing with a problem where the time windows cannot be violated, this is, each node must obligatorily be visited within the specified time limits, we have what is known as a *hard time window*. A solution where at least one of the interest points is visited outside its time windows is not a feasible solution since the limits imposed by the hard time windows are absolute.

If, on the other hand, we are dealing with *soft time windows*, the time windows do not have the same mandatory character. Serving a vertex outside of its time window no longer automatically makes a solution infeasible. However, visiting the interest points far away from their respective time windows is

[1] The term *working time span* is used to denote the time interval during which requests can be serviced by the vehicles

not free of charge since it makes the solution unattractive. To translate this, a *penalty cost* is added to the fitness of a solution, deteriorating it. This penalty function usually scales with how far away from the time window limit the solution is, allowing solutions with vertices visited outside respective time windows feasible, but undesirable.

Depending on the tacked problem instance, time windows can also be a hybrid between soft and hard. This happens when both limits of a single time window need to be interpreted differently. As an example, imagine an interest point representing a package to be delivered by a courier service. It can only be gathered by the vehicle after it is available at location, denoting a hard initial time window limit. The client might have said it will be available for a certain number of minutes to give the package to the vehicle, defining the ending limit of the time window. If the client can no longer pass the parcel to the vehicle after the specified time, the ending limit is also considered a hard time window. However, if the client does wait indefinitely for the courier service to gather the package even after the designated time, we are before a soft ending time window. Obviously the client will be increasingly dissatisfied with the time it took for an interest point to be serviced after the stipulated limit, this is, with the *lateness* incurred when servicing the node.

The most common scaling of the penalty function is linear, i.e., it has direct proportionality to by how much it missed the relevant time window limit. Other variants also exist, as an example on [12], where the penalty growing over difference from time limit doesn't change linearly but quadratically. It is theorized by the author that this change will lead to a more uniform distribution of lateness compared to when applying a linear penalization.

2.1.4 The Pickup Delivery Vehicle Routing Problem

As mentioned earlier, the Pickup Delivery Vehicle Routing Problem (PDVRP) is a particular case of the VRP where people or goods have to be gathered at a certain location and then moved and dropped off at a different destination. Both pickup and delivery must be done by the same vehicle, including both locations on the same route, and pickup must obligatorily happen before delivery [6].

According to [9], we can classify them into three different groups regarding the ratio between number of pickups and deliveries:

- **Many-to-many**: this is the category if every vertex may initially contain an object of a specific type as well as a desired final type of object. An example of this is the aforementioned "Swapping Problem", where we have to construct a route to switch each locations' goods, using a vehicle of finite unitary capacity (only one object can be transported at a time). The goal is that vertex ends up with the desired object, independently of what it contained at the initial state.

- **One-to-many-to-one**: formulations for distribution of a specific type of supply will fall into this category, where we want to distribute a certain commodity from one depot (a warehouse, restaurant, etc.) to all customer vertices, at possibly varying quantities. The same logic can be applied the other way around, in the case customers vertices have commodities to transport to the depot, and

both cases can also be combined. As an example problem we can think of a restaurant distribution services, where one restaurant serves many customers.

- **One-to-one**: when all goods to transport have a given origin and a specified destination we are before a *one-to-one* problem. These problems will in most cases be capacity constrained, since goods have a certain volume and/or weight while vehicles have finite availability on those attributes. Typical examples include courier operations, door-to-door transportation services, etc.

We are especially interested in the *one-to-one* cases, since it is a more general formulation and the *one-to-many-to-one* can be solved also as a *one-to-one* problem, but with several simultaneous pickup points at the same location.

2.1.5 Heterogeneous Fleet

Dealing with a heterogeneous fleet adds extra effort to a problem formulation. When working with a homogeneous fleet, all specifications are equal between vehicles, which makes them interchangeable, and we only have to worry with the routing. With a heterogeneous vehicle formulation we also have to consider if each vehicle is being efficiently utilized for its designated route. Depending on the cases it might also mean adding extra strategies to test differences between vehicles.

Different vehicles may differ in speed, capacity, costs and available paths between two nodes. A car will travel faster than a bicycle, but it might not move as swiftly on congested traffic. Accounting for each individual specification for all vehicles, in cases when dealing with large instance sizes, steals computational power from finding the best solution. This is why in many formulations an homogeneous fleet is considered.

2.1.6 Dynamic problem formulations

When all input data is known beforehand, the routing problem is said to be *static* and it means no changes will happen in the inputs, namely requests, number of vehicles, travel costs, etc. This also means that any feasible solution obtained is always valid. When input data is revealed or modified during a routing problem's working span, we are before a *dynamic* problem and we can no longer approach it in the same way.

Usually, the input only fully revealed over time is the user requests. This means that on a dynamic instance the planning horizon may be unbounded. Therefore, strategies must be devised to compute updated directives to handle the changing inputs over time.

Inside the dynamic VRP category, most problems can fit in three different groups [9]. The natural extension of the PDVRP is the dynamic pickup delivery vehicle routing problem (PDVRP), when requests are for the transportation of goods and each vehicle can service multiple requests at once. If vehicles can only service one request at a time due to all requests having capacity equal to every request load, the problem is named dynamic stacker crane problem (SCP). Finally, if the transportation deals with moving passengers it is called a dial-a-ride problem (DARP). While this might be similar to the PDVRP,

there are extra constraints that must be taken into account when transporting people instead of goods associated with good quality of transport, such as maximum ride time.

To solve a dynamic problem we can adapt a method able to solve static instances into one that handles the input changes over time. The most basic way is to run the static solver every time the input conditions change (i.e., there is a new request or a cancellation). A troubling drawback from this path is that doing a complete reoptimization every time there is an update on the input conditions may not leave enough time for the static algorithm to arrive at a satisfactory solution between restarts.

The next relatively simple path to follow is to use the static algorithm from scratch only once, before any activity begins. Each time there is an update on the input conditions, the current solution is also modified to accommodate all input changes into a feasible solution, using heuristic methods, interchangeable moves or local searches fast enough to be used in realtime. This is the most commonly used option.

An interesting approach is presented by [12], where the working span is separated into successive time intervals of fixed length and buffering all new requests for later insertion. Starting from the solution of the static algorithm for all the pre-defined requests, each time interval will receive the selected best solutions from the instance before, insert any requests that were buffered and deploy the best solution. During the rest of the interval a simulation is made to predict the system state at the end of the interval, and that prediction is then optimized as if it were a static solution. When the end of the interval is reached, the deployed solution and the solution from the reoptimization are used as input solutions of the next interval. A more detailed explanation is present of Section 3.3.7.

Another example of a used strategy to model the dynamic behaviour of the system is using the notion of partitioning the working horizon, focusing on making good short term decisions without adverse long term effects. Detailed at [13], a so called double-horizon based heuristics for the dynamic pickup and delivery problem with time windows breaks the working span into two horizons, each with a different goal. Good results were achieved by the authors, but improvements over similar methods were only relevant on smaller instances.

Another extra complication when dealing with dynamic problems is how to compare the quality of solutions. While measuring the performance on a static counterpart is straightforward, comparing running times and solution quality of solutions to dynamic problems require different criteria to successfully measure performance. A useful comparison method is the *value of information* [13] and can be used when the optimal solution is known. Given an optimal solution $\dot{s}$ of the dynamic instance and an optimal solution s_{best} of the static instance, the authors adapt the relative error formula to define *value of information, V* as

$$V = \frac{f(\dot{s}) - f(s_{best})}{f(\dot{s})} \tag{2.1}$$

In the cases where we don't know either of the optimal solutions, an adaptation of the Equation 2.1 can be made into *value of information under heuristic* $\mathcal{H}$, making it a measure of effectiveness of used heuristic.

$$V(\mathcal{H}) = \frac{f(\dot{s}^{\mathcal{H}}) - f(s^{\mathcal{H}}_{best})}{f(\dot{s}^{\mathcal{H}})} \tag{2.2}$$

Most studies over the years have fallen into the static problem analysis, but in recent years a surge has appeared on cases dealing with dynamic problems. On the latter, instance solutions tend to ignore most real-world constraints, for example removing the heterogeneity of a fleet, dealing only with homogeneous formulations [6]. For a further review on dynamic VRP and PDVRP refer to [8] and [9], respectively.

2.1.7 Diversion

Another relevant topic to account for in dynamic problems is vehicle diversion. This is defined as redirecting a vehicle from its current immediate destination to another customer location while travelling [14]. In static problems there is no need to account for this, but on dynamic instances it might be advantageous to apply it, specially when new requests appear during vehicle travelling. This can only be applied if we assume there is communication between the dispatching central and the drivers.

2.1.8 Waiting Times

When dealing with dynamic problems, the way waiting times are handled has an increasing impact. Since the problem inputs arrive at different times, choices made at an early stage may prove to be detrimental to solutions later on. Most likely with any dynamic PDVRP, vehicles will have to wait at several locations along the working span, any time the vehicle arrives before the interest point (hard time window) is available. How this waiting time is handled can be a factor of the overall solution quality.

At [15] the authors present an empirical study of different waiting time strategies. The most basic one is the *Drive First (DF)* strategy. As the name implies vehicles drive to the next location as soon as it is possible, as soon as it has finished serving the current location. This might imply waiting at the target location if it arrives before being able to service the target location. This is the most common strategy.

Next, an example on the opposite side of the scale is used, the *Wait First (WF)* strategy. It makes the vehicles wait at the last serviced location, leaving only at the latest departure time that ensures it can still make the next delivery on time. This approach builds shorter routes than the previous strategy, but utilizes more vehicles to do so. More requests are known by the time vehicles leave their positions, leading to a greater potential to optimize the routes, but new requests appearing during the working span might not be possible to service due to all the idle time at the beginning.

The best strategy described by the authors is the *advanced dynamic partitioning* (ADP). It uses the notion of *dynamic partitioning of a route*. The concept is to group requests close in terms of location and time windows into successive *service zones*. Inside each service zone the vehicle uses the DF strategy. Between service zones it resembles a WF strategy, but this waiting time is only a portion of the longest feasible waiting time. This portion can be related to the total time span or to the sum of available slack times [2]. The advanced dynamic waiting strategy is considered to be the best option in terms of total

[2] "Slack time" is here used as the amount of time a certain task can be delayed without causing another task to be delayed.

route length.

2.1.9 Starting and ending route locations

Most formulations begin at a specific depot, ideally at a central location on the map, since this is the point vehicles have to start from and return. A more complex problem would consider the vehicles starting from any random position on the map, to further approximate a real-world case where workers might start from any current location.

The usual VRP formulation also finishes the route at the same vertex they started, completing a cycle. Even though commonly this point (usually the depot) is at a central location, it might be more efficient to distribute the unoccupied vehicles throughout the map.

In [8] *sampling* is introduced. The concept is that vehicles, after finishing their scheduled route, don't have to return to the depot and wait there for the next requests. Instead, as they move to a statistically more likely place where new requests might appear on the future, based on where customer nodes have been distributed in past instances. This in theory will lead to a shorter response time.

2.2 Methods to solve VRP

Computational methods used to tackle such optimization problems can usually fit in two categories, exact algorithms and approximation algorithms [16].

With an exact algorithm it is possible to guarantee that the optimal solution will be computed. The problem with these methods is that in the worst case scenario, we might have to list all possible solutions before reaching the optimal one. Due to this, problems with larger size become infeasible due to the exponential increase in the amount of options (example: Dijkstra's algorithm [17]). For problems with larger complexity, we need to use approximation methods.

Approximation methods work on a balance between quality of solution (closeness to optimality) and computational effort. These approximation methods can be grouped into three different definitions: construction techniques, local search techniques, and metaheuristics techniques [18].

- **Construction methods**: Being the most simple of the three types, these methods usually start from scratch, with the initial set either empty or starting at a random node. To compute a feasible solution, heuristic rules iteratively add new elements until all elements have been selected. Example: Nearest Neighbourhood Search, detailed at Section 2.2.1.

- **Local Search methods**: starting from a feasible solution, systematically explores the neighbourhood looking for improvements in the current solution. If any improvement is found, the search is continued from the new found best solution and repeats the problem until a full search of the neighbourhood finds no improvements. The inherent problem with these methods is that without implementing any extra strategies they are not able to escape a local minimum if one is encountered. Example: 3-opt exchange, detailed at Section 2.2.2.

- **Metaheuristics methods**: While also being driven by a search process around the neighbourhood, there are two differences from local search methods that allow us to escape local minimums. First, metaheuristics make use of stochastic components in their searching process. Second, the algorithm might choose solutions which do not improve current solution. When well designed, they are able to find solutions very close to the real optimal solution. Unfortunately, these methods are slow and with large instances the time required to achieve good performance might be prohibitive. Example: Ant Colony Optimization, detailed at Section 2.2.3.

One way to overcome the methods' limitations can be to combine them. Merging the exploration of a metaheuristic with the exploitation of a local search, to complement the shortcomings of each model. This approach is widely applied to different problems [19] [20] [16]. While the metaheuristic generates new solutions, the local search ensures there is no better solution on the neighbourhood. This approach allows to efficiently solve combinatorial problems in reasonable time.

2.2.1 Construction methods

A simple construction method, which our initial solution is based on, is the Nearest Neighbour Search (NNS). Even though it is a greedy algorithm and most of the time produces unsatisfactory results, it is widely used to construct an initial feasible solution from scratch.

The approach to NNS in a TSP is simple. From all the available nodes from the current node, it identifies the closest node and travels through the respective. It repeats this greedy selection iteratively until all nodes are visited once, at which point it returns to the starting point to complete the cycle. This process is also described on Algorithm 1 and can be extrapolated to other problem formulations.

Algorithm 1 General steps of a Nearest Neighbour Search for the Travelling Salesman Problem

```
1: start route on an arbitrary node
2: while there are unvisited nodes do
3:     find out shortest edge from current node to all other available nodes
4:     move to closest node
5:     mark current node as visited
6: end while
7: return to initial node
8: return route
```

2.2.2 Local search

Local search (LS) intends to exploit the current solution s in search of improvements. It consists on iteratively exploring current solutions neighbourhood, $N(s)$, looking for a better solution. It does this by locally changing the structure of a given solution, hence the notion of neighbourhood search.

When using a small $N(s)$ we might only get a few improving iterations before reaching the local optimum, but the running time is also small. With a large $N(s)$ exploration takes a long time and since computational time is a finite resource it might not be enough and also lead to poor quality solutions. Choosing the appropriate search neighbourhood requires specific insight on the problem being solved.

Local search procedures for VRP usually fall into the *edge-exchange* category. Starting from a feasible solution, they search for solutions on $N(s)$ with a better fitness by deleting k edges and replacing them with other edges. These connect the same nodes in a different way, completing the cycle, performing what is known as a *k-exchange*. This is iteratively executed until no additional improvement is found, arriving at a local optimum, which can also be the global optimum.

When such a local optimum is found we can call the achieved solution *k-optimal*, which required $O(n^k)$ time to achieve. It has been shown [16] that increasing k does return better quality solution, but the computational time quickly becomes prohibitive. Because of this, as a general rule $k \leq 3$.

Algorithm

Starting from a feasible solution s, iteratively local search works by exploring the solution's neighbourhood $N(s)$ for a more attractive solution. If the objective is the minimization of a fitness value, it searches for a minimum s' such that $s' \in N(s)$. Denoting $f(s)$ as the fitness function of a solution s, if $f(s') < f(s)$ the algorithm repeats the iteration after setting $s = s'$. If $f(s') \geq f(s)$, the algorithm stops and returns s as best found solution. It is important to make clear that the returned solution is a local optimum, but not necessarily the global optimum.

Algorithm 2 Pseudo code for general Local Search methodology

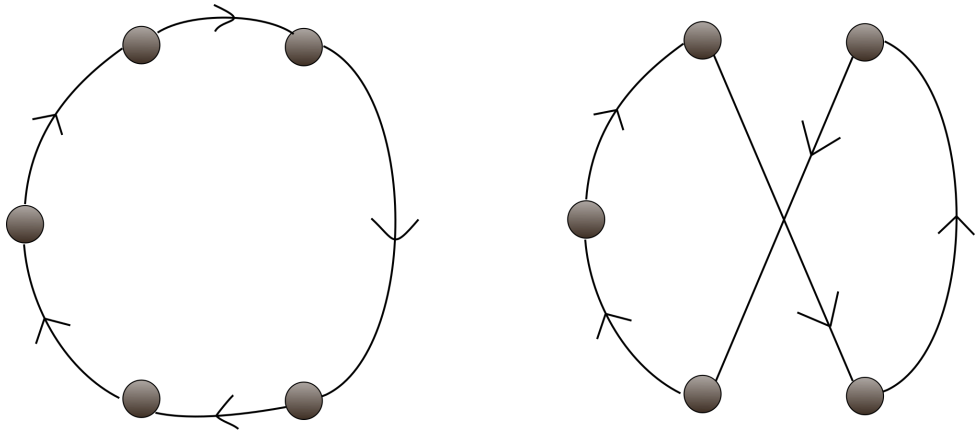
```
1: define initial solution s
2: while no termination condition do
3:     generate set of feasible neighbourhood solutions N(s)
4:     get s', minimum fitness solution from N(s)
5:     if f(s') < f(s) then
6:         s = s'
7:     else
8:         terminate algorithm
9:     end if
10: end while
```

A simple but common method is the *k-opt* algorithm, where k different arcs are deleted before another some rule reconnects them, forming a new path. After deleting, the reconnecting strategy varies according to the specific problem constraints. For precedence constrained routing problems, further considerations can be found on [21]. To illustrate the concept, Figure 2.2 exemplifies a step for *2-opt*, from where the 3-opt exchange can be extrapolated. Iteratively, two edges are deleted and replaced, creating a new solution.

Another noteworthy local search strategy is the *or-exchange* or *or-opt*. It is similar to the *k-opt* formulation, but intended to generate more feasible solutions for a precedence constrained problem. Since it will be used in the proposed model approach, it is here described in detail on a subsection of its own.

Instead of deleting arcs like the *k-opt*, a certain chunk of size *s* from the route, or *slice*, is separated from the rest. The *neighbourhood search space* is composed by all feasible solutions when moving the slice to each position around the original one, up to *L* steps in every direction. Each time all solutions from the neighbourhood search space have been tested and no better solution has been found, the

Figure 2.2: 2-opt local search strategy for TSP



whole process is repeated for a slice with one less element. The method can be visualized at 2.3 and a detailed algorithm can be consulted at Section 6. Note that when exchanging more than one node their original travel order is maintained.

Due to a possible prohibitively high computational effort, we might limit the range at which we try to insert the k-set to only include indexes on a viable neighbourhood. If we are dealing with time tied problem it might make sense to only search the neighbourhood solutions where the k-set is inserted on at similar times. If we are dealing with a multi-vehicle route encoding it would also make sense to test the k-set at other compatible times in all vehicles.

Advantages and disadvantages of Local Search

By using an appropriately dimensioned neighbourhood search we can thoroughly explore any promising solutions in terms of their local optimum regarding the defined neighbourhood search space. Yet the more thorough we want our search for the local optimum the higher computational effort scales.

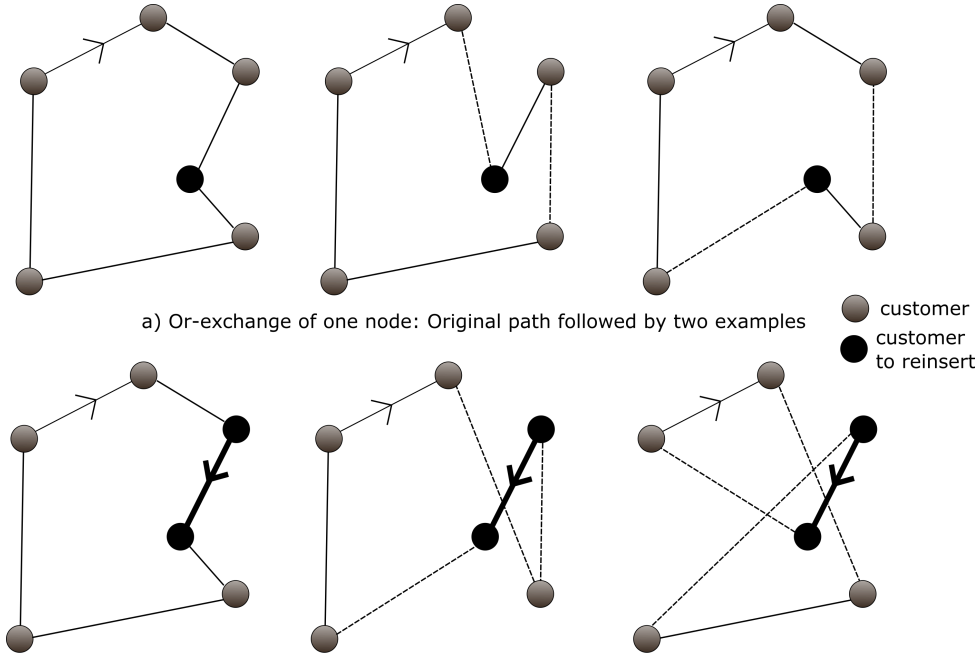
While extremely good on exploiting current solutions in search of improvements, a local search method is not able to escape a local minimum by itself. It must be aligned with another strategy, perhaps a metaheuristic, to sufficiently explore the solution set.

Applying a local search method when dealing with constrained problems, such as precedence (PDVRP) or capacity (CVRP) constrained problems, implies evaluating the feasibility of the solutions, which in turn can become cumbersome when having to be repeated for every new generated solution.

2.2.3 Metaheuristics

As mentioned before, metaheuristics have been replacing exact methods when these are deemed inefficient for large search space solutions. They are a set of general directives that can be adapted into a big variety of problems with little changes in the inner algorithm workings. Many of the metaheuristics are based on natural processes we try to mimic from nature. One example is the Ant Colony Optimization

Figure 2.3: Or-opt local search strategy for VRP, for k = 1 and k = 2



(ACO) [22], where the inspiration comes from the foraging ant process. A detailed analysis on this particular formulation can be found on Section 2.2.3

While heuristics are more problem dependent techniques, commonly tailored to fit the problem at hand, metaheuristics come as more general approach that can in theory fit most problems. Heuristics are usually too greedy, suffering from reaching and being trapped in any local minimum they might encounter. Metaheuristics overcome this by allowing a temporary deterioration of the solution, enabling them to explore further the search space.

Some notable examples of metaheuristic formulations, besides ACO include Genetic Algorithms, which draw inspiration on natural selection, and Tabu Search, utilizing memory structures to support and encourage a diverse search.

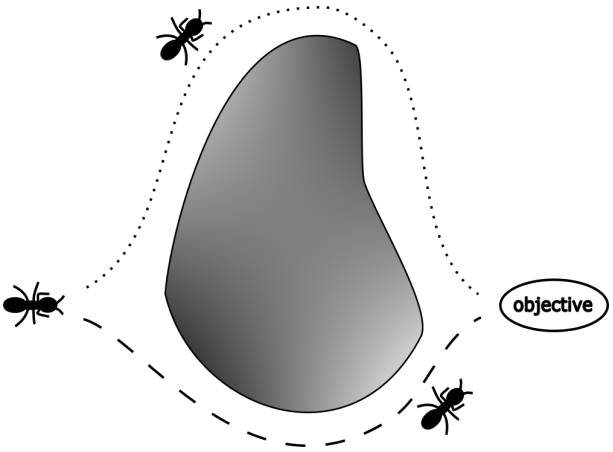
Tabu Search is a method that uses memory to guide the search process. The basic idea is to use a greedy local search algorithm complemented by a short term memory to further diversify the search space and escape local minimum.

Genetic algorithms draw inspiration from Darwin's theory of evolution, creating methods based on reproduction, generations, inheritance and natural selection. A mechanism that evaluates the fitness of a function chose which individuals will mate to create an offspring based on the combination of the parents. Only the best solutions will be used to generate the next offspring (survival of the fittest) resulting in iteratively generating better individuals by merging the bits of the best achieved parent solutions.

Ant Colony Optimization Metaheuristic

The ACO algorithm is a system based on the behaviour of real ant colonies, as it mimics the way ants search for food and find the way back to the nest. Real ants are capable of finding food as a colony

Figure 2.4: How real ants overcome decision points



without visual clues in a changing environment.

Ants start by exploring the neighbourhood area of their nest randomly. When one of them finds food it carries some back to the colony, but also leaves a chemical *pheromone trail* on the ground to guide other ants towards the source of food. The quantity of pheromones will depend on the perceived quality and quantity of food available at the source. After some time more ants have travelled in the same path and left more pheromones, making it even more appellative for other ants. When no more food is found, ants may still travel that path due to high pheromone trails, but wont leave any more pheromones. With time, the pheromones *evaporate* and the trail will no longer be so attractive for ants.

Figure 2.4 represents how real ants overcome a decision point as a colony. After randomly choosing a path they both arrive at the same objective and when returning both leave the same amount of pheromones. Yet, ants travelling through the shortest path take less time to get the food and return, allowing them to make more trips using the shortest path comparing with using the longest. Increasing the number of trips equals leaving pheromones more times, intensifying its pheromone trail against less efficient alternatives.

In ACO, each artificial ant builds a solution by making its way through a weighted node graph, visiting every node once and stopping at the starting point. Each vertex $i \in V$ in the graph represents an interest point (be it pickup, delivery or depot) and each edge $ij \in \mathcal{C}$ has associated a cost c_{ij} translating the effort to transverse that arc (usually distance between nodes). This is called the *heuristic information*.

Just as there is a cost c_{ij} associated with each arc, there is also a *pheromone trail value* τ_{ij} equivalent to each c_{ij} component. This τ_{ij} will be a model constructed value iteratively (within given limits) during the algorithms search, and it represents the memory of the colony about the current problem, similar to how pheromones work on real ants. When moving from node to node along the edges, ants use the pheromone information together with the heuristic information to pick the next point. Ants leave an amount of pheromone $\Delta\tau$ on the travelled solutions, and this amount depends on the quality of the solution.

Algorithm

ACO metaheuristics usually follow the same logic, where after proper **initializations**, such as algorithm parameters and pheromone trails, a step called **schedule activities** contains and manages the following operations: a) Construct Solutions; b) Daemon Actions; c) Update Pheromones. It repeats this cycle iteratively until a certain stopping criterion is achieved (be it number of iterations or, as is our case, CPU time).

Algorithm 3 Pseudo code of general ACO metaheuristic

```
initializations
while termination conditions are no met do
    schedule Activities:
        construct solutions
        daemon actions
        update pheromones
end while
```

Construct Solutions: A specified number of m artificial ants generate new solutions by choosing its next move probabilistically. Using Equation 2.3, we define the probability with which ant k in city i chooses to move to the city j, being j contained in a set of all feasible nodes from i. Each ant starts with an empty set and at a random point in the graph. At each construction step it will successively select one feasible node from the unvisited nodes list. It stops when all nodes are visited once and the ant returns to its starting position.

$$p_{ij}^{k} = \frac{\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}}{\sum_{j \in feasible\ set} \tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}} \tag{2.3}$$

with η_{ij} being the heuristic information associated with c_{ij} and both α and β being ACO model parameters intended to balance relative weight importance.

Daemon Actions: this includes any specific actions, for example problem specific operations, that cannot by implemented by single ants. This includes for example, applying a local search procedure to further explore any desired solution.

Update Pheromones: Ideally, the pheromone matrix will shape accordingly to the current problem and present high pheromone trail values on the edges belonging to the optimal solution, while presenting low values (and thus, diminishing the chances of probabilistic pick) on all the bad ones. It can be seen on Equation 2.4 that this is achieved by decreasing all pheromone values through evaporation and increasing the pheromone trail values corresponding to good solutions.

$$\eta_{ij} = (1 - \rho) \times \eta_{ij} + \rho \times f(s_{best}) \tag{2.4}$$

where ρ is the evaporation coefficient (rate at which pheromone trail values wear off) and $f(s_{best})$ is a value derived from the quality of the best solution s_{best}, commonly called the fitness value.

The amount of pheromones added by each ant is computed from the fitness of the current solution being used to update pheromone trails. Evaporation appears as a useful way to forget old solutions in favour of better/most recent ones.

ACO is then able to find good solutions on big search spaces without stopping at local minimums. This is attributed to the use of stochastic moves and memory while accepting solutions worse than the previous ones.

Historical context

The first description of an Ant Colony Optimization algorithm is credited to Marco Dorigo's PhD thesis in 1992 [23]. From these initial concepts many variations appeared, being three of the most important the Ant System (AS) [24] [25] [22], Ant Colony System (ACS) [26] and MAX-MIN Ant System (MMAS) [27].

- *Ant System (AS)*: First ACO algorithm defined in literature. With this method pheromones are updated by all the ants that have completed the tour and the fitness value if $F(s) = 1/L_k$, where L_k is the tour length of the k-th ant.
- *Ant Colony System (ACS)*: The major difference from AS is the decision rule used by the ants during the construction process. Using a parameter q_o, the ACS use a pseudorandom proportional rule , which translates into:
  - if $q > q_O$, equation 2.3 is used (biased exploration)
  - if $q \leq q_O$, next node is dictated by $arg\ max_{u \in J_r^k}\{[\tau_{ij}]^\alpha \times [\eta_{ij}]^\beta\}$ (exploitation)
- *MIN-MAX Ant System (MMAS)*: The novelty improvements offered by the MMAS are that only the best ant adds pheromone trails and the minimum and maximum values of the pheromones are limited by an explicit value set by the algorithm designer.

Advantages and disadvantages of Metaheuristics

The transversal advantage of metaheuristic strategies is their ability to explore large search spaces without getting trapped in local minimum. This is achieved using its ability to temporarily accept a worse solution than the current one in search of better alternatives not immediately on the feasible neighbourhood.

Unfortunately, as with most metaheuristic, this strategy needs to be accompanied by long computational times to ensure reachability of a good solution. These computational times scaling with the number of points to service, translating into a bigger demand in terms of computational efforts.

Another unfavourable aspect of metaheuristic strategies is how much solution quality depends on parameter tuning. No such thing as optimal parameters for all problem instances exist and so the output of the models will always depend greatly on the specified input parameters. This can only be solved by an experienced tuning targeted at each problem's specifications.

Points for and against ACO

A point in favour of using ACO on our problem instead of other metaheuristics is the way the solutions are encoded. Artificial ants make their way through what can be seen as weighted graphs, since each

arc between nodes has an associated heuristic value. This is similar to the distance (or time) it takes to move through several arcs from point A to B on a city node graph of many streets, avenues, etc. A logic way to represent a path can simply be the successive order of which interest nodes are visited.

Due to all constraints on the tackled problem, some unvisited nodes might not be available to the artificial ants. Pairs of requests, capacity check, heterogeneous fleet and the time windows can prohibit certain moves depending on the time, which we now have to account for. While overrides can be made to generate feasible solutions even under such hard constrains, this leads to extra mathematical complexity, deteriorating solution quality by requiring an increased computational time.

How to encode constraints on ACO

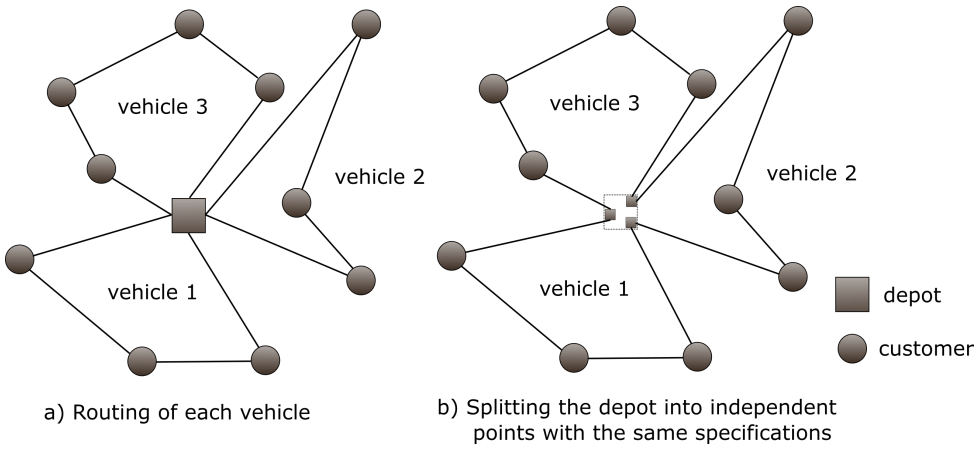
The usual way to encode constraints into a ACO formulation is to limit the available nodes each ant can travel to next, as it is important to not construct infeasible solutions. Theoretically, we start with an infinite number of vehicles to assure an initial solution is feasible and systematically iterate throughout the search space to find a better solution.

The standard way to model the capacity constraint is to only allow visiting vertexes whose pickup of goods added with the current vehicle load wouldn't exceed maximum vehicle capacity. Other interesting concepts include using fuzzy numbers to determine when nodes could be visited and then using a fuzzy logic system to determine the optimal vehicle for each vertex [11].

Dealing with time windows is similar to the capacity constraint. When dealing with hard time windows, a node is only available if the time when it would be visited is within the designated time window. For soft time windows this constraint is relaxed, and in terms of path construction there is no apparent downside to visit a node outside its time window. Only when computing the attractiveness to travel to a set of nodes can we see the influence of the penalty associated with visiting nodes outside their desired time window. This makes it more likely to visit nodes with no associated lateness.

Another tricky constraint to encode into the ACO formulation is dealing with a fleet of several vehicles, since one ant travels to all nodes once before returning to the starting location. An approach presented in [16] consists in splitting the depot into many nodes equal to the original one, in the same number as the number of deployed vehicles. Each of these depot points as the same characteristics as the original depot and can't be visited in a row. This way nodes visited between two depot nodes represent the route of a single vehicle. Not all depot nodes need to be visited for a solution to be considered feasible. A representation can be seen on image 2.5. This way all interest points can be serviced by the same ant using different vehicles.

Figure 2.5: How to represent several vehicles with only one ant



Chapter 3

VRP for courier delivery

3.1 Problem Formulation

An adaptation from the well-known mathematical formulation of the VRPTW [28] is presented below, where the goal is to service as efficiently as possible a set of customers requests $\mathcal{R} = \{1, ..., n\}$. Every request is defined by a pickup and a delivery location, each represented by a unique graph node out of a total $2n$ nodes. The full set of customers to service is given by $\mathcal{C} = \{p_1, d_1, p_2, d_2, ..., p_n, d_n\}$ where p_r is the pickup node of request r from the subset $\mathcal{P} = \{1, 3..., 2n-1\} \subset \mathcal{C}$ and d_r is the delivery node of request r from subset $\mathcal{D} = \{2, 4..., 2n\} \subset \mathcal{C}$.

In order to service each customer request we have available k vehicles. Since the strategy from 2.2.3 for handling multiple vehicles is being used, the depot location is split into k nodes forming the set of depot nodes $\mathcal{W} = \{1, ..., k\}$.

The mathematical formulation is dimensioned for a graph $\mathcal{G}(\mathcal{N}, \mathcal{A})$, where $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$ is the set of graph edges representing all different travel possibilities between nodes and $\mathcal{N} = \mathcal{W} \cup \mathcal{C}$ represents the graph nodes. $\mathcal{V} = \{1, ..., k\}$ is the set of homogeneous vehicles, each with e_k vehicle costs and capacity $q \geq l_i,\ i \in \{1, ..., n\}$ where l_i is the load capacity demand for customer i, i.e. how much of a vehicle's available capacity a request will occupy.

The variable x_{ij}^k is a binary parameter that expresses if a vehicle travels directly from node i to node j. For each arc ij, it takes the value 1 if vehicle k travels directly from i to j and 0 otherwise. x_{kj}^k represents an arc between a depot node and node j, serviced by vehicle k. Similarly, x_{ik}^k expresses an arc between a customer node i and the depot, also serviced by vehicle k Each arc is also defined in terms of travel time, t_{ij} specific positive travel cost, c_{ij}, for each arc in $\mathcal{A}$. Finally, s_i^k is the exact time of service at each point i by vehicle k and $[a_i, b_i]$ is the time window specified for node i.

$$\text{minimize} \quad \mathcal{M}_1 \times \sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}^k + \mathcal{M}_2 \times \sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{C}} e_k x_{kj}^k + \mathcal{M}_3 \times \sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} \max(s_j^k - b_j, 0)x_{ij}^k + \mathcal{M}_4 \times \sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} \max(a_j - s_j^k, 0)x_{ij}^k \tag{3.1}$$

$$\text{subject to} \quad \sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{C}} x_{ij}^k = 1, \ \forall\, i \in \mathcal{C} \tag{3.2}$$

$$\sum_{(i,j)\in\mathcal{A}} l_i x_{ij}^k \leq q, \ \forall\, k \in \mathcal{V} \tag{3.3}$$

$$\sum_{p\in\mathcal{P}} x_{hp_n}^k - \sum_{d\in\mathcal{D}} x_{gd_n}^k = 0, \ \forall\, h \in \mathcal{N}, \ \forall\, g \in \mathcal{N}, \ \forall\, k \in \mathcal{V}, \ \forall\, n \in \mathcal{R} \tag{3.4}$$

$$\sum_{j\in\mathcal{C}} x_{kj}^k = 1, \ \forall\, k \in \mathcal{V} \tag{3.5}$$

$$\sum_{i\in\mathcal{V}} x_{ih}^k - \sum_{j\in\mathcal{N}} x_{hj}^k = 0, \ \forall\, h \in \mathcal{C}, \ \forall\, k \in \mathcal{V} \tag{3.6}$$

$$\sum_{i\in\mathcal{V}} x_{ik}^k = 1, \ \forall\, k \in \mathcal{V} \tag{3.7}$$

$$x_{ij}^k(s_i^k + t_{i,j} - s_j^k) \leq 0, \ \forall\, (i,j) \in \mathcal{A}, \ \forall\, k \in \mathcal{V} \tag{3.8}$$

$$a_i \leq s_i^k, \ \forall\, i \in \mathcal{N}, \ \forall\, k \in \mathcal{V} \tag{3.9}$$

$$x_{ij}^k \in \{0,1\}, \ \forall\, (i,j) \in \mathcal{A}, \ \forall\, k \in \mathcal{V} \tag{3.10}$$

The objective function presented on Equation 3.1 is what differs most from conventional formulations. It is formulated with personalization in mind, giving the user control of the relative importance of the terms by adjusting the function *weights*, which is a vector composed by all scaling factors $weights = [\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4]$. Typically, VRP's focus is on minimizing the sum of travel costs over of the arcs used in the solution ($\sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}^k$). Since we are dealing with an heterogeneous fleet, we must also consider each vehicles' respective cost ($\sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{C}} e_k x_{kj}^k$). The first edge of a non-empty tour is used (x_{kj}^k) in order to only add vehicle costs on relevant tours. In this proposed approach we are dealing with soft ending time windows, which means servicing a node after its time window end (b_i) does not make a solution infeasible, only unappealing. When a node is visited after its time window ends, b_i, the extra time it took between the limit and the service time is called lateness. If it is allowed, its respective minimization must also a part of the objective function ($max(s_j^k - b_j, 0)x_{ij}^k$). The early limit of a time window, a_i, is still hard and inviolable. Finally, it proved beneficial to specify drivers' idle time as undesirable. With this in mind, the minimization of the total waiting times ($max(a_j - s_j^k, 0)x_{ij}^k$) appears as the last term and is especially relevant when dealing with heavily clustered data.

The constraint represented in Equation 3.2 assures that all customers are serviced only once. Vehicle capacity constraint is represented by Equation 3.3. Equation 3.4 assures the same vehicle services both pickup and delivery nodes of the respective request. The expression 3.5 defines that there is only one tour per vehicle, which can be empty. Equation at 3.6 ensures that if a vehicle arrives at a costumer location it also departs from the mentioned customer location. Equation 3.7 define all tours' ending

location as the depot. For a vehicle to travel directly from i to j Equation 3.8 states the arrival time at customer j is such that it allows travelling between i and j. Expression 3.9 specifies that the early limit of a time window a_i is hard and 3.10 denotes the x_{ij}^k variable as binary. Thus, the model is non-linear due to the non-linear *max* operations in 3.1, quadratic terms in 3.8 and integrality constraints at 3.10.

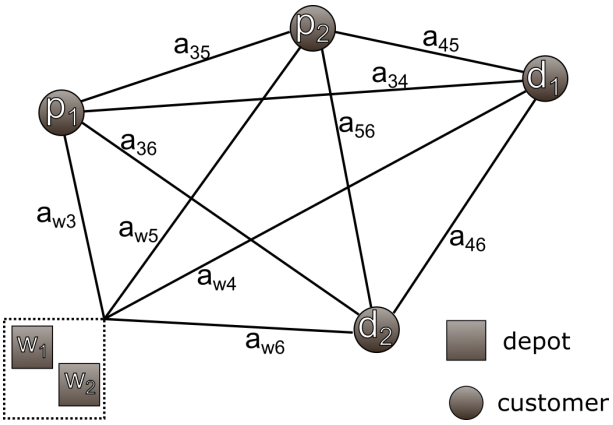
To illustrate these concepts, a simple problem instance is demonstrated below. Suppose, for example, we are before an instance where we have to service two requests, $\mathcal{R} = \{1, 2\}$, using a maximum of two vehicles, meaning we have available two depot nodes $\mathcal{W} = \{1, 2\}$. Our graph nodes are given by:

$$\mathcal{N} = \mathcal{W} \cup \mathcal{C} = \{w_1, w_2\} \cup \{c_1, c_2, c_3, c_4\} = \{w_1, w_2, p_1, d_1, p_2, d_2\} \tag{3.11}$$

This leads to a 6 by 6 graph arcs matrix $\mathcal{A}$ where each element a_{ij} represents the connection between i and j. If we want to reference the arc between pickup p_1 and delivery d_1 and we refer to a_{34}. Assuming the problem is symmetric $a_{ij} = a_{ji}$ and that $w = \{1, 2\}$, we can visualize the graph on Figure 3.1.

$$\mathcal{A} = \begin{array}{c} \begin{matrix} w_1 & w_2 & p_1 & d_1 & p_2 & d_2 \end{matrix} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix} \end{array} \begin{matrix} w_1 \\ w_2 \\ p_1 \\ d_1 \\ p_2 \\ d_2 \end{matrix}$$

Figure 3.1: Example of graph visualization, $w = \{1, 2\}$



3.2 Existing similar approaches

The used strategies for solving VRPs resembling our proposed approach are the following:

- **Tabu Search combined with Adaptive Large Neighbourhood Search**

On [29], the authors develop a strategy merging Tabu Search and another metaheuristic named Adaptive Large Neighbourhood Search (ALNS) to solve a capacitated vehicle routing problem with time windows. The ALNS behaves similarly to the Large Neighbourhood Search, where from a given route a certain number of requests are unconnected by a *destructive procedure*, to later all be reinserted by a *repairing procedure*. For more details refer to [29] and [30]. Although it is considered by the authors as a rather complex model, it is possible to achieve good results in reasonable time.

- **Genetic Algorithms**

  In another article a genetic algorithm is combined with a constructive heuristic to target a very specific problem, supply-chain scheduling for distribution of ready-mixed concrete [31]. It circumvents the slowness of the metaheuristic by giving it only part of the problem to solve and finishing with the construction procedure. The comparison between devised metaheuristic hybrid strategy and other commonly used construction heuristics resulted in interesting potential on the proposed solution.

- **Iterated local search**

  At [28], the author uses a metaheuristic that implements a local search method known as Iterated Local Search. The result has been named SPIDER and has since then been turned into a commercially available routing software. After exhaustive testing the model is considered robust and efficient on a large variety of VRPs, both from literature and industry test problems.

- **ACO paired with LS**

  A combination of the Ant Colony System and local search methods applied to the probabilistic travelling salesman problem is detailed in [20]. The focus of this thesis is on the stochastic component of the environment, and both a detailed literature review and development report of the developed strategy is present. Another example can be found on [32].

The problems tackled in these references are comparable to our formulation. Nevertheless, none is fully applicable to our defined model. The closest example resembling the previously defined formulation is found on the Iterated Search Example [28].

3.3 Proposed Approach

For the remainder of this Chapter the employed approach to solve the problem is broken down into modules and described in detail. The key of the approach is the *static solver*, a module that receives an existing solution and tries to optimize it further. It runs in loop until a stopping criteria is meet and uses a hybrid Ant Colony System paired with Local Search. The *static solver* only needs as input a valid solution, for example one from the *initial solution constructor*, to solve a static instance. The results of such strategies can be seen on Section 4.3.

Our approach to solve dynamic instances is based on the presented solution for static problems. After creating an initial feasible solution, the *static solver* is applied for a limited period of time. This intends to represent the pre-computation of requests already known beforehand. At this point we are at the beginning of the working span and will next repeat the same set of directives until all requests have been serviced:

- Insert any new requests from the previous interval into the best selected routes.
- Deploy the best obtained solution from the insertion to the physical vehicles.
- Predict current interval's end-state, namely vehicle positions and serviced requests.
- While current interval's end isn't reached, optimize the end-state prediction with the *static solver*.
- Output from the *static solver* the best found solution and the latest found solution.
- Group these two solutions with the state of the deployed route to form the best selected routes.
- Update vehicle positions, serviced requests and distance matrix.

Each of these modules is further described below.

3.3.1 Initial solution

The *initial solution constructor* is used to generate a feasible solution from scratch. This intends to be used as a starting point for the *static solver* module. The implemented strategy to generate an initial solution is based on the Nearest Neighbourhood Search (NNS), whose inner workings have been previously described in Section 2.2.1.

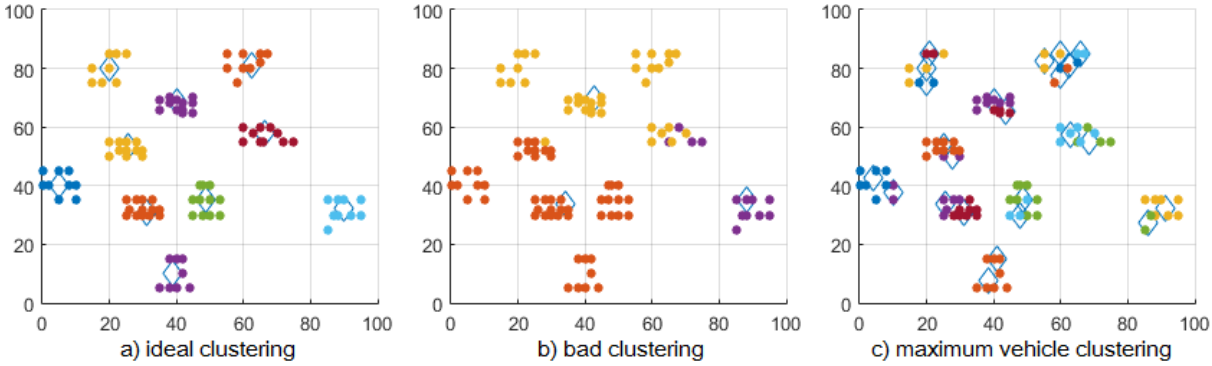
Only applying a NNS leads to unsatisfactory results. Even though it deserves merit for quickly obtaining a feasible solution to our model without any previous solution as guideline, this greedy solution provides increasingly bad routing decisions towards the end of the solution. The better the initial solutions, the faster the convergence of the model to a good solution. With this in mind, a *clustering* of each midpoint of all customers' requests is done. To each of the newly generated clusters, a single vehicle is assigned, which travels through its respective clustered nodes, following the NNS heuristic.

This request grouping is done using *subtractive clustering*, explained below, to generate an initial solution more promising than just using the NNS on all the requests. A comparison between using only NNS or clustered NNS is showed in Table 4.1, which supports the decision to use pre-clustering. More comments on these results are presented alongside the respective table at Section 4.1.

Subtractive clustering works by computing the likelihood of each node being a cluster centre, based on the density of surrounding data points. The highest potential data point is selected as the first cluster centre and every nodes that are close in distance are tied to it. The selection of which nodes are considered near each tested cluster centre depend on *cluster influence range* parameter. The previous three steps, likelihood computation, cluster centre selection and node allocation to new cluster are repeated until every node is paired with one cluster [33]. On Figure 3.2 we have three examples of clustering with different *cluster influence range* parameter values, respectively 0.2, 0.5 and 0.065.

Ideally we would reach a clustering as accurate as Figure 3.2 a), but without problem dependent parameter tuning we might end up with badly clustered dataset, as seen on Figure 3.2 b). On top of

Figure 3.2: Examples of pre-optimization clustering, for file *lc101*



that, reaching the clustering seen on Figure 3.2 a) is very uncommon since there is no guarantee every problem will be as easy to group as the presented instance.

Since we are not considering the time availability of the nodes, and are clustering instead based on geographical location, lateness might be present on the solution. To avoid assigning too many nodes into a single vehicle to service we make an effort to go in the opposite direction and assign as many vehicles as possible, as seen Figure 3.2 c). A small set of nodes per vehicle is easier to service, leading to smaller lateness. Since it is the main objective of our proposed approach it should be come before the number of vehicles or the total distance.

To start with as many vehicles as possible, and since this is an instance specific variable, a strategy is designed to dynamically choose an adequate *cluster influence range* such that the initial solution contains a number of vehicles close to the maximum available amount of vehicles for the specific problem instance. Starting with a very low *cluster influence range* parameter, subtractive clustering is applied on the midpoint of each pickup-delivery pair, taking into account only their location.

Having an extremely low *cluster influence range* results in each midpoint being a cluster centre, i.e, it asks for as many vehicles as there are pickup-delivery pairs. Iteratively, the *cluster influence range* is increased and the clustering process repeated until the amount of clusters is within the maximum vehicle number for each problem. A vehicle is then assigned to each cluster of points. After splitting each midpoint node into the original pickup and delivery nodes, the nearest neighbourhood search is used to construct a starting route.

3.3.2 Cost function

The *cost function* module analyses a given route and outputs the requested information. For a given input route, the cost function is based on the number of vehicles, total distance, waiting times and lateness of a route, usually for objective function computation. The module works by running through the input route step by step while simulating travel times, waiting times and service times of a route for the given inputs, such as distance matrix. Its schematics are represented on Algorithm 4.

To steer away from multi-objective minimization, the fitness of a function will instead be a sum of each term multiplied by its scaling factors, *weights*. The comparison between two solutions is made applying the respective scaling factor to the relative difference between solutions.

Algorithm 4 General steps for cost function computation

```
Inputs: route, distance matrix, vehicles' information, output option
Outputs: number of vehicles, distance travelled, waiting times, lateness
for each node on route do
    compute travel time between last and current node
    update travel time, travelled distance, waiting time and lateness counters
    if node is a vehicle then
        save values and tie them to previous vehicle
        reset counters and return to initial time
    end if
end for
```

3.3.3 New ant computation

Traditionally, the starting node of an artificial ant is picked randomly. Since we are encoding a multi-vehicle route into each ant, some changes from the classical formulation are noteworthy. First, the depot location is split into independent nodes paired with each allowed vehicle, creating the *depot node* set, as described in Section 2.2.3. This means that after adding the multi-vehicle constraint we no longer start randomly on any graph node. Instead, each artificial ant can only start randomly on a depot node.

An ant still has to return to the initial depot node at the end of a route to complete the cycle, but not all nodes of the graph $\mathcal{N}$ need to be visited. Only all of the customer nodes $\mathcal{C} \subset \mathcal{N}$ have to obligatorily be serviced, in accordance to Equation 3.2. An ant can only return to a depot node if there are no started pickups still requiring delivery, in conformity to Equation 3.4, the pairing constraint. Moving between two depot nodes in a row is forbidden. The amount of depot nodes visited during the ants path is solution specific.

Starting from a random vehicle, artificial ants select the next node to visit from the available nodes based on heuristic information and the pheromone trail matrix, using the aforementioned pseudorandom proportional rule. The set of available destination nodes depends on the currently occupied node, corresponding vehicle and previously visited nodes. Algorithm 5 further details the used strategy.

When starting and an ant is on a depot node, only pickup nodes unvisited by other vehicles are in the feasible node list. When the ant moves onto one of these pickups, the corresponding delivery is added to the feasible node list for this vehicle. From anywhere else but the depot nodes, all unvisited depot nodes are also feasible.

When a depot node is selected as an ant's next destination, this move is frozen. All nodes that must obligatorily be in the route to comply with the mathematical formulation are added, following the same rules as before. These nodes that must be added are the deliveries of already visited pickups and during this time no new requests are deemed feasible besides the necessary to complete all half serviced pairs of requests.

3.3.4 Local Search method

The local search implemented is based on the *or-exchange* method explained at 2.2.2. Careful parameters tuning must be made to achieve a good balance between an exhaustive neighbourhood

Algorithm 5 Pseudo code of New Ant computation

Inputs: route, distance matrix, pheromone matrix, vehicles' information, ACO parameters
Outputs: new ant route and corresponding fitness

```
get initial random vehicle
while pickup-delivery list isn't empty do
    list feasible next stops for current node and vehicle
    select next node according to pseudorandom proportional rule
    if selected node is a pickup then
        make corresponding delivery feasible and add node to must_add list
    else if selected node is a depot then
        while must_add list isn't empty do
            select node from the must_add list according to pseudorandom proportional rule
            add node to route, remove from must_add list
        end while
    else
        remove delivery node from must_add list
    end if
    add node to route
    update pickup-delivery list
end while
```

search space and computational speed. By trial and error, the values of $s = 3$ and $L = 3$ were found to provide good results consistently. The detailed steps of the computational implementation is described in Algorithm 6.

At the start, a part of the route (from now on called *slice*) of size s is removed from the original route. This slice will tentatively be inserted between every route node up to L times around the slice's original position, without leaving the current vehicle. If this slice is independent of the other customers in the same vehicle, this is, if only a pickup and the corresponding delivery are selected, the slice is also inserted on equivalent indexes at all other vehicles. The new insertion indexes are generated by a simple linear interpolation between available insertion indexes at each vehicle. When a feasible solution is generated, the corresponding fitness value is saved for later comparison.

This process is repeated for each possible slice with length s. Only after all options have been computed do we compare the best achieved result with the disturbed solution. If at this point any new feasible solution is an improvement comparing with the current disturbed one, it replaces the disturbed route and repeats the previous steps all over again. When no solution is found, slice size is decreased and the cycle is repeated. All this is stopped only when the slice's length is decreased to 0.

Due to heavy computational times, this local search cannot be executed on all computed ants. Instead, it only runs on the top promising ants from each iteration. If none of the top ants give a solution better than s_{best}, local search is applied to a solution derived from inducing a disturbance on s_{best}, as further explained on the next section.

3.3.5 Static Solver

The *static solver* is the key piece to the employed solution. Besides the pseudo code on Algorithm 7 detailing the logic behind this module, the flowchart A.1 on the appendices presents the schematics for further comprehension. It pairs the previously explained *local search* module with an ACO metaheuristic formulation.

Algorithm 6 Implementation of *or-exchange* method

Inputs: route, distance matrix, vehicles' informations
$route$ - solution where to apply Local Search
s - length of slice to exchange
L - size of search neighbourhood
Outputs: new ant route and corresponding fitness
Initializations:
n - length of original route
t - slice position index, starting at 1

```
1: while s > 0 do
2:     while (s + t) < n do
3:         separate route into slice = route(t : t + s), and rest, the remaining route
4:         compute search neighbourhood indexes on slice's original vehicle
5:         if slice is only a full request OR is a full request and contains a depot node then
6:             map search neighbourhood into other vehicles
7:         end if
8:         for each new solution of the search neighbourhood solution do
9:             if two vehicles in a row then
10:                test solution with both vehicles and keep most attractive
11:            end if
12:            if new solution is feasible then
13:                save and compute solution's fitness value
14:            end if
15:        end for
16:        if any feasible solution better than best then
17:            update best
18:        else
19:            increment t
20:        end if
21:    end while
22:    reinitialize t
23:    decrement s
24: end while
25: return best
```

Starting from a feasible solution, module specific variables are initialized. The *pheromone trail matrix* is also here generated. It is a $Q \times Q$ matrix, where Q is the total number of graph nodes, and it is initialized uniformly at the value $\tau_0 = 0.5$. Pheromone limits are dynamic and depend on the quality of the current best solution, meaning they will be updated every time a new global best solution, s_{best}, is found. Pheromone trail matrix limits are computed using Equations 3.12 and 3.13 [16], where $f(s)$ represents the *cost function* of a solution s. After referred initializations the next steps run in loop until a certain time interval passes

$$\tau_{max} = \frac{1}{\rho \times f(s_{best})} \tag{3.12}$$

$$\tau_{min} = \frac{\tau_{max}}{2Q} \tag{3.13}$$

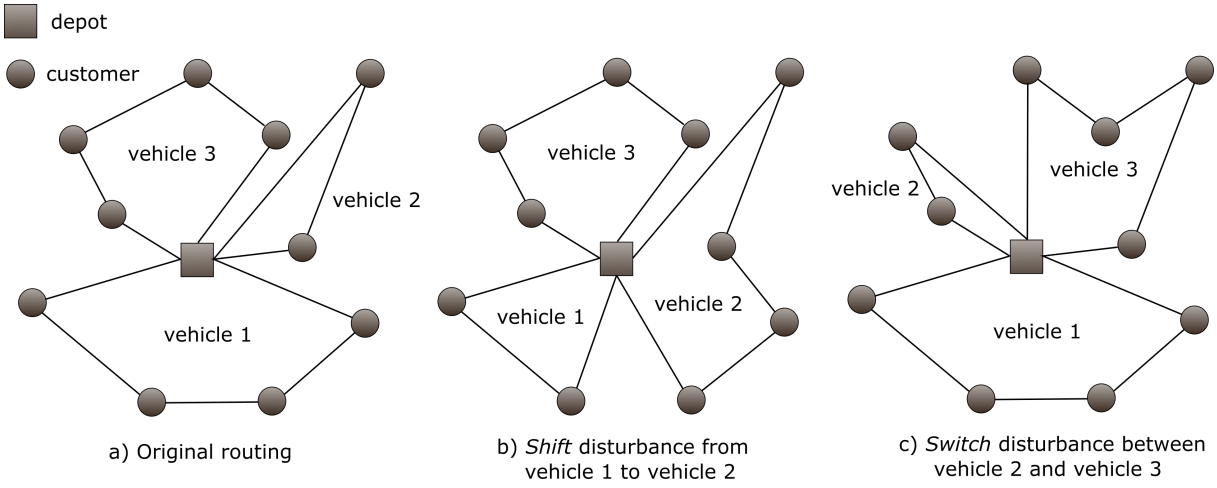
First, m new ants are generated using the *new ant* generator module. For a specified limit, *LS_limit*, the module runs purely as ACO algorithm since *local search* method is not yet used. The best ant from each iteration is compared with the best overall solution. If an improving solution is found, both s_{best} and pheromone matrix limits are updated.

Even though Local Search improves greatly initial solutions, at a starting stage the improvements given by it don't make up for the increased computational time. In other words, time spent doing local search on very bad quality solutions is more useful if instead redirected to letting the ant colony system work by itself for a while. Using local search will assuredly lead to the neighbourhood search space's most attractive solution, but at an initial phase exploration of the solution set is more important than exploitation of the current solution.

As explained at Section 2.2.3, a pseudorandom proportional rule is used. Applying it on our implementation gave better solutions early on, but took a toll on the method's achievable exploration. With this in mind, the parameter q_0 is initialized to 0.9, and then successively decreased to 80% of the previous value with each iteration where no new solution is found. When eventually one is found again, q_0 is reinitialized to its original value. This provided an improvement solutions when applied to our method. Besides these variations of the q_0 parameter, at each iteration there is one biased artificial ant. For this ant only, the value of q_0 is set to 1 to make this ant always select the most attractive move. For all other ants on the same iteration q_0 is changed back to the previous value. Both points were empirically tested and are further illustrated on Section 4.2

If in any iteration no new solution is generated, the *LS_counter* is incremented. Whenever a new better solution is found, this counter is reinitialized. When it eventually surpasses a certain threshold, *LS_limit*, local search procedures begin. The *LS_limit* variable is computed for each problem instance and represents how many times a value equal to the maximum pheromone limit, τ_{max}, could suffer *pheromone evaporation* before being lower than the minimum pheromone limit, τ_{min}. This leads to Equation 3.14. For the rest of the explanation it will be assumed we are already before a situation where the local search method has started to be used.

Figure 3.3: Representation of disturbance method *shift* and *switch*



$$LS\_limit = \frac{\log(\frac{1}{2Q})}{\log(1-\rho)} \tag{3.14}$$

After all ants are computed, the *local search* method is applied to the most promising ants on the top ant fitness list from current iteration (size specified by *top_ants_number*) until no improvement is found, and are compared with the global best ant, s_{best}. From now on, any time a comparison is made between any solution and the global best, s_{best}, it is implied that if the new solutions is better it will replace s_{best} and update the pheromone limits accordingly.

If no best solution was found so far, a disturbance is induced on s_{best} ant to the output s_{new} the local search method is applied. The number of new solutions generated by the disturbance method is given by *perturbed_ants_number* times. Every s_{new} is saved on *exchange_memory* to avoid applying a local search on equal solutions and save valuable computational time. The disturbance introduced can either be a *shift* or a *switch*, both represented at Figure 3.3. On the first one, a randomly selected pickup-delivery pair is removed from a vehicle and attributed to another at a random location. On the second disturbance method a shift is applied two times on the same pair of vehicles, removing and inserting a new pickup-delivery pair in each. A disturbance can happen to a route multiple times to further increase the search space, as is the case of the proposed model where it can happen from one to three times.

If s_{best} is updated on current iteration, the *LS_counter* is reinitialized, *exchange_memory* cleared and q_0 equalled to its original value. If on the contrary s_{best} remains the same as in the previous operation *LS_counter* is incremented. After local search is turned on, *LS_counter* starts to dictate when *pheromone reinitialization* will be applied. When this is not the case, the pheromone trail matrix is updated according to rule 2.4. When it is and we are before a pheromone matrix reinitialization, the pheromone matrix is equalled to the midpoint between τ_{max} and τ_{min}, the *LS_limit* is doubled and L in incremented by one and q_0 is again reinitialized. All these changes aim to increase the neighbourhood search area and the time spent searching on it before reinitialization.

When eventually the time limit is surpassed for the *static solver* module, it stops running iteratively and outputs the best found solution as result.

Algorithm 7 Pseudo code of Static Solver method

```
    Inputs: requests, vehicles, distances, run time
    Outputs: best found route and corresponding fitness
 1: while static solver is within run time do
 2:     iteration specific initializations
 3:     for desired m number of ants do
 4:         compute new-ant
 5:         update top-ants
 6:     end for
 7:     for each ant on top-ants do
 8:         apply local search
 9:     end for
10:     if no new solution was found this iteration by new ants then
11:         for desired number of times do
12:             induce a perturbation on best-ant to create new-best
13:             while an improvement is found on new-best do
14:                 apply local search to new-best
15:             end while
16:             update best-ant if new-best is better
17:         end for
18:     end if
19:     if best-ant was updated this iteration then
20:         update pheromone limits
21:         reinitialize stagnation counter
22:     else
23:         increment stagnation counter
24:     end if
25:     if stagnation counter over the limit then
26:         reinitialize pheromone trails
27:     else
28:         global pheromone trail update
29:     end if
30: end while
```

3.3.6 Least-cost Insertion of new requests

The LCI module is used to insert new requests into already existing solutions. It is costly timewise to generate a new solution from scratch for every new iteration of the static solver, especially since it will be used on a relative short time span (15 minutes) each time. Since in our proposed approach the output solution of the LCI is immediately deployed, it must generate a satisfactory solution.

The proposed LCI strategy is exhaustive in the sense that it will insert the most urgent pickup-delivery pair between all existing nodes and select the best. This exhaustive approach is employed due to the high number of constrains to guarantee a good deployed solution even though this is costly in computational time. If there are unused vehicles when doing a least-cost insertion it is also possible to insert a node into a previously empty vehicle, as long as the fitness value of the resulting solution is more attractive. The order of node insertion is the ascending latest service time at which a node can be serviced within specified time windows.

This was implemented by tentatively inserting the highest priority unserviced costumer in the existing routes. If this customer node is a delivery, the corresponding pickup is inserted first instead. This way the method can be implemented assuming the insertion order of each pickup-delivery pair is always pickup first. Until no more requests need insertion, a pickup is inserted between the first two nodes of a route. Between this insertion index and the route's end, the corresponding delivery is tentatively inserted at each possible index. For each feasible generated solution the fitness function is saved. After doing all delivery combinations, the best generated route is saved.

Next, the process is repeated, but at this time with the pickup between second and third node. This is done until the pickup has been in all possible indexes. When all combinations have been evaluated, the best values from each pickup insertion index are compared and the best route from all evaluated is deemed as the new route. This cycle is repeated until all pickup-delivery pairs are inserted. To better illustrate the described concepts above, the Algorithm 8 describes the process.

Algorithm 8 Steps for an exhaustive least-cost insertion

```
    Inputs: current route, requests to insert, distance matrix, vehicle's info
    Outputs: new route
 1: define pickup-delivery pair insertion order
 2: while there are customers to insert do
 3:     select pair to insert
 4:     for each possible insertion index of the pickup node do
 5:         for each possible insertion index of the delivery node do
 6:             if both insertions feasible then
 7:                 compute fitness value and save temporary route
 8:             end if
 9:         end for
10:         save new route with minimum fitness from temporary list
11:     end for
12:     update route with minimum fitness from new route list
13: end while
```

Obviously, this approach is costly in terms if computational time. When adding many nodes or the insertion route is too large, this process becomes prohibitive. Yet, since we aim to implement a dynamic

system where requests become available a fixed time interval before they need to be serviced, the proportion of requests in relation to the total route nodes is low and this strategy can be used.

3.3.7 Dynamic Solver

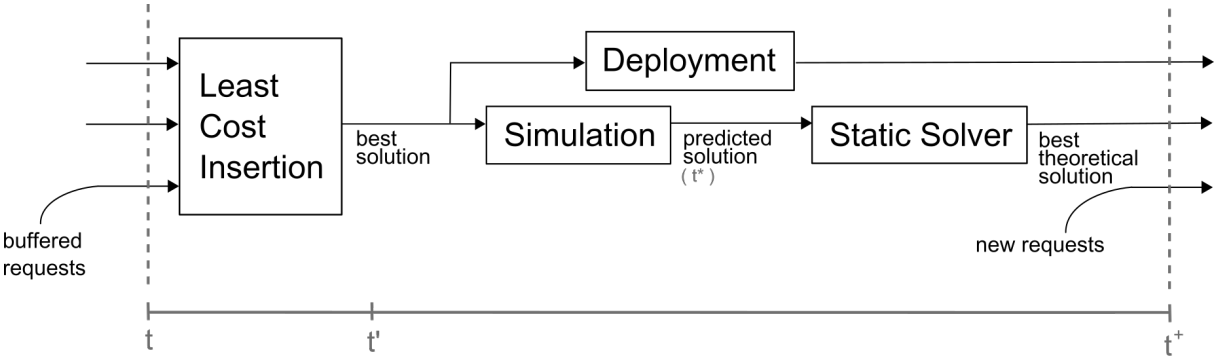
To handle the dynamic changes over time, the working horizon is divided into successive intervals of length $time_{ss}$. During each interval, the working conditions (time and distances matrix) will remain unchanged and any new requests appearing during this interval will be buffered, i.e., saved for later insertion. When the end of the interval is reached, time and distance matrix are updated and new requests inserted in the already existing routes. This way, during the length of the interval, the problem instance does not change and the *static solver* method can be applied.

The only place where a solution is generated from scratch is at the start of the dynamic solver. Before beginning this cyclic approach described above, depending on the instance to solve, it can be relevant to run the *static solver*, depending on the number of requests or other instance related properties. This step is optional and belongs to the initializations.

After this pre-optimization time interval is over, if any occurred, we enter the iterative cycle represented on Figure 3.4. Denoting the start of the interval as T, the best solutions obtained in the previous optimization interval serve as input to the *least cost insertion* module, which adds any new requests buffered during the previous cycle and outputs the best found solution. This is the solution to be deployed to the physical vehicles, $\dot{s}_{deployed}$, at time T', which would start to travel immediately accordingly to this route, ignoring any previous orders. Simultaneously to deployment, and while vehicles service their stipulated pickups and deliveries, another module called *end-state simulator* predicts where the vehicles will be at the end of the current time interval, based on how long the optimization interval is and how long each path takes to travel. This predicted route, $\dot{s}^*$, is used as input to the *static solver*, which will try to find improvements to this solution for $t_{ss} - (T' - T)$ minutes.

When the end of the time interval T^+ is reached, the *static solver* module is stopped. When dealing with static instances, the best solution is outputted from the module and it is terminated. In dynamic instances however, the best solution from the *static solver*, now $\dot{s}_{best}$, is grouped with the previously deployed solution, $\dot{s}_{best}$, and together they serve as input to next cycle's *least cost insertion model*, at T^+. All requests made available between T and T^+ are now introduced into the routes. The cycle is

Figure 3.4: Schematics for Dynamic Solver cycle



repeated until no more requests need to be serviced.

Algorithm 9 Pseudo code of Dynamic Solver

```
    Inputs: requests to service, fleet information
 1: pre-process requests known beforehand to create an initial solution (optional)
 2: while there are still requests to service do
 3:     LCI bufered requests on last solutions
 4:     select and deploy best solution to real vehicles
 5:     predict state of vehicles at the end of current interval
 6:     while current interval's end not reached do
 7:         execute static solver on predicted system end state
 8:     end while
 9:     update static solver solutions according to current time
10:     pass updated solutions to next iteration
11: end while
```

When solving dynamic instances, some changes are needed in the *static solver* module to account for requests that have already been fully serviced or whose pickup has already been serviced in the real world and thus is irreversibly tied to a vehicle. This means that the *new ant* module must also account for vehicle history when constructing the feasible node list. When generating new solutions with the local search strategy, any time a feasibility check is done it needs to also take into account nodes outside the current planned route but which are on the vehicle history.

The module *system end state* predicts where vehicles will be and what nodes have serviced during t_{ss}, time according to current distance matrix costs. To account for changes between the predicted end-state environment and the real environment at T', another module is needed to check if at the interval's end the predicted state matches the real state and fix anything needed accordingly. This step is present in the Algorithm 9 on line 9. For the benchmark problems, the distance between nodes is computed using the Euclidean distance but for the case study it is calculated based on the Haversine formula[1].

3.4 Validation

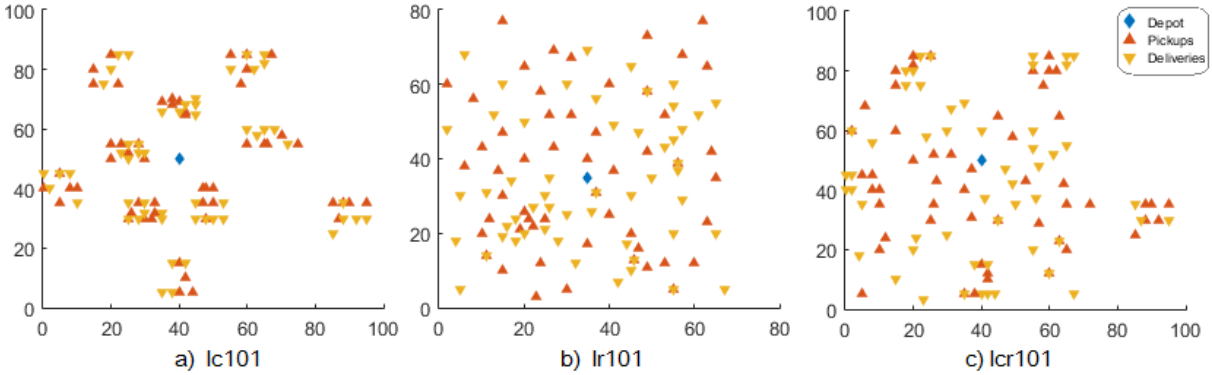
To validate obtained results, the benchmarks for a pickup delivery problem with time windows instances generated by Li & Lim [34] will be used. They can be found on [2], under the PDPTW section, as well as the best results for each file found, which shall be referred to as the optimal solutions for each respective instance. The datasets for the dynamic problem will be generated from these files.

Since the presented results do not allow lateness and have as objectives: 1) minimization of vehicle number; 2) minimization of travelled distance and hard time windows, weights must be adjusted accordingly. This will be implemented by having the scaling factors such that $\mathcal{M}_4 \gg \mathcal{M}_2 > \mathcal{M}_1 \gg \mathcal{M}_3$. The used relations between scaling factors for this thesis are $\mathcal{M}_4 = \mathcal{M}_1 \times 10^4 = \mathcal{M}_2 \times 10^5 = \mathcal{M}_3 \times 10^7$

Using a much greater factor for M_4 proved to be enough to steer away from any lateness in the solutions. The scaling factor for the waiting times, M_3, is given a very low influence so that our solutions incur in extra vehicles or travelled distance to decrease waiting times.

[1] useful to compute the shortest distance between two points on the surface of a sphere given their longitudes and latitudes

Figure 3.5: Comparison of plane distribution between *lc*, *lr* and *lcr* data types



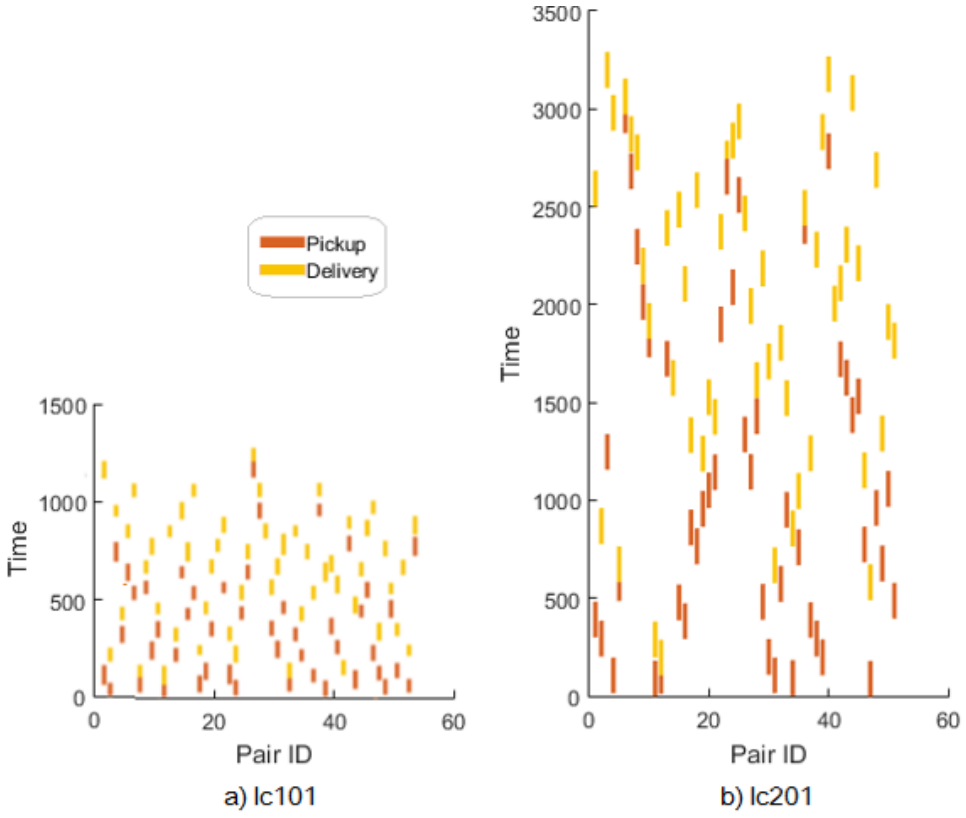
Benchmark file names contain encoded information on the type of problem. Using as example the 100 customer category (file name of type ”lc1xx”), each file is labelled in accordance to the following logic:

- **lc**: generated requests are clustered geographically
- **lr**: generated requests are randomly distributed geographically
- **lcr**: generated requests are partially clustered and partially randomly distributed
- **1**: time windows have short and conflicting scheduling horizon (many vehicles)
- **2**: time windows have long scheduling horizon (few vehicles)
- **xx**: file identifier

To better visualize the data, on Figure 3.5 we can see plotted the customer locations of files *lc101*, *lr101* and *lcr101* for coordinate distributed comparison.

Similarly, on Figure 3.6 we can see plotted the time window distribution of files *lc101* and *lc201* for working horizon comparability, where on the second plot the working span is over the triple from the first one for the same amount on customer nodes.

Figure 3.6: Scheduling horizon comparison between *101* and *201* data types



Chapter 4

Results

4.1 Initial solution construction comparison

To compare implementing the construction method with and without pre-clustering the customer nodes, 5 runs of 20 minutes for 12 different files were executed using both strategies. The selected files intend to represent as best as possible all type of benchmark distributions. A shorter time is used for this run when comparing with the other tests since we are interested on the initial impact using either strategy brings. Everything besides the initial construction remains unaltered for both strategies. Results can be seen at Table 4.1.

For each file and for each strategy, the average between all runs is present, listing: initial fitness value, the direct output of the constructor; run time at which the first solution without lateness appears; final fitness after 20 minutes. The best values in each case are highlighted in bold.

For both cases, the initial fitness shows no special trend. The first time a solution with no lateness is generated, the *transition time* is either almost identical for both strategies or the pre-clustering ones are at least three times lower than only using NNS, trend which happens for half of the cases. So far, only a slight advantage is shown for pre-clustering instead of only doing the NNS.

However, in the majority of cases using NNS on pre-clustered points provides a better final fitness. Pre-clustering the requests and then applying a NNS to each cluster will be the strategy used in all other sections.

4.2 Parameter Tuning

In this section two concepts are explored, changing the value of the *pseudorandom proportional rule* parameter for next node selection described in Section 3.3.3, and the usage of one fully biased ant. For the first case, 4 different files were tested with slight variations of the studied parameters. On the second case it is showed several runs for 5 files, with and without the biased ant.

The adopted *pseudorandom proportional rule* from ACS [16] introduces a new parameter to our model, q_0, giving the ratio between biased and probabilistic nodes selected when constructing a route.

Table 4.1: Comparison of initial and final solution values using 1-NN with and without pre-clustering. For each instance, the best solution out of five 20 minute runs is displayed in bold

Instance	Strategy	Initial Fitness	Transition Time (min)	Final Fitness
lc101	1-NN	**188.58**	**0**	**182.89**
	Clustering	190.49	**0**	**182.89**
lc105	1-NN	3.31E+11	0.44	89.16
	Clustering	**8.04E+10**	**0.07**	**88.89**
lc201	1-NN	9.97E+09	8.04	437.11
	Clustering	**5.53E+09**	**2.41**	**310.02**
lc205	1-NN	**347.24**	**0**	212.63
	Clustering	3.18E+10	1.73	**178.65**
lr101	1-NN	**7.08E+09**	**2.54**	**348.73**
	Clustering	9.67E+09	2.82	355.91
lr105	1-NN	**1.13E+11**	1.99	264.47
	Clustering	1.46E+11	**1.58**	**252.04**
lr202	1-NN	**182.89**	**0**	**182.90**
	Clustering	190.49	**0**	**182.90**
lr205	1-NN	3.18E+11	0.59	89.16
	Clustering	**1.89E+11**	**0.09**	**88.89**
lcr101	1-NN	8.66E+09	6.42	421.25
	Clustering	**3.98E+09**	**2.19**	**309.89**
lcr102	1-NN	2.44E+11	**1.74**	225.76
	Clustering	**6.02E+10**	2.02	**173.84**
lcr201	1-NN	**7.33E+09**	3.57	367.03
	Clustering	1.06E+10	**2.60**	**359.92**
lc205	1-NN	**9.6E+10**	2.06	275.44
	Clustering	1.48E+11	**1.42**	**247.83**

Table 4.2: Different tests done on the pseudorandom proportional rule parameter, q_0

File ID	q_0	Multiplier	Vehicles	Total Distance
	0	-	24	1960.55
	0.9	-	24	1849.61
18	0.9	0.9	**23**	1995.27
	0.9	0.8	**23**	**1878.41**
	0.8	0.5	24	1895.69
	0	-	15	1440.08
	0.9	-	15	1450.43
23	0.9	0.9	16	1555.86
	0.9	0.8	**14**	**1338.17**
	0.8	0.5	**14**	1417.95
	0	-	12	1362.36
	0.9	-	12	1267.83
28	0.9	0.9	13	1322.94
	0.9	0.8	**11**	**1177.16**
	0.85	0.5	12	1244.31
	0	-	4	1235.79
	0.9	-	**3**	1269.61
33	0.9	0.9	**3**	1618.42
	0.9	0.8	**3**	**1131.67**
	0.8	0.5	**3**	1195.65

For each of the selected files, five different variations were tested for the q_0. It was observed that the suggested value of 0.9 from [16] proved to be useful at the early solutions, but inhibited exploration in the later stages. Using $q_0 = 0$ also achieved good results but required longer computational times.

Since this thesis aims at achieving a good solution in a short time, the concept proposed is to start with a value of 0.9 and with each successive iteration decrease it. When a new improving solution is found, the parameter is set at its original value. A comparison can be seen on Table 4.2 , which lead to using the proposed approach on the implemented methods with an initial value of $q_0 = 0.9$ and a decrease of 20% per unsuccessful iteration, this is, multiplying the value by *0.8*.

Once per iteration, the current value of q_0 is temporarily replaced with 1. This *biased ant* will select the most attractive option as the next node instead of following the specified probabilistic rule. This is done to further exploit the current solution without compromising exploration. Since the test with the biased ant did not show a meaningful difference, the files for clustered data are not displayed on Table 4.3

The results confirm the usage of a single biased ant is beneficial to the solution quality. All strategies described above are present on the remainder results.

Table 4.3: Test runs with and without biased ant

File	Strategy	Vehicles	Distance
lr101	without ant	23.33	1875.33
	with ant	**20.70**	**1775.71**
lr109	without ant	13.33	1463.15
	with ant	**13.20**	**1376.57**
lr205	without ant	4.00	1224.59
	with ant	**3.60**	**1190.37**
lrc102	without ant	14.33	1728.43
	with ant	**13.20**	**1679.72**
lrc102	without ant	6.00	**1756.58**
	with ant	**5.20**	1763.81

4.3 Static Problem Solution

The static benchmarks tested are part of the 100 customer group PDPTW instances available at [2]. Modelling for the dimension of these files is enough to guarantee applicability into the case study presented by the company within reasonable computation times, since the example made available travels between a similar number of customer orders and scheduling horizon.

On the static analysis we present number of vehicles and total distance as basis of comparison, similarly to what is available at the source of the datasets. Tables with the absolute and relative error, e_{abs} and e_{rel} respectively, are annexed and display the comparison with the optimal solution values. Relative error is displayed in percentage.

For the clustered problem instance files it is also relevant to display the computation time of the last obtained solution, since in these runs optimal results were achieved. For the other files, the time of last obtained solution is close to the static solver time limit, and since for most files the optimal solution is never reached it adds no meaningful contribution.

Since the objective is applying the *static solver* in the dynamic module, short run times of 30 minutes will be considered for the benchmark test runs. This is done to see model's applicability for a problem of similar dimensions under what is a short time interval for most metaheuristic approaches.

Negative values in the error tables represent an improvement in relation to the existing solution. This happens on achieved solutions whose final number of vehicles is bigger than the optimal's and by chance the distance is more efficiently distributed, as can be seen on file *lc109q*. It still deems the solution as sub-optimal since the priority is minimizing vehicle number.

Files with short scheduling horizon and files with long scheduling horizons will be denoted as type 1 and type 2, respectively.

4.3.1 Clustered datasets

Results for the clustered data are very promising, as seen on Tables 4.4 and B.1. The proposed model is able to solve 11 out of 17 clustered data benchmarks to optimality at every attempted run, in a

Table 4.4: Clustered data files for 100 customers

File	Optimal values		Mean values		Best values		Time (min)
	Number of vehicles	Distance	Number of vehicles	Distance	Number of vehicles	Distance	
lc101s	10	828.94	**10**	**828.94**	**10**	**828.94**	0.05
lc102s	10	828.94	**10**	**828.94**	**10**	**828.94**	0.55
lc103s	9	1035.35	10	828.06	10	828.06	4.04
lc104	9	860.01	9.8	864.97	**9**	1025.95	8
lc105s	10	828.94	**10**	**828.94**	**10**	**828.94**	0.15
lc106s	10	828.94	**10**	**828.94**	**10**	**828.94**	0.3
lc107s	10	828.94	**10**	**828.94**	**10**	**828.94**	3.28
lc108s	10	826.44	**10**	**826.44**	**10**	**826.44**	1.05
lc109q	9	1000.59	10	829.48	10	827.82	4.3
lc201s	3	591.56	**3**	**591.56**	**3**	**591.56**	4.2
lc202s	3	591.56	**3**	**591.56**	**3**	**591.56**	6.99
lc203*	3	591.17	**3**	**591.17**	**3**	**591.17**	5.62
lc204	3	590.60	3	625.94	**3**	**590.60**	7.57
lc205s	3	588.88	**3**	**588.88**	**3**	**588.88**	1.95
lc206s	3	588.49	**3**	**588.49**	**3**	**588.49**	1.66
lc207s	3	588.29	3	588.32	3	588.32	1.68
lc208s	3	588.32	3	595.99	3	595.99	2.33

short time span. For the other files, once the achieved solutions also matched the optimal solution.

Even though these results are good, the average run time column already shows some trouble in the near future. For files where the optimal solution could not be reached, the best solution is found under 5 minutes on a total of 30. If a new solution isn't being generated in the remainder 25 minutes it might indicate the current exploration is not enough to escape local minima close to the optimal solution.

Using a bigger L to increase the local search is not enough. Even though it explores the neighbourhood solutions better, it takes so much extra computational time that the slight better solutions its finds do not compensate the lesser number of total iterations, leading to overall worse results. Similarly, repeating the implemented disturbance methods to generate solutions further away from the local minimum might lead to an increase in solution quality but also increment the time spent per iteration.

Table B.1 highlights the number of reached optimal solutions. Absolute error on vehicle number is always 1 or lower, if any. Distance errors are also below the 10 % mark, which is considered acceptable.

4.3.2 Randomly distributed datasets

As it was expected, considerably worse results are found when solving random distributed data files, most likely due to the absence of a clear best path as in the clustered files. Still, optimal solutions could be found for 2 files and the optimal number of vehicles is reached for half of the tested files. Even thought it shows the model can still reach the optimal solution for clustered data, it fails to do as consistently as before.

Table 4.5: Random data files for 100 customers

File	Optimal values		Mean values		Best values	
	Number of vehicles	Distance	Number of vehicles	Distance	Number of vehicles	Distance
lr101	19	1650.80	20.7	1775.71	20	1739.51
lr102	17	1487.57	17.3	1605.41	**17**	1569.66
lr103	13	1292.68	13.7	1431.88	**13**	1338.01
lr104	9	1013.39	11	1169.27	11	1106.19
lr105	14	1377.11	14.7	1432.09	**14**	**1377.11**
lr106	12	1252.62	13.2	1367.17	**12**	1284.99
lr107	10	1111.31	11.3	1256.38	**10**	1135.45
lr108	9	968.97	10.3	1133.49	10	1047.14
lr109	11	1208.96	13.2	1376.57	12	1265.68
lr110	10	1159.35	12.3	1288.17	12	1241.69
lr111	10	1108.90	11.1	1170.41	**10**	1126.01
lr112	9	1003.77	10.9	1211.35	10	1105.98
lr201	4	1253.23	5.6	1479.69	5	1354.42
lr202	3	1197.67	4.6	1539.21	4	1350.44
lr203	3	949.40	3.6	1255.53	**3**	1085.14
lr204	2	849.05	3.2	1251.12	3	1086.37
lr205	3	1054.02	3.6	1190.37	**3**	**1054.15**
lr206	3	931.63	3.7	1448.35	**3**	1291.96
lr207	2	903.06	3.4	1461.95	**3**	1212.83
lr208	2	734.85	2.6	1134.33	**2**	895.88
lr209	3	930.59	3.5	1135.86	**3**	1061.15
lr210	3	964.22	4.1	1357.38	4	1229.53
lr211	2	911.52	3.2	1076.67	**3**	940.5

Table 4.6: Partially clustered and partially random data files for 100 customers

File	Optimal values		Mean values		Best values	
	Number of vehicles	Distance	Number of vehicles	Distance	Number of vehicles	Distance
lrc101	14	1708.80	15	1788.36	**14**	1753.40
lrc102	12	1558.07	13.2	1679.72	13	1635.32
lrc103	11	1258.74	11	1283.13	**11**	**1258.74**
lrc104	10	1128.40	10.8	1261.37	**10**	1149.12
lrc105	13	1637.62	16	1802.10	15	1718.42
lrc106	11	1424.73	12.6	1541.35	12	1479.69
lrc107	11	1230.14	11.4	1260.70	**11**	**1230.14**
lrc108	10	1147.43	11	1283.53	11	1199.25
lrc201	4	1406.94	5.6	1759.37	5	1586.48
lrc202	3	1374.27	5.2	1763.81	5	1614.26
lrc203	3	1089.07	4	1520.36	4	1326.89
lrc204	3	818.66	3	988.83	**3**	869.21
lrc205	4	1302.20	5.6	1712.07	5	1576.05
lrc206	3	1159.03	4.6	1469.76	4	1346.24
lrc207	3	1062.05	4	1389.99	4	1244.70
lrc208	3	852.76	3.67	1058.44	**3**	936.17

As seen in Table B.2, errors are more accentuated in the type 2 files. Difference from optimal number of vehicles is below 2, but for the type 2 files this translates into a big percentage. Distances also have bigger percentual error on type 2 files.

4.3.3 Mixture of clustered and randomly distributed datasets

Results for these files are similar to the randomly distributed ones. Once again, the type 2 files present a larger relative error. The model found the optimal solution for two files, but unlike the previous section where half the times we reach the optimal vehicle number, in this case it only happened 4 out of 23 times.

4.3.4 Comparison

To sum up, Table 4.7 presents the average values for the error tables per type of file. Clustered data behaves differently from the others files as it manages to always reach the optimal number of vehicles for type 2 files. While for type 1 it does not reach the optimal value for all of them, it reaches a lower distance than the given by the optimal. For the other two the conclusions are similar, with average vehicle number error of 2 or less. Distance does not fall below the optimal value as with the clustered files, but instead has an average error around 20%, which is alarmingly high.

Table 4.7: Average results of model versus optimal values for each file type

Instance	Vehicle Number				Distance Travelled (km)			
	Mean	%	Best	%	Mean	%	Best	%
lc1	0.31	3.46	0.22	1.39	-41.49	-4.06	-23.79	-2.00
lc2	**0**	**0**	**0**	**0**	5.38	0.91	0.96	0.16
lr	0.18	1.83	0.12	1.31	-19.44	-1.72	-12.14	-0.98
lr1	1.39	12.99	0.67	6.57	131.87	11.33	58.50	4.99
lr2	1.05	40.54	0.55	21.96	329.41	35.31	168.63	18.15
lr	1.23	26.17	0.61	13.93	226.34	22.80	111.17	11.29
lcr1	2	16.62	1.5	12.42	161.70	7.34	98.89	2.84
lcr2	1.35	38.51	1	29.16	371.76	28.86	226.77	15.66
lc	1.68	28.93	1.25	21.83	266.73	17.86	162.83	9.07

4.4 Dynamic Problem Solution

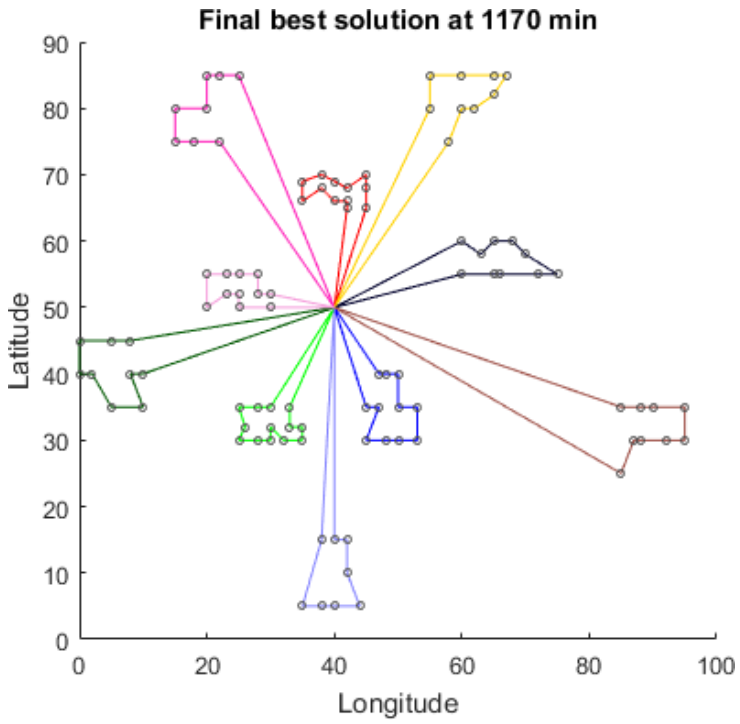
For dynamic testing, a problem instance will be generated based on *lc101*, chosen since the *static solver* is able to reach its optimal solution under 5 seconds. This way we are focusing only on testing the dynamic changes and the least insertion procedure by removing all difficulty for the *static solver* to find a solution. By looking at Figure 4.1 it is clear that using a clustered set of nodes makes it more intuitive to visualize the routes and the current solution quality.

Since no quantitative comparison can be made between dynamic and static solution we use this next section as a demonstration exercise, showing that the model works for the purpose it was created.

For simulation purposes it is necessary to distinguish between the actual run time of the static solver and the perceived time advanced by the algorithm. The first one is the already defined t_{ss} and the perceived time by the algorithm will be deemed t_{sim}. For data files whose requests' working horizon are over 20 hours it is not feasible to run the simulation in realtime, this is, when $t_{ss} = t_{sim}$. So, for each time the *static solver* runs for $t_s s$ time, the algorithm will see as if the time passed was actually t_{sim}. For example, as used on the next figure, $t_{ss} = 1$ minute is more than sufficient for optimization, while in the meantime the simulation advances $t_{sim} = 15$ minutes with each cycle.

As mentioned before, since when using the dynamic model the requests will be inserted over time, there will be less nodes to service at once with the *static solver*, leading to faster computational times. Besides the nodes that have yet to be inserted, some requests already had their pickup serviced, tying the delivery to the vehicle and reducing the search space greatly, since we only have to place each delivery on one specific vehicle instead of being able to move both the pickup and the delivery between every available vehicle. Eventually, some requests will be fully serviced and do not need to be accounted for anymore. This reduces each cycle's loop time immensely and makes it feasible to apply a smaller t_{ss} and still achieve good solutions. This is especially true for our selected case, since average run times are below 5 seconds we can be comfortable that 60 seconds is more than enough to solve any current dynamic instance. No elaborate mechanism is used to handle waiting times, so the basic *drive first* strategy is used.

Figure 4.1: Optimal route plotting for the *lc101* file



For the dynamic test an important variable needs to be defined, the *lookahead* parameter, t_{look}. The generation of dynamic files consists in making nodes available to be serviced only after a specified time. Up to that point they are not accounted for in the route planning. This particular time is defined for each request as the earliest service time (so the a_i of the respective time window limit) of both the pickup and the delivery. In other words, a pickup-delivery pair is inserted into the set of customers to serve as soon as any of the two have their time window starting before the current simulation time, $t_{current}$, plus the lookahead parameter, which can be represented as when the condition $a_i < t_{current} + t_{look}$ is true.

On the visualizations of the simulation, as seen on Figure 4.2, for each vehicle their current location is represented by a diamond shape, each have a unique color and when outside of the depot are associated to their unique ID number. A dashed line represents the planned route for a given vehicle while a solid line represents the path previously travelled. Grey lines represent already serviced routes whose respective vehicles are back at the depot and circles represent customer nodes.

4.4.1 Dynamic Requests

Here is presented a solution for the *lc101* file with $t_{look} = 45$ min, a $t_{static} = 1$ min and $t_{sim} = 15$ min. For this specific file, service time at the nodes is 90 minutes for almost all cases, being 0 on the other few. On Figure 4.2 we have plotted successive time instances, representative of the simulation's temporal evolution.

Starting at zero time, Figure 4.2 a) shows the planned route for the few initial nodes to service from the beginning, already distributed by the optimal number of vehicles. Next on Figure 4.2 b) is depicted the system state after 240 minutes, where vehicles have already moved from the depot. Some customer

Figure 4.2: Visualization of dynamic routing for requests, geographical distribution

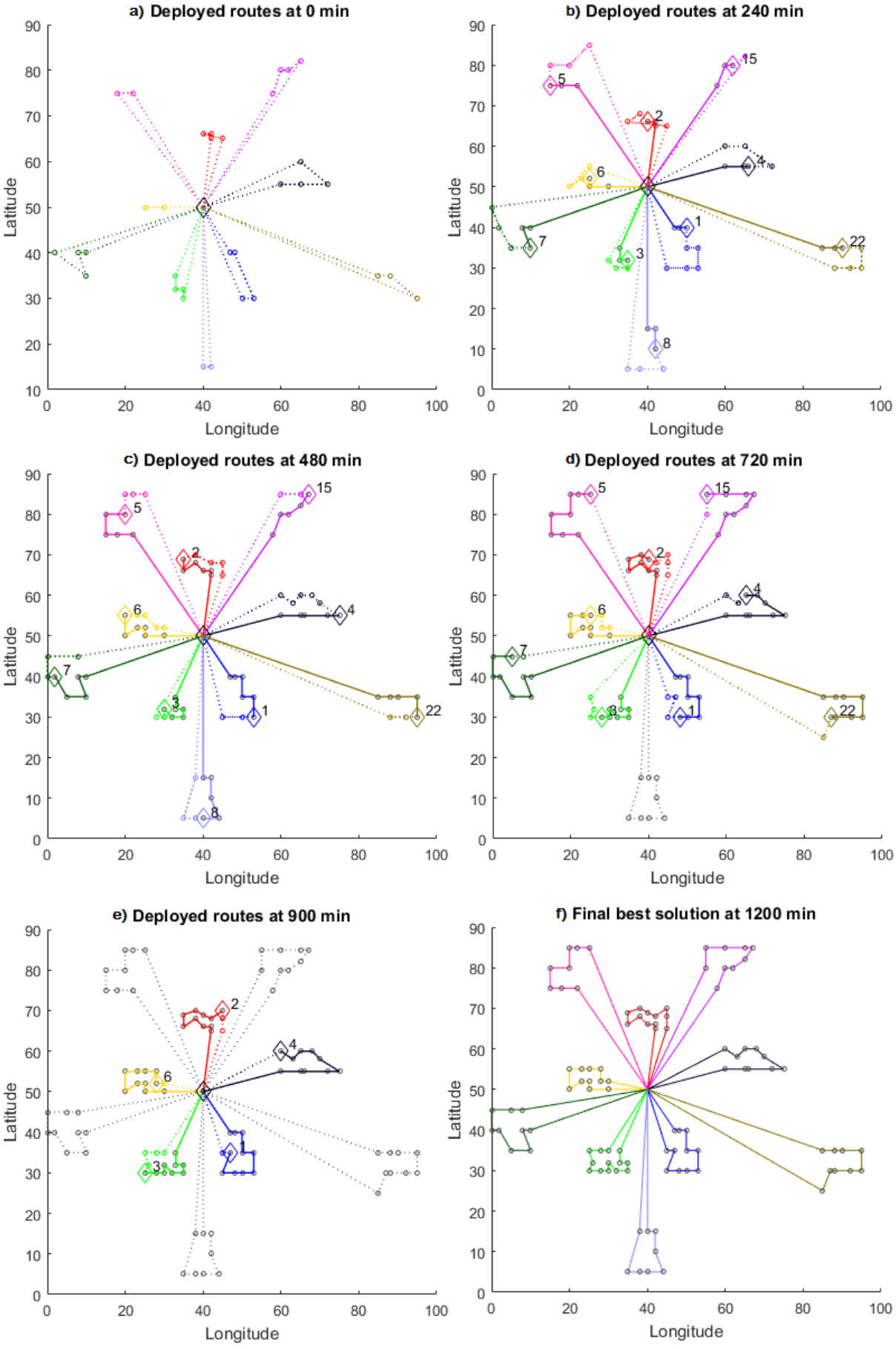
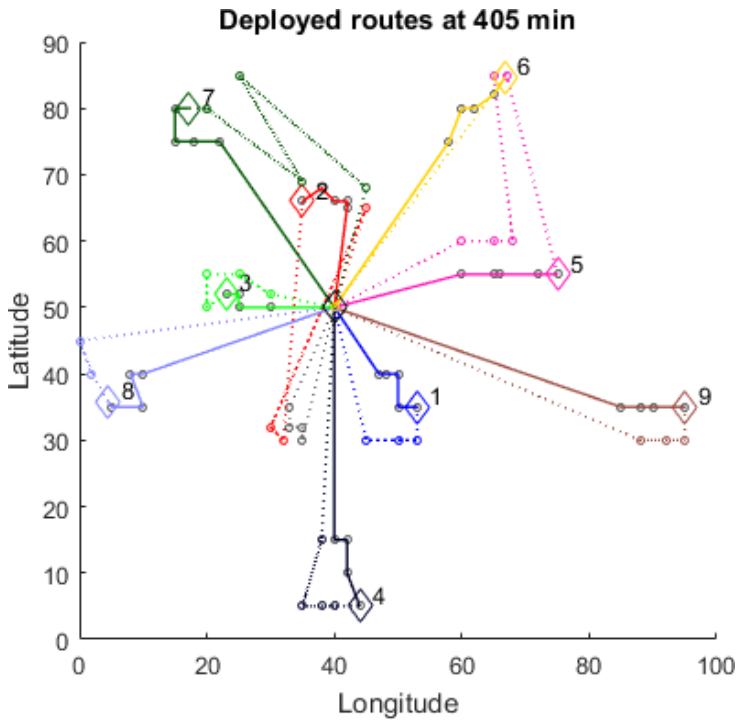


Figure 4.3: Example of a sub-optimal LCI insertion



nodes have been visited at this point and the corresponding deliveries are now tied to the vehicles irreversibly.

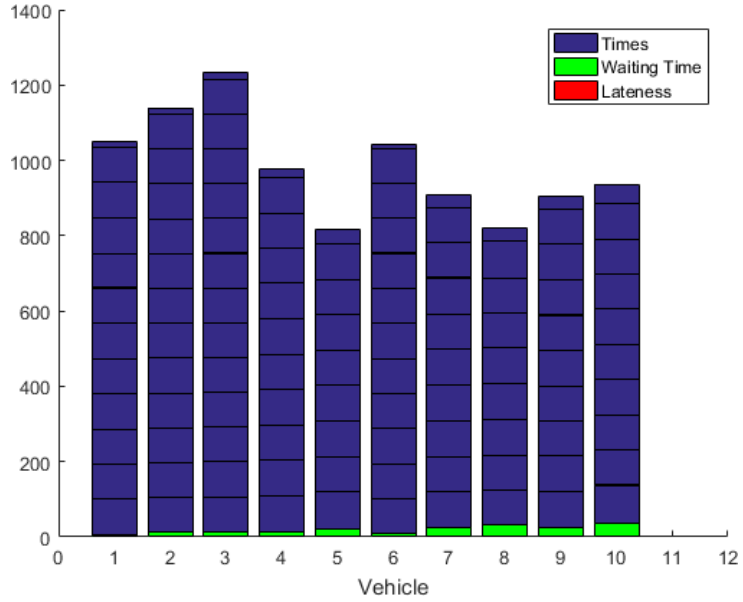
At Figure 4.2 c), in the 480 minute mark, almost all requests have been inserted and vehicles have travelled through some more customers. At this point the speed of the *static solvers* increases greatly since some requests have already been serviced and some others had their respective pickup node visited, decreasing iteration times greatly, as mentioned above.

On Figure 4.2 d) the first vehicle completes its route, and his previously travelled path is displayed in grey. At Figure 4.2 e), minute 900, almost all nodes have been serviced and only 5 vehicles remain active. Finally at Figure 4.2 f) all routes are completed, with all nodes serviced and the vehicles back at the depot. For this case, the achieved routes match the corresponding static solution.

The LCI module is responsible for adding new requests to the current routes at the end of each t_{ss} interval. Since insertion order is defined by the earliest time windows of a request, the LCI might lead to sub-optimal insertions, . If t_{sim} and t_{look} parameters are adequate to the current problem instance, the recently added nodes will only be visited after t_{ss} time, meaning they will pass at least once through *static solver* method.

LCI is not designed to find the optimal insertion, but instead to provide a good and fast response, giving higher priority to the most urgent nodes first. This means a selected insertion move might be sub-optimal, such as depicted on Figure 4.3. Even with well adjusted t_{ss} and t_{look}, a recently added node at a sub-optimal position can be serviced right away, and thus become immutable. When such move is implemented, it is the *static solver*'s job to find the best possible solution given the visited node history. If t_{sim} and t_{look} parameters are adequate to the current problem instance, the chance of adding new nodes to route and service them without passing the *static solver* first decreases.

Figure 4.4: Visualization of dynamic routing for requests, distribution of time in minutes per vehicle



Lastly, it is presented the final time distributions per vehicles at Figure 4.4. Each bar is a unique vehicle and each segment a travel between two locations. As the label says, blue bars represent the vehicle travelling, green bars identify the vehicle waiting at location and red bars, if any, represent lateness of a delivery.

4.5 Case study

The presented case study was proposed by the company UDD, together with most design constraints detailed next. The data is from a restaurant distribution service, meaning all pickups happen at the depot location. Even though this specific problem would be better of modelled as a *one-to-many* PDVRP, our *many-to-many* formulation is still valid by having all pickups at the depot location.

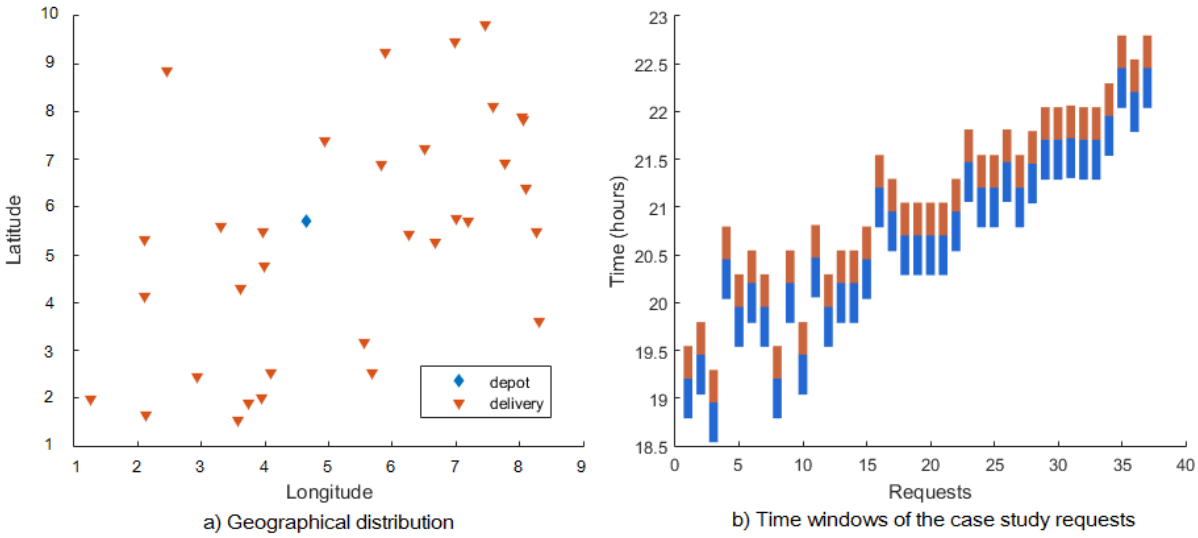
Due to the restaurant's nature, their current assumption for this data is that after a customer makes an order, the delivery time window starts in 45 minutes, which lasts for 15 minutes. Since production is estimated to take 20 minutes, a pickup time window is then defined to start 25 minutes before a delivery's earliest service time. The duration of the pickup is assumed to last indefinitely.

Service time length at the node is assumed to be 1 minute at the pickup and 5 minutes at the delivery locations. For vehicle speed it is used 20 km/h as average. Order weight and vehicle capacity are also specified.

Currently, the distances between customer coordinates are calculated using the Haversine formula as a poor approximation to the actual road distances a vehicle has to travel using the traffic network. However, the created model can work with any matrix giving the distances and travel times between nodes. This means that instead of the Haversine formula one could easily switch to a better alternative, for example the *Google Maps Distance Matrix API*, and apply this model directly to a live problem.

The provided data contains 47 requests to serve from 18:30h to 23h. The geographical distribution

Figure 4.5: Case study dataset plotting



can be seen plotted on Figure 4.5 a) and the corresponding time distribution can be seen on 4.5 b). Since an optimal solution for the tested data is not known, demonstration will be done comparing the dynamic run with the best achieved static solution.

First, the data is processed by the *static solver* module for 30 minutes, similarly to the approach on the static benchmarks. This will give a solution to be considered as the optimal when performing any dynamic tests. The case study data is similar to the benchmarks in size, with 47 requests to service. In terms of scheduling horizon, the case study matches the type 2 files. Visualization of a solution found can be seen on Figure 4.6.

For the static run, 2 vehicles were able to service all requests without lateness for a total of 51.4 km travelled. For the dynamic solution, with a $t_{sim} = 15$ minutes and a $t_{look} = 45$ minutes the solution obtained is represented in Figure 4.7. It uses 4 vehicles instead of 2 and has a total travelled distance of 68.6 km. It also arrives late at one location, but only less than a second after time window end.

Figure 4.6: Static solution for the case study, geographical distribution and service times plot

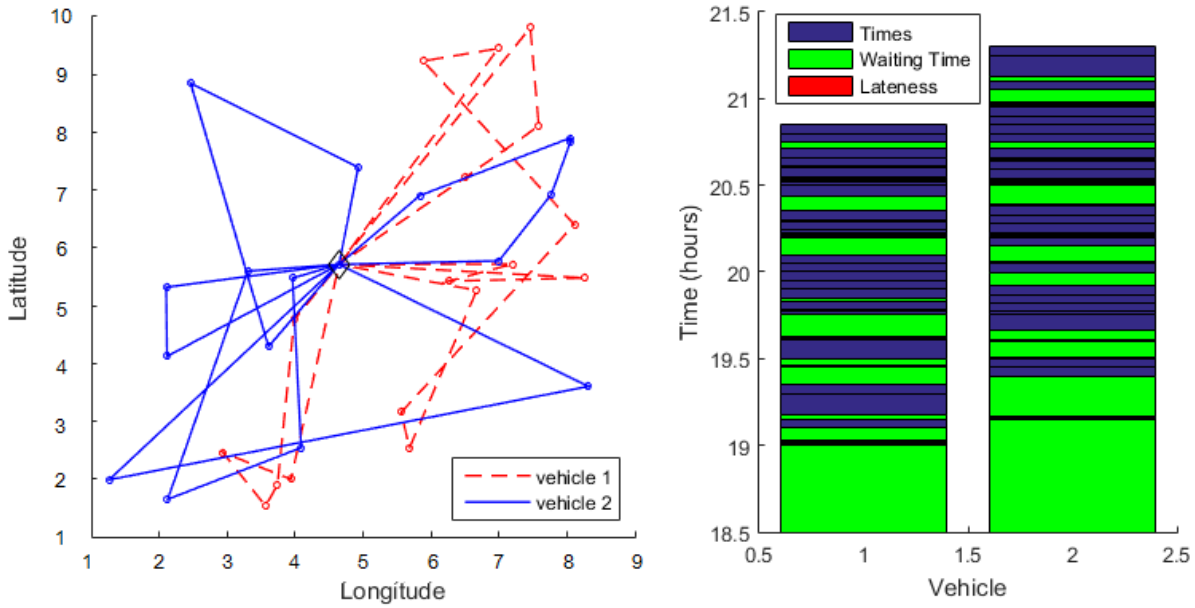
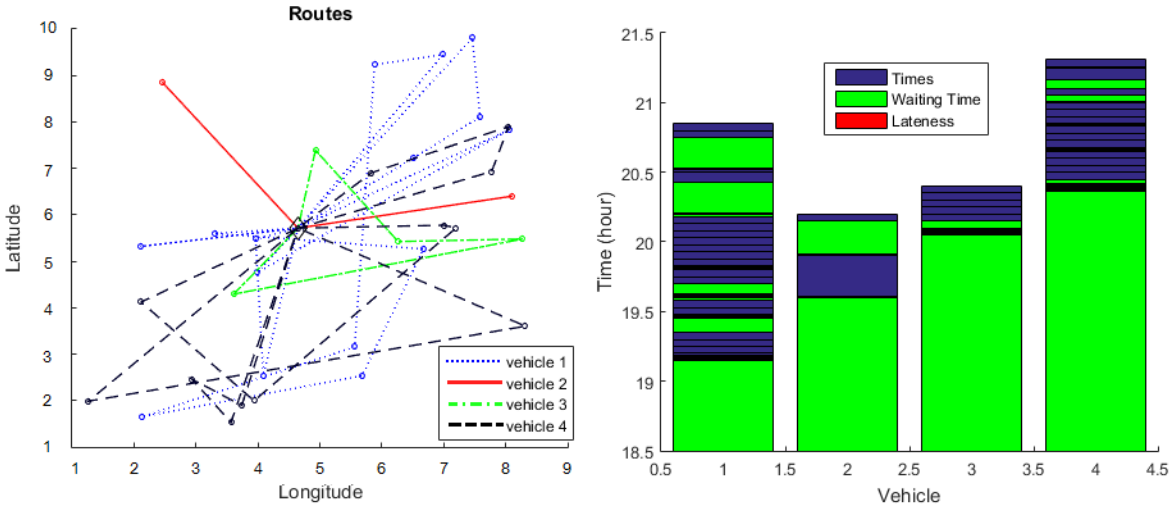


Figure 4.7: Dynamic solution for the case study, geographical distribution and service times plot



Chapter 5

Conclusions

5.1 Conclusions

The main objective was accomplished and a functional model was created to solve Capacitated Pickup Delivery Vehicle Routing Problems with Time Windows on a Dynamic Environment. Further, the proposed approach is suitable for implementation in a real world environment, being able to deal with the tight time windows available to solve such heavily constrained problems. In general, the proposed approach shows a good performance in the validation benchmarks. The introduction of the initial clustering step improved the overall results of the proposed approach, with only little improvements needed to consider them competitive with other approaches from the state of the art.

Considering the benchmark problems where the data is not clustered, the proposed approach does not match the competition when comparing with other multi-vehicle pickup delivery problems, such as the one presented in [27], a commercially available software developed over 20 years by researchers or [32], where the benchmarks are introduced for validation of Pickup Delivery Problems with Time Windows. Even so, for at least two of these benchmarks is still able to find the optimal solution, which means that it is a viable approach.

Under the case study presented, the proposed approach is able to solve the problem, presenting a solution with negligible delays in the delivery time windows.

The developed strategy for initial solution construction seems very promising and worth exploring further outside of this thesis. Pairing the ACO with Local Search proved to be a very good strategy since each method compensates for what the other is lacking. Nevertheless, the overall implemented model is slower than the initial model, due to the local search procedure, but the overall results are better. The used Local Search method really slows down the problem due to a feasibility check that runs for every generated route to the point where more than two thirds of the time is spent doing feasibility checks.

5.2 Future Work

The proposed model can be improved by adding the ability to handle different types of vehicles and to account for the actual distances and times needed to travel through the roads of a city in between the nodes location.

Further, solutions for the case study should be provided to correctly assess what has already been developed and to see if it behaves better or worse than industry solutions.

One of the difficulties of the proposed approach is when the data is unordered, which means that the model of the static solver should be improved to provide better solutions. The first attempt to improve the algorithm would be to further elaborate the Local Search method, namely adding more diversity to different types of solution disturbance. As for the dynamic approach, besides the already mentioned usage of a more complete distance and time matrices and handling a heterogeneous fleet correctly, it is important for a model applied to a real case to refuse new requests if they will badly influence the already accepted routes. Using a more advanced waiting strategy could also lead to better solutions.

Bibliography

[1] United Nations. The World's Cities in 2016 – Data Booklet (ST/ESA/ SER.A/392). Technical report, Department of Economics and Special Affaris, Population Division, 2016.

[2] SINTEF Applied Mathematics. Transportation Optimization Portal - TOP, 2008. URL https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/.

[3] K. Steiglitz and C. H. Papadimitrou. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Mineola, New York, 1982.

[4] M. M. Flood. The Traveling-Salesman Problem. *Operations Research*, 4:61–75, 1956.

[5] G. B. Dantzig and J. H. Ramser. The Truck Dispatching Problem Stable. 6(1):80–91, 1959.

[6] K. Braekers, K. Ramaekers, and I. V. Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 99:300–313, 2016. ISSN 0360-8352. doi: 10.1016/j.cie.2015.12.007. URL http://dx.doi.org/10.1016/j.cie.2015.12.007.

[7] J. R. Montoya-torres, J. López, S. Nieto, H. Felizzola, and N. Herazo-padilla. A literature review on the vehicle routing problem with multiple depots. *COMPUTERS & INDUSTRIAL ENGINEERING*, 79:115–129, 2015. ISSN 0360-8352. doi: 10.1016/j.cie.2014.10.029. URL http://dx.doi.org/10.1016/j.cie.2014.10.029.

[8] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia. A review of dynamic vehicle routing problems, 2013. ISSN 03772217.

[9] G. Berbeglia, J. F. Cordeau, and G. Laporte. Dynamic pickup and delivery problems. *European Journal of Operational Research*, 202(1):8–15, 2010. ISSN 03772217. doi: 10.1016/j.ejor.2009.04.024.

[10] L. H. Lee, K. C. Tan, K. Ou, and Y. H. Chew. Vehicle Capacity Planning System : A Case Study on Vehicle Routing Problem With Time Windows. 33(2):169–178, 2003.

[11] M. Maalouf, C. A. Mackenzie, S. Radakrishnan, and M. Court. A new fuzzy logic approach to capacitated dynamic Dial-a-Ride problem. *Fuzzy Sets and Systems*, 255:30–40, 2014. ISSN 01650114. doi: 10.1016/j.fss.2014.03.010. URL http://dx.doi.org/10.1016/j.fss.2014.03.010.

[12] F. Ferrucci and S. Bock. Real-time control of express pickup and delivery processes in a dynamic environment. *Transportation Research Part B: Methodological*, 63:1–14, 2014. ISSN 01912615. doi: 10.1016/j.trb.2014.02.001.

[13] S. Mitrović-Minić, R. Krishnamurti, and G. Laporte. Double-horizon based heuristics for the dynamic pickup and delivery problem with time windows. *Transportation Research Part B: Methodological*, 38(8):669–685, 2004. ISSN 01912615. doi: 10.1016/j.trb.2003.09.001.

[14] S. Lorini, J. Y. Potvin, and N. Zufferey. Online vehicle routing and scheduling with dynamic travel times. *Computers and Operations Research*, 38(7):1086–1090, 2011. ISSN 03050548. doi: 10.1016/j.cor.2010.10.019.

[15] S. Mitrovi c Minié and G. Laporte. Waiting strategies for the dynamic pickup and delivery problem with time windows. *Transportation Research Part B*, 28:635–655, 2004. doi: 10.1016/j.trb.2003.09.002.

[16] L. M. Gambardella. *Coupling Ant Colony System with Local Search*. PhD thesis, 2015.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Section 24.3: Dijkstra's algorithm. In *Introduction to Algorithms*, pages 595–601. MIT Press and McGraw–Hill, 2 edition, 2001. ISBN 0-262-03293-7.

[18] Scholarpedia. Ant Colony Optimization, 2007. URL `http://www.scholarpedia.org/article/Ant{_}colony{_}optimization`.

[19] D. S. Johnson and L. A. Mcgeoch. The Traveling Salesman Problem : A Case Study in Local Optimization. In *Local Search in Combinatorial Optimization*, pages 215–310. 1997.

[20] L. Bianchi. *Ant Colony Optimization And Local Search For The Probabilistic Traveling Salesman Problem: A Case Study in Stochastic Combinatorial Optimization*. PhD thesis, 2006.

[21] N. Psaraftis. k-Interchange procedures for local search in a precedence-constrained routing problem. 13:391–402, 1983.

[22] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System : Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 26(1):29–41, 1996.

[23] M. Dorigo. *Optimization, Learning and Natural Algorithms (in Italian)*. PhD thesis, Politecnico di Milano, 1992.

[24] A. Colorni, M. Dorigo, V. Maniezzo, D. Elettronica, and P. Milano. Distributed Optimization by Ant Colonies. In *Proc. First Europ. Conf ArtiJicial Life*, pages 134–142, 1991.

[25] A. Colorni, D. Elettronica, P. Milano, M. Dorigo, D. Elettronica, P. Milano, V. Maniezzo, D. Elettronica, and P. Milano. An investigation of some properties of an " Ant algorithm ". In *Parallel Problem Solving From Nature*, pages 509–520, 1992.

[26] M. Dorigo, S. Member, and L. M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *Transactions on Evolutionary Computation*, 1(1): 53–66, 1997.

[27] T. Stützle and H. H. Hoos. MAX – MIN Ant System. *Future Generation Computer Systems*, 16: 889–914, 2000.

[28] G. . Hasle, K.-A. . Lie, and E. Quak. *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*. 2007. ISBN 9783540687825.

[29] D. Taş, N. Dellaert, T. van Woensel, and T. de Kok. The time-dependent vehicle routing problem with soft time windows and stochastic travel times. *Transportation Research Part C: Emerging Technologies*, 48:66–83, 2014.

[30] D. Pisinger and S. Røpke. Large Neighborhood Search. In *Handbook of Metaheuristics*, chapter Large Neig, pages pp. 399–420. Springer, 2 edition, 2010.

[31] D. Naso, M. Surico, B. Turchiano, and U. Kaymak. Genetic algorithms for supply-chain scheduling: A case study in the distribution of ready-mixed concrete. *European Journal of Operational Research*, 177(3):2069–2099, 2007. ISSN 03772217. doi: 10.1016/j.ejor.2005.12.019.

[32] C. A. Silva, J. M. C. Sousa, T. A. Runkler, and J. M. G. Sá. Distributed supply chain management using ant colony optimization. *European Journal of Operational Research*, 199(2):349–358, 2009. ISSN 0377-2217. doi: 10.1016/j.ejor.2008.11.021. URL `http://dx.doi.org/10.1016/j.ejor.2008.11.021`.

[33] J. S. Jang, C. T. Sun, and E. Mizutani. *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, New Jersey, 1997.

[34] H. Li and A. Lim. A Metaheuristic for the Pickup and Delivery Problem with Time Windows.

Appendix A

Flowcharts

A.1 Static solver

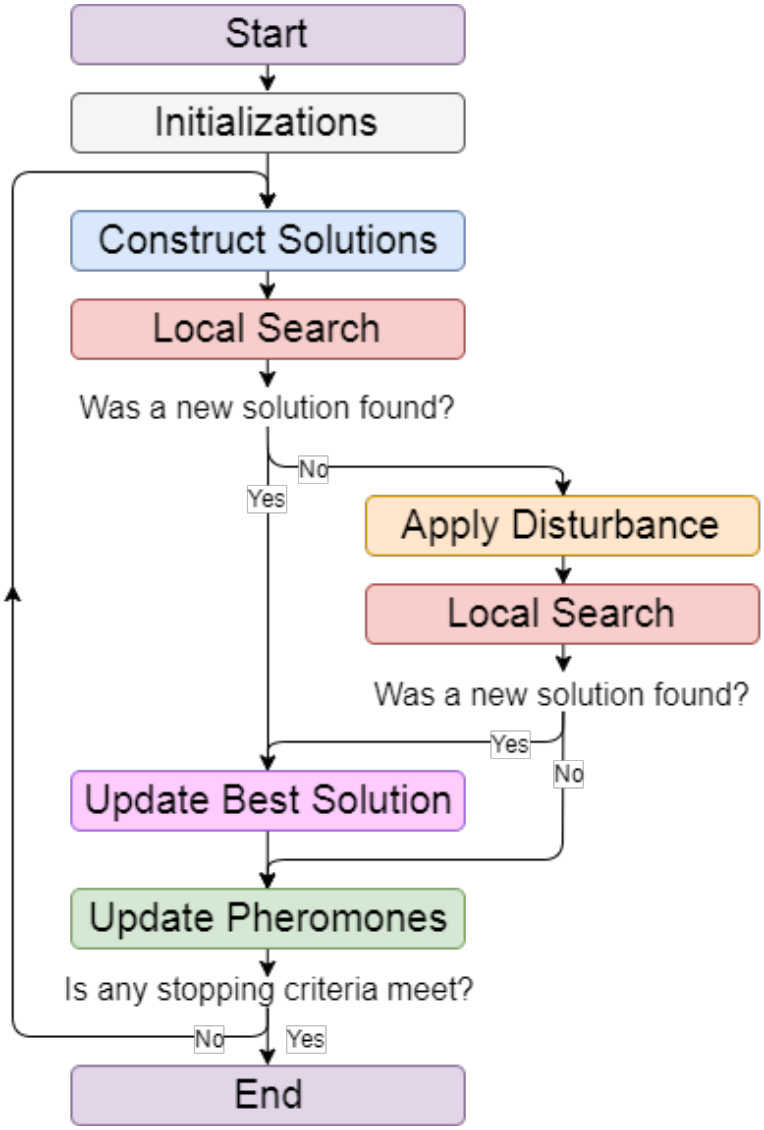


Figure A.1: Static solver flowchart

Appendix B

Tables

B.1 Absolute and Relative errors

Table B.1: Absolute and relative error for the clustered files

File	Mean Values				Best Values			
	Vehicles		Distances		Vehicles		Distances	
	e_{abs}	e_{rel}	e_{abs}	e_{rel}	e_{abs}	e_{rel}	e_{abs}	e_{rel}
lc101	0	0	0	0	0	0	0	0
lc102	0	0	0	0	0	0	0	0
lc103	1	11.11	-207.29	-20.02	1	11.11	-207.29	-20.02
lc104	0.8	8.89	4.96	0.58	0	0	165.94	19.29
lc105	0	0	0	0	0	0	0	0
lc106	0	0	0	0	0	0	0	0
lc107	0	0	0	0	0	0	0	0
lc108	0	0	0	0	0	0	0	0
lc109	1	11.11	-171.12	-17.10	1	11.11	-172.78	-17.27
lc201	0	0	0	0	0	0	0	0
lc202	0	0	0	0	0	0	0	0
lc203	0	0	0	0	0	0	0	0
lc204	0	0	35.34	5.98	0	0	0	0
lc205	0	0	0	0	0	0	0	0
lc206	0	0	0	0	0	0	0	0
lc207	0	0	0.03	0.01	0	0	0.03	0.01
lc208	0	0	7.67	1.30	0	0	7.67	1.30

Table B.2: Absolute and relative error for the randomly distributed files

File	Mean Values				Best Values			
	Vehicles		Distances		Vehicles		Distances	
	e_{abs}	e_{rel}	e_{abs}	e_{rel}	e_{abs}	e_{rel}	e_{abs}	e_{rel}
lr101s	1.5	7.89	114.70	6.95	1	5.26	44.32	2.68
lr102s	0.2	1.18	104.13	7.00	0	0	66.91	4.50
lr103s	0.7	5.38	139.20	10.77	0	0	45.33	3.51
lr104s	2	22.22	155.88	15.38	2	22.22	92.79	9.16
lr105s	0.7	5	54.98	3.99	0	0	0.00	0.00
lr106s	1.1	9.17	114.36	9.13	0	0	32.38	2.58
lr107s	1.3	13	140.92	12.68	0	0	24.14	2.17
lr108s	1.3	14.44	155.85	16.08	1	11.11	78.17	8.07
lr109	2	18.18	159.16	13.17	1	9.09	56.72	4.69
lr110s	2.1	21	-12.74	-1.10	1	10	-1146.35	-98.88
lr111s	1.1	11	61.51	5.55	0	0	17.10	1.54
lr112s	1.9	21.11	207.58	20.68	1	11.11	102.21	10.18
lr201	1.6	40	226.46	18.07	1	25	101.19	8.07
lr202s	1.4	46.67	323.92	27.05	1	33.33	152.77	12.76
lr203s	0.6	20	306.13	32.24	0	0	135.74	14.30
lr204s	1.2	60	402.07	47.35	1	50	237.32	27.95
lr205s	0.6	20	136.35	12.94	0	0	0.12	0.01
lr206s	0.67	22.22	516.72	55.46	0	0	360.33	38.68
lr207s	1.38	68.75	558.89	61.89	1	50	309.77	34.30
lr208s	0.6	30	399.48	54.36	0	0	161.03	21.91
lr209	0.5	16.67	205.27	22.06	0	0	130.56	14.03
lr210s	1.1	36.67	393.16	40.78	1	33.33	265.31	27.52
lr211	1.2	60	165.15	18.12	1	50	28.98	3.18

Table B.3: Absolute and relative error for *lcr* files

File	Mean Values				Best Values			
	Vehicles		Distances		Vehicles		Distances	
	e_{abs}	e_{rel}	e_{abs}	e_{rel}	e_{abs}	e_{rel}	e_{abs}	e_{rel}
lrc101s	3.2	22.86	114.70	21.43	3	199.26	44.32	114.12
lrc102	3.4	28.33	104.13	16.67	2	239.57	66.91	182.13
lrc103s	0.2	1.82	139.20	0	0	65.10	45.33	39.20
lrc104s	1.2	12	155.88	10	1	200.60	92.79	115.93
lrc105s	4.2	32.31	54.98	23.08	3	253.28	0	175.40
lrc106	2	18.18	114.36	18.18	2	147.83	32.38	112.50
lrc107	0.6	5.45	140.92	0	0	54.65	24.14	0.01
lrc108	1.2	12	155.85	10	1	133.33	78.17	51.82
lrc201	2	50	159.16	25	1	393.12	56.72	237.21
lrc202s	2.4	80	-12.74	66.67	2	424.51	-1146.35	313.86
lrc203s	1	33.33	61.51	33.33	1	534.88	17.10	394.56
lrc204	0.2	6.67	207.58	0	0	238.54	102.21	50.55
lrc205s	1.6	40	226.46	25	1	409.90	101.19	273.85
lrc206	1.6	53.33	323.92	33.33	1	359.22	152.77	187.21
lrc207	1	33.33	306.13	33.33	1	327.94	135.74	182.65
lrc208s	1	33.33	402.07	33.33	1	285.98	237.32	174.22