Solving the multi-objective Dial-a-Ride problem without using routing heuristics

---

Master's thesis

Jip Man Vuong

Solving the multi-objective Dial-a-Ride problem without using routing heuristics

THESIS

submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jip Man Vuong

born in Dordrecht, the Netherlands

Algorithmics Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Solving the multi-objective Dial-a-Ride problem without using routing heuristics

Author:	Jip Man Vuong
Student id:	1263234
Email:	jipmanvuong@gmail.com

Abstract

Door-to-door transportation services are very important for disabled or elderly people, as they have difficulties using regular public transport services. These services work on a on-demand basis and use taxi vehicles that can service multiple jobs at the same time to reduce the operational costs. The problem however is that deciding how to schedule each taxi to pickup and drop-off a customer is very difficult, as there are multiple objectives involved such as operational costs, vehicle efficiency and service quality (which itself is composed of several subobjectives). Since these objectives are related to each other it is very difficult to balance them. Two major decisions in this problem are the clustering (assigning customers to taxis) and routing (computing the driving schedule of each taxi). In this thesis we introduce the Random Insertion Genetic Algorithm (RIGA) that solves both clustering and routing using only a genetic algorithm. We show that this algorithm may outperform other (genetic) algorithms that do rely on a specialized routing procedure.

Thesis Committee:

Chair:	Prof. dr. C. Witteveen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M.M. de Weerdt, Faculty EEMCS, TU Delft
Committee member:	Dr. ir. P. Wiggers, Faculty EEMCS, TU Delft
Guest member:	Dr. P. Bosman, CWI

Preface

This thesis would not exist if it weren't for a few very important individuals who helped me over the course of this project.
First of all, I would like to thank my supervisor Mathijs de Weerdt for his assistance in realizing this thesis. His extensive knowledge of the dial-a-ride problem in general as well as his ability to dispel my occasional procrastinatory fits were essential in getting anything done.
Secondly, I would like to thank dr. Peter Bosman for sharing his knowledge of genetic algorithms and multi-objective optimization as well as his very insightful comments. Particularly, his suggestion to disregard binary representations for population elements was a very important one.
My gratitude goes out to Cees Witteveen and Pascal Wiggers for being in the committee and spending time to read this thesis, even though I am sure they must have many other matters to attend to.
Furthermore, I would like to thank my parents for giving me the opportunity to study in the first place and for making sure I got some food/sleep during busy periods as I tend to forget about both when I'm busy.
Finally, I would like to thank the reader (you) for taking the time to read this thesis and I hope that you will find its contents interesting (or at least not a complete waste of your time).

Jip Man Vuong
Zwijndrecht, the Netherlands
July 20, 2011

Contents

Part I

Introduction

Chapter 1

The Dial-a-ride problem

In most developed countries most people usually have some form of personal transportation like cars or bikes. The less fortunate are dependent upon the public transport services, that were designed to move around as many people as possible, as profitable as possible. This means that these services are not very flexible and require the passengers to cope with unnecessary delays caused by fixed arrival/departure times, transferring vehicles (trains, buses, subway, etc.) and fixed pickup/drop-off places. These problems present a huge problem for elderly and disabled people, who may not be able to cope with these limitations. Another potential issue with the transport of elderly and disabled people is that they may have special needs with regard to transport, examples of which may include the need for special equipment, different seating possibilities, etc.

To cope with the particular requirements of this group, taxi-agencies exist. These companies work on a on-demand basis, which means that one must contact the company in order to request a taxi for a trip. Since sending a taxi for each individual client is too expensive, these services operate by having customers share the taxis. In the Netherlands, the elderly and the disabled usually do not have to worry too much about the costs, since those are covered by their insurance or through government funding. The downside is that due to government involvement (long term contracts auctioned off to the company bidding the best/cheapest service), the quality of these services is often lower than what was agreed upon because companies are usually too optimistic and overestimate their capabilities. Once a company secures a contract, it becomes difficult for the government to penalize low quality since this leads to even worse quality (or even bankruptcy), so the only remaining option is to maintain the status quo.

When taking above situation into consideration from the perspective of a taxi-company, it is very important to be able to accurately estimate the quality of service it can provide given some budget. If the estimate is too high, they may not be fined, but the poor service that was delivered will most likely lower their chances of winning the next auction. On the other hand, a too conservative estimate will affect their chances of winning the auction in the first place, since the number of better bids is higher. If all companies in the auction can provide good estimates then actual quality will be close to that which was promised, resulting in satisfied customers.

The quality of service reflects how well the company can assign its requests for trips to its vehicles such that the quality (from the client's perspective) of each trip is good. Note that the company usually does not know all requests it needs to fulfil at the beginning of a working day as they can arrive during any time of the day. After receiving a request for a trip, the taxi-agency must find out to which vehicle of their fleet they will assign to this request, this procedure depends on the state

of the vehicle (location, jobs already assigned to, available capacity, etc.) and whether this taxi can serve the job within the time constraints given by the customer (pickup not before a certain time, arrival not after another time). This assignment is a very important decision, because it influences the experience of other passengers on the same vehicle (multiple customers share the same vehicle) as well as the quality the new passenger may experience. After assignment, the actual route the vehicle must take must also be changed to incorporate the new customer. This problem is known as the Dynamic Dial-A-Ride Problem with Time Windows (DDARPTW), where dynamic refers to uncertainty of when new requests will arrive and time windows are the earlier mentioned time constraints.

Currently many algorithms solve the problem by first solving the assignment and then use some heuristic to generate a vehicle route, this strategy is called cluster-first-route-second. However, we believe that a more integrated variant of this strategy may yield better results as assignment and routing are not independent subproblems. As such, the question we try to answer in this thesis is as follows:

> "Can we improve the service quality for Dial-a-Ride problems by using a more integrated variant of the traditional cluster-first-route-second method?"

In the following chapter we begin with a brief discussion of existing work and identify the contributions we are going to make.

Chapter 2

Previous work and our contribution

As the dial-a-ride problem is quite similar to other routing problems, a lot of prevous work exists on the problem. Baugh et al.[3] presented an algorithm that uses simulated annealing and tabu-search for the static problem that solves the clustering by using simple operations that move one or two customers around. One of their most interesting contributions is the space-time-nearest-neighbour heuristic that computes the routes by constantly selecting the closest next stop in terms of space as well as (violated) time constraints. This routing heuristic was also used by Jørgensen et al.[14] in their genetic algorithm that also uses a cluster-first-route-second strategy where clustering and routing are strictly separated. The clustering is done by expressing a cluster as a binary template and combining it with another solution using a method based on bitmasking. Another approach where clustering is not done using bitmasks is presented by Borndörfer et al.[5]. Cluster-first-route-second is based on the idea that clustering has a greater effect on the overall solution quality than routing[1, 3], so it is not a strange idea to do that first.

Cordeau et al.[7] use a tabu-search algorithm that starts with partially randomly generated solutions (random cluster, simple sequential insertion of stops into the routes) moves around within the neighborhood of its search space. A tabu-list is used to prevent cycles by banning recurring solutions from being considered. The randomly generated problem instances used to test their solver are publicly available, so they serve as an important way to compare algorithm results. Even more interesting is the fact that Jørgensen et al. also use the same instances, which means that there are even more results available to compare to.

Xiang et al.[20] describe an interesting method that models the dynamic problem using an event-driven approach, where each new piece of information is considered as a new event and handled appropriately. Furthermore, they experiment with a simple insertion heuristic that finds the best insertion for a stop into some existing route in a optimal way given the stop and the route. As far as insertion heuristics are concerned, there is the well-known algorithm by Jaw et al.[12] that sequentially inserts new jobs into its existing solution quickly by identifying schedule blocks and computing additional time windows for each stop to facillitate faster searching. Madsen et al.[15] use a modified version of the algorithm by Jaw et al. to support multiple capacity types and multiple objectives by defining the objective function as weighted sum of the subobjectives.

A survey performed by Cordeau and Laporte[13] shows that not many approaches focus on the combination of cost and quality objectives. Since we have multiple objectives we need to make sure that our solution focuses on all of these objectives that may or may not conflict with each other. Furthermore, Paquette et al.[16] note that it is important to take into consideration the immeasurable

aspects of quality such as the customer experience. Even though we may not be able to express the experience in measurable values, this reminds us that it is really important to consider how the multiple objectives are combined. This process of dealing with several objectives is also called multi-objective optimization.

Coello et al.[6] point out several characteristics of objective functions that need to be taken into consideration. They make the distinction between *commensurable* (measured in the same units) and *non-commensurable* (measured in different units) objective functions. In the DARP objective functions are usually non-commensurable since operational cost has to be balanced against quality, where the latter is often formulated using time-related measures.

A simple approach to multiobjective optimization would be to assign weights to each of the attributes of the objective function to signify which attribute is more important than the other and take the sum of these weighted attributes as the final value. An important reason for taking the sum of the attributes is that it is the most straight forward way to aggregate the separate attributes. An example of this can be found in the paper of Jørgensen et al.[14] that presents a genetic algorithm that uses this method to create a objective function that uses seven weights for each of the attributes that indicate their importance for the function as a whole. The advantage of this approach is that this method will provide some solution that has the best value according to the objective function. However, for this method it is very important to choose the weights properly because they define the relation between the various attributes. Another problem is that this is especially difficult when we have to deal with a non-commensurable function because in that case even the units are different.

A well known method to solve a non-commensurable multi-objective problem using genetic algorithm is the NSGA-II algorithm by Deb et al.[8]. Since this algorithm uses Pareto optimality to select its best solutions it does not need to compute some overall value for the solution quality, solving the problem of non-commensurable objectives. Their algorithm assigns domination ranks to population elements and selects the undominated ones accordingly. When there are too many elements on the Pareto front than is needed, another method is used to select the most important elements of a frontier. The crowding distance as defined by Deb et al. is a metric that determines how far away a solution is from its neighboring solutions in terms of objective values. Another characteristic of the crowding distance metric is that the endpoints of a front are always given the highest crowding distance value, which means they are at the top of the list after sorting. The goal of this metric is to select solutions that are far away from other solutions to maintain diversity (which explains why the endpoints of a front are given the highest priority).

Parragh et al.[18] describe a variant of the DARP that finds a Pareto frontier of solutions optimizing for cost and service quality using variable neighborhood search. However, service quality itself is expressed as a sum of weighted subobjectives even though we see no reason not to compare the subobjectives separately. Interestingly, they propose another VNS algorithm in [17] that claims to achieve an average improvement of 71% over results by Jørgensen et al. However, the way they compared their results is not completely fair in our opinion as they use different (faster) hardware while their fixed iteration limit of 250000 sometimes results in CPU times quite close to the ones found by Jørgensen et al.

As stated in the introduction we want to improve the solution quality by using a more integrated approach to the cluster-first-route-second strategy. Because there are only experimental results available for the static variant of the dial-a-ride problem we first test our idea on the static dial-a-ride problem to see whether it is viable. So the contributions we hope to make in this thesis are as follows:

- We describe a strategy based on cluster-first-route-second that is less rigid with respect to separating the clustering and the routing elements of the problem in a genetic algorithm, which means

that we should be able to search through a larger solution space while we do not significantly increase the amount of computation needed.

- We define a crossover operator that enables us to directly operate on the two parent solutions without the need for an alternative representation such as bitstrings. Furthermore, even though Jørgensen et al.[14] claim that 'elaborate fix-ups' would be required to handle precedence constraints, we show that a simple and intuitive method exists to enforce the precedence constraints during crossover.
- We describe a method to solve the routing part of the dial-a-ride problem by only using genetic algorithms and local search instead of routing heuristics, demonstrating that genetic algorithms can actually do a lot of the work in solving the DARP.
- An attempt is made at a more fair comparison between our results and those by Jørgensen et al. We also provide a crude comparison to the results by Parragh et al.[17] as their improvement of 71% is indeed impressive, even if the comparison may not have been completely fair.

2.1 Thesis outline

The next part is mainly concerned with the problem definition, which means we discuss what a problem instance of the dial-a-ride problem looks like as well as the constraints on the results. Each element of the dial-a-ride problem is discussed separately and the design choices leading to the implementation are also presented. The final part is dedicated to the implementation of our genetic algorithm and consists of an in-depth comparison/discussion of our empircal results. We will end this thesis with a summary and point out some interesting subjects for further research.

Part II

Problem definition

Chapter 3

Introduction

A complete instance of the DDARPTW consists of all information on the state of the road network, taxis, customers, and events. However, this information is not completely available to the solver. For example, a new job may come in at any time or some traffic jam may occur. The problem that we need to solve is that given the above information, we need to know which taxis to assign to which customer and when the taxi should service which customer. More concretely, the output of a solver consists of a schedule for each taxi that describes a list of places the taxi has to visit and at what time the taxi should arrive at those locations. For our model we assume there is some fixed number of taxis during the execution of the algorithm and that all taxis start their routes from the depot l_0. From a cost perspective we assume that there is a one-time cost incurred when the taxi leaves the depot. When a taxi is moving the costs for that movement depend on the distance traveled multiplied by some fixed cost per unit distance. Alternatively, the costs can be calculated using the travel time in a similar manner.

Ideally, we want a solver that produces a schedule that maximizes the quality for customers while making sure that the costs are acceptable. Since we are dealing with online optimization, there should be some iterative element in the solver that continuously reads (new) input and produces a new output. This also means that some portion of the output of one iteration may also become part of the input of the next one. To summarize, during some iteration of the computation the following information is considered as input:

- State of the road network
- Existing schedules
- State of the taxis
- Existing and new jobs
- Existing and new events

The output of a solver consists of the newly (re)computed schedules. However, it has to be noted that a schedule (if it is executed) also implicitly changes the state of the road networks, taxis, the jobs and possibly even the events that may occur. For example, a schedule directly influences the remaining capacity of a taxi for future iterations of the solver. When a taxi has completed a job, the realized quality of that job can be computed instantly and can be used to evaluate the overall quality of the

schedule. So if we talk of a solution for the DDARPTW we mean the schedule that was computed during the final iteration of the algorithm, because by then the schedule has accounted for all available information and no more changes will occur.

The above problem definition can be easily adapted to a static model (DARPTW) with the most problematic part being the events. However, since we know all of them in advance we can imagine that they can be resolved simply by modifying the jobs to account for them. For example, cancellation events can be omitted as we can compensate by removing the corresponding job. The other simplification we make is the disregarding of the time arguments in our distance function, which means that travel times are considered constant for any given pair of locations.

These simplifcations are necessary as we otherwise will not be able to make a good comparison to the work by Jørgensen et al.[14]. The major advantage of dealing with the static problem is that we only need one solution (the final one), whereas in the dynamic variant we need to compute one after every important event. The model we use can be easily adjusted to be compatible with most static problem definitions proposed by many researchers[17, 20, 7, 12]. Finally, this model can be also be adapted for use in a dynamic setting, which is further discussed in the chapter on future work.

Chapter 4

Specific model elements

This chapter covers the five most important elements of our model of dial-a-ride problem (except for the objective function). The main question for each of these elements is to decide which assumptions to make as well as how to implement them for our experiments. These elements are as follows:

- Modeling the road network
- Deciding on what a solution looks like
- Defining what a vehicle is and how it operates
- Specify what a job looks like
- How to handle unexpected events

The following sections cover each of the elements mentioned in more detail. However, as stated in the previous chapter the events are not implemented as we are experimenting with a solver for the static problem. The objective function for our model is discussed in the next chapter.

4.1 Road network and map

For the map we assume a Cartesian coordinate system, so any *location* on the map can be defined by a tuple (x, y) where $x, y \in \mathbb{R}$. We further assume that the locations given by customers are accessible by the taxi, so the distance between the pickup/dropoff location and the actual destination of the customer is neglibigle. For the travel time between two locations we assume the existence of the following function:

Definition 4.1.1. The *travel time* needed to move from one location l_1 to another l_2 is the value of the function $t(l_1, l_2, t_1, t_{now})$ where t_1 is the time at which the entity wishes to start travelling and t_{now} is the time at which the computation was performed.

We think it is necessary to have a variable like t_{now} because otherwise we cannot make a distinction between a time computed earlier and one computed right now. It is important to make this distinction because the road network may change after computing a result. This is also the reason why one can consider the new state of the road network to be part of the output as well, as a change in the schedule may lead to a change in the road networks. Furthermore, it is reasonable to assume that if we estimate

the travel time a few days in advance it may be less accurate than if we would do so one hour in advance.

Note that the additional functionality by supporting the arguments t_1 and t_{now} means our model will be able to deal with variable traveltimes, which should allow the user to account for rush hour conditions etc. This feature is also present in the model used by Xiang et al.[20].

However, for our experiments we drop the additional parameters allowing for variable travel times, which means the resulting function is now $t(l_1, l_2)$. This actually does not simplify the algorithm a lot, but does have a positive effect on the speed of the algorithm.

As a basic method we express the distance between two points (x_1, y_1) and (x_2, y_2) using taxicab geometry (also known as the Manhattan distance), which means that $|x_1 - x_2| + |y_1 - y_2|$ computes the distance where $(x_1, y_1) = l_1$ and $(x_2, y_2) = l_2$. We think this is a more appropriate method because taking the minimum distance between the two points using the Pythagorean theorem seems to imply the existence of a direct path between any two locations. The resulting value is the multiplied by some constant α to get the travel time in time units. Thus the function that describes the distance is denoted as $t((x_1, y_1), (x_2, y_2)) = \alpha \times (|x_1 - x_2| + |y_1 - y_2|)$. As Jørgensen et al.[14] use $\alpha = 1$ and the same function we also need to use this function to keep our results comparable.

Ichoua et al.[11] introduce a property called the "first-in-first-out" (FIFO) property. This property guarantees that if a vehicle travels from node A to node B, any other identical vehicle doing the same at a later time will arrive after that first vehicle. This situation only arises when variable travel times are possible since it is caused by travel times changing after a vehicle has left and can be fixed by updating the travel time for each vehicle as the value changes. However, whether this property should hold is debatable as we can imagine situations where this property should not hold. For instance, if a taxi takes a great detour because a road is temporarily blocked it may be overtaken if another taxi finds the road open again. Nevertheless, this issue is important to keep in mind when we want to deal with variable travel times.

4.2 Schedules

The goal of the resulting algorithm is to produce a schedule for each vehicle in the fleet. A schedule is nothing more than just a list of stops for each vehicle to visit. For our work we make the following additional assumptions:

- All taxis in the fleet start their operations at the same time from the same initial stop (depot) and end at the same stop.
- All drivers may decide for themselves when to leave the depot to pickup their first customers. However, the schedule dictates at what times they have to arrive.

The first assumption is used because we can imagine that the taxi drivers need to go to the depot anyway to pickup their vehicle. Furthermore, assuming all vehicles may start at the same time simplifies the generation of solutions as it does not need to handle different working hours. As each driver is supposed to act independently, it is reasonable to suggest that some drivers need to leave earlier if their stop is farther away. Given the above assumptions the notation of a schedule is as follows:

Definition 4.2.1. A *schedule* for one vehicle can be denoted as
$R = \{(t_1, l_1, j_1), (t_2, l_2, j_2), \ldots, (t_n, l_n, j_n)\}$ for some value of n where t_i and l_i denote the desired arrival time and location, j_i is the job that is supposed to be serviced at that particular location and time.

However, at each stop the taxi needs some amount of time to actually make the pickup/dropoff which should be accounted for. This is done by associating a service time $sv(l)$ which indicates how long service takes or how long the taxi must wait before it can depart from l. A solution for the DARP thus can be defined as a set of schedules $S = \{r_1, \ldots, r_i\}$ where r_i is a schedule for vehicle i. Note that a schedule is only considered feasible if it satisfies all precedence constraints and the capacity constraints of all vehicles. A precedence constraint enforces that pickup locations are visited before destination locations and a capacity constraints means that a vehicle can never be scheduled to have more people on board than its maximum capacity allows. A good objective function will automatically provide us with a way to filter out 'bad' solutions.

By choosing to model our schedules this way we have turned the job related constraints into soft constraints. Many authors make a distinction between hard and soft constraints (especially for time windows)[14, 20, 9, 3]. Soft time windows are useful when evaluating the tradeoffs between service requirements and their costs. By imposing some amount of penalty when a soft constraint is violated, any further violation is discouraged. Intuitively, a soft constraint with a very large penalty is quite comparable to a hard constraint in practice, in the sense that hard constraints may be violated in real life, but may have such severe consequences that it is not practical/sensible to do so in most cases. This is somewhat similar to the triangular fuzzy numbers described by Teodorovic et al. [9] to denote desired pick-up times. In their situation, constraints are also soft in the sense that they allow a certain interval of values with different strengths (preference), however, a major difference is that no penalization occurs, since only values within the fuzzy set may be selected (which, itself is a hard constraint). On the other hand using hard constraints allows an algorithm to skip large portions of the search space and focus only on the really good solutions. Parragh et al.[17] present such an algorithm and shows that solutions of solvers that use soft constraints can be improved upon significantly by turning all constraints into hard ones. Obviously, these algorithms only work if there is a solution available that does not violate these constraints.

Apart from just time constraints imposed by the jobs there are also other limitations, in the next section we take a look at the capacity constraints for vehicles and in a later section we will discuss the precedence constraints.

4.3 Taxi vehicles

Given our situation, our model of a taxi needs to deal with different types of capacity. This means that the vehicle provides a value for each type of capacity that denotes its maximum capacity for that type. An example of this would be the number of seats. We propose to denote this as follows:

Definition 4.3.1. The *capacity* of a taxi is denoted as a tuple $c = (c_1, \ldots, c_k)$, where each $c_i \in \mathbb{Z}$ represents a type of capacity (number of seats, wheel chair places, lying seats, etc.). Note that $c_i = 0$ is allowed, as this means that the taxi does not provide for the type of capacity at all.

We believe that it is reasonable to use natural numbers to denote capacity because this is the case in many situations (for more examples, see [15]) and it simplifies the reasoning while computing a schedule. A constraint related to the capacity of a vehicle is that a schedule should keep in mind that it is not possible for a vehicle to carry more passengers than its capacity allows. This can be done by returning the capacity a customer uses to the taxi when the passenger arrives at his destination.

Definition 4.3.2. The *capacity constraint* ensures that a vehicle does not exceed its maximum capacity. If the change in capacity at each location in the schedule is denoted as $d = (\pm c_1, \ldots, \pm c_k)$

and the whole list of changes is denoted as $D = \{d_1, \ldots, d_n\}$, a vehicle satisfies this constraint if $\sum_{i=1}^{m} d_i \geq (0, 0, \ldots, 0)$ for any $m \leq n$.

A *taxi* is represented by a tuple $v = (l_{now}, l_t, c, R)$ where l_{now} is the actual location of the taxi, l_t is the current destination of the taxi, c is the remaining capacity of the taxi and R is the schedule such that there is some t_i such that $(t_i, l_t, j_i) \in R$. Note that the actual location of the taxi does not have to be in the schedule, as it only defines the points the vehicle has to pass through. Furthermore, this location is only relevant if we are dealing with the dynamic problem, as the 'inbetween locations' of a route are simply non-existent otherwise.

Although our model does allow for multiple capacity types, this feature is not used by Jørgensen et al. and as such we have also reduced the number of capacities to one, although this means our algorithm cannot handle different types of seating. However, we can imagine that this can be somewhat accounted for by expressing different capacity types in terms of each other. Nevertheless, using multiple capacity types does not increase the runtime complexity of the algorithm significantly.

4.4 Customers and jobs

Before we can define our model we need to have clear understanding of what we mean by a customer. A *customer* corresponds to one person or entity that submits a job that can require some amount of each of the various capacities. At this point there is little need to view customers and jobs as separate entities. As such, a job is defined as follows:

Definition 4.4.1. A *job* represents the customer's needs and can be represented by a tuple $j = (c, t_{now}, l_s, l_t)$ where l_s and l_t are tuples (l, t_a, t_b) where l is a pair of coordinates, t_a and t_b are the earliest and latest times allowed for the serice to begin at the stop. The time at which the job request was submitted is t_{now} and the required capacity is c. The time window specified is soft in the sense that solutions may contain schedules that violate them.

A job request also implies a precedence constraint between l_s and l_t during scheduling, since one cannot deliver a passenger to his destination if he wasn't picked up first. This can be formulated in the following way.

Definition 4.4.2. The *precedence constraint* requires that for any job request the pickup location must be accessed before the arrival location by the same vehicle, or: A job request $j = (c, t_{now}, l_s, l_t)$ that is scheduled in a schedule R satisfies the precedence constraint if $R = \{\ldots, (t_{s'}, l_s, j), \ldots, (t_{t'}, l_t, j), \ldots\}$ holds.

In addition to this hard constraint both locations should be visited within the specified time window, which is a soft constraint. Futhermore, slight variations from the time window are allowed but penalized by the objective function. It has to be noted that job requests may not be rejected and we assume that time windows are reasonable in the sense that they can be realized. Finding these time windows is outside the scope of our problem, but one can imagine that the service operator can assist the customer to find one given a desired arrival or departure time.

Definition 4.4.3. Another property of jobs is that they can be either inbound or outbound. An inbound request is a job where a customer does not care about the pickup time, but has a high priority on the drop-off time. This is evidenced by setting the time window of l_s to the entire timespan of the

algorithm and setting a normal interval for l_t that indicates between which two times the customer wants to arrive. The idea behind this method is that the pickup can occur at any time, as long as the drop-off is on time. Similarly, a outbound request is just the opposite.

4.5 Events

In our model we account for the following events that may occur during the execution of the system. An event is something that may drastically change an earlier computed schedule and/or will lead to the schedule being infeasible. Thus from a technical point of view, a new job is an event as well, but in this section we primarily focus on the more unexpected situations. The events we describe are based on the ones described by Xiang et al.[20]

Definition 4.5.1. When a taxi is stuck in a traffic jam or otherwise realizes that it not make the scheduled arrival time at some location, it has suffered from a *delay event*. This means that the travel time needed between two locations is going to be more than the earlier computed time and that the taxi is somewhere between these two locations. We denote this as $(l_s, l_t, t_d, t_s, t_r)$ where l_s and l_t are the source and target locations, t_d is the extra travel time and t_s and t_r is the time window during which the delay is active. Any vehicle that is within the time window $[t_s, t_r]$ and en route from l_s to l_t suffer the delay.

It is important to differentiate between a small delay and a real traffic jam, as arriving a few minutes later may be normal whereas a delay of 30 minutes may be considered a traffic jam. However, both events can be modeled using a traffic jam event as delay is basically a traffic jam with a small value for t_d. For the algorithm however, a delay of 30 minutes may lead to many changes to the schedule whereas a few minutes delay isn't going to much of a difference. Note that the value for t_d may be negative, indicating that the taxi is ahead of schedule. If this value is large enough, the algorithm may decide to change the schedule.

An even more extreme situation may occur where taxi may not be able to service the customers in the foreseeable future. The major difference from the two events that were defined earlier is that in this case existing jobs are altered too.

Definition 4.5.2. When a taxi cannot continue with its service we call this a *breakdown*. When this occurs the location of the taxi becomes a new pickup location for all the passengers and other taxis are scheduled to pick them up. All future jobs of the vehicle are cancelled and the jobs corresponding to all the passengers on board are modified by changing the pickup location to the location of the broken down vehicle and reinserted into the algorithm.

Note that we do not consider the repairing/salvaging of the broken down vehicle and picking up of the driver. Also, for this event we assume that the schedule, its workload and the distance it has already travelled does not affect the time at which a vehicle fails.

Another event we describe is the *cancellation event*, in this case either the customer cancels the or does not show up at the designated pickup location. This event can occur at any time given that the corresponding job was already submitted.

Definition 4.5.3. A *cancellation event* may occur for any submitted job. If this event occurs before a taxi is en route to the customer the pickup and drop off locations are removed from its schedule. Otherwise, the taxi arrives at the pickup location, finds that there is nobody to pickup and will continue with the rest of its schedule, except that it will skip the drop off location.

Note that after handling the cancellation event, an algorithm may decide to optimize or recalculate the schedule again, the action chosen depends on how much time is freed by cancelling the job. As we we have now covered each of the elements we now proceed with the definition of our fitness/objective function.

Chapter 5

Evaluating quality

As mentioned earlier, the goal of the algorithm is to produce a schedule for each of the vehicles. Since the DARPTW is a optimization problem, we would like to have a schedule that optimizes our objectives. Even though we want to maximize the quality for every customer, it makes more sense to express quality as the deviation from the perfect ride (exactly on time, shortest route, etc.). Thus our problem changes to minimizing the lack of quality, which we now refer to as the disutility.

Since disutility is composed of several factors, we need to determine which elements of the service are considered important. Swinkels et al.[10] compiled lists of service elements that are considered most important to customers in the Netherlands for different sorts of taxi services. We combined these lists and selected the most important factors which are as follows:

- The taxi departing from the pickup location at the scheduled time (f_{jm_1}).
- Arriving at the destination on time (f_{jm_2}).
- The amount of time the customer has to wait for the taxi (f_{jm_3}).
- Driving time or the time spent in the taxi (f_{jm_4}).
- Not having to share the taxi with other customers (f_{jm_5}).

The work by Jørgensen et al. uses an objective function that weighs a number of elements very similar to the ones we use, these are listed below:

- Customer transportation time (f_{jorg_1}), corresponds to f_{jm_4}.
- Excess ride time (f_{jorg_2}), no mention is made by Swinkels et al. about ride times relative to direct ride times.
- Customer waiting time (f_{jorg_3}), can be expressed using f_{jm_1}.
- Work time of the vehicles (f_{jorg_4}), mostly corresponds with f_{jm_4}, but since we do not consider the working time of vehicles themselves it does not completely correspond to anything.
- Time window violations (f_{jorg_5}), a combination of f_{jm_1} and f_{jm_2}.
- Excess maximum ride time for the customers (f_{jorg_6}), not mentioned.

- Excess work time of the vehicles (f_{jorg_7}), not mentioned.

We see that f_{jm_5} is not accounted for in the model by Jørgensen et al. but as it is one of the least important factors according to Swinkels et al. this does not severely affect our empirical results.

Since we are dealing with multiple objectives we also need to choose a method to compare two solutions to each other where one is better in one objective and the other solution is better in another one. One of the possible ways to solve this problem is to use a weighted sum of all subobjectives. The disutility value of a solution is often composed of several factors such as the travel time, waiting time, the route that was taken and possibly many more.

Definition 5.0.4. The function $d_i(j, R)$ computes the disutility value for objective i for a certain job j wthin a schedule R. The function returns 0 if the schedule provides the optimal quality (no disutility) for that particular element of service quality. In other cases it returns some value $v \in \mathbb{R}$. The output of the function is only defined if j is serviced by the schedule, so there exists some $(t_k, l_k, j) \in R$ for some value of k.

For example, a function for a disutility factor "timely arrival of the taxi" could return how much later a taxi arrived at a pickup location in terms of time. The value 0 would be the optimal value as it means it was right on time, any other value means the taxi was either late or early.

When we look at the disutility from a higher perspective, we can see that a problem instance contains many jobs. Since handling each disutility factor separately is unwieldy when we are comparing solutions, we also need some method to aggregate all disutilities to find the disutility of the solution as a whole. If we would not do so, we need to resort to using Pareto optimality as we otherwise cannot make any good comparison between two solutions.

In this situation an evaluation function will produce a vector of values that denote the evaluation of each specific attribute in the objective function. To compare two solutions the values in the vectors are compared to each other and in this way one can determine which of two is to better. A solution for which no other better solution can be found is then called Pareto optimal.

Definition 5.0.5. A solution x is *Pareto optimal* iff there is no other solution y such that $F(y) = (q_1(y), \ldots q_k(y))$ dominates $F(x)$, where $F(x)$ and $F(y)$ denote the evaluation function solutions x and y respectively. Note that $q_k(y)$ represents a single dimension (subgoal) in the whole evaluation function and that x dominates y if $q_{1\ldots k}(x) \leq q_{1\ldots k}(y)$ and $q_i(x) < q_i(y)$ for some i (minimization).

An algorithm using this theory may produce a set of Pareto optimal solutions that are not dominated by any other solution. This set of Pareto optimal solutions can be called the Pareto frontier or Pareto set. Coello et al.[6] claim that although it may not directly point the decision maker to a single solution, it can be used to gain insight in how improving one dimension of the objective function will lead to degradation of one or more other dimensions. Thus, the bottomline is that a set of good solutions is produced, but it is still up to the operator to decide on which solution to use. This decision is necessary because the Pareto set actually contains "acceptable" solutions and the decision maker should choose the ultimate solution based on the non-modelled (human) preferences. The disadvantage of this approach is that human interaction is still needed during computation. However, this interaction should also result in a solution that conforms more to the requirements the decision maker had in mind. If we do not use Pareto optimality, a function to compute the quality of a solution looks as follows:

Definition 5.0.6. The disutility of a solution s for the DDARP is described by the function $q(s)$. This function produces a disutility value that represents the disutility of s.

This function obviously should use the $d_i(j, R)$ functions to find how well each of the subobjectives is met and use a way to combine the results into some aggregated value (weighted sum). Note that we do not take into account a cost constraint, meaning that we cannot say anything about the cost of the final solution. On the other hand, this means our algorithm can be used to give an indication of the cost needed for a good solution, which is also very valuable information. Furthermore, we can see that it is also easy to add cost as one of the subobjectives.

To be able to make a valid comparison we need to use the same objective function as Jørgensen et al. as well as the same values for the weights. Before we can describe the objective function we need to specify what the input of the objective function consists of. Note that during the computation of the objective value many variables found in the problem instance itself are also needed, so the actual input of the objective function consists of the problem itself and the solution.

The problem information p consists of the information already present before a schedule was computed and includes the state of the road network, travel times for each of the stops, maximum ride times etc. The complete list of all problem related variables can be found in Table 5.1.

Table 5.1: **Problem related variables** - The following variables are problem specific

Variable	Description
N	The set of all jobs
$(x, y) \in N$	Job with pickup stop x and drop-off stop y
$x \in N$	Any stop x from N
$t(x, y)$	Traveltime needed to go from x to y
$t_a(x), t_b(x)$	Time window for stop x
$sv(x)$	The service time at stop x
m	Maximum ride time
w	Maximum work time for a vehicle

The new information provided by the solution s consists of the vehicles routes and other variables such as the waiting time at each stop and scheduled arrival times. A description of all variables can be found in Table 5.2.

Table 5.2: **Solution related variables** - The following variables are solution specific

Variable	Description
R	Set of all vehicle routes
$r_i \in R$	The schedule r_i of vehicle i in solution R
$e_{x,y}$	Equals 1 iff a vehicle travels from x to y in its schedule
$t_{arr}(x)$	Specifies at what time a taxi arrives at x
$lb(v, s)$	Denotes the load of vehicle v **before** it has serviced stop s
$w(s)$	The waiting time at stop s
$t_{route}(v)$	Time between vehicle v leaving the depot and arriving at the depot

Note that when a vehicle arrives at a stop early, it will not immediately commence service, instead choosing to wait until the time window begins. The advantage is that time windows will not be broken as much, but it results in extra waiting time for the customers already on the vehicle. The entire

objective function is the weighted sum of each of the subobjectives $f_{jorg_1}, \ldots, f_{jorg_7}$ as defined by Jørgensen et al.[14] and is as follows:

$$
\begin{aligned}
q(p,s) &= w_1 \sum_{x \in N} \sum_{y \in N} e_{x,y} \times t(x,y) && (5.1)\\
&+ w_2 \sum_{(x,y) \in N} t_{arr}(y) - t_{arr}(x) - sv(x) - t(x,y) && (5.2)\\
&+ w_3 \sum_{r_i \in R} \sum_{x \in r_i} lb(i,x) \times w(x) && (5.3)\\
&+ w_4 \sum_{r_i \in R} t_{route}(i) && (5.4)\\
&+ w_5 \sum_{x \in N} max(0, t_a(x) - t_{arr}(x), t_{arr}(x) - t_b(x)) && (5.5)\\
&+ w_6 \sum_{(x,y) \in N} max(0, t_{arr}(y) - t_{arr}(x) - m) && (5.6)\\
&+ w_7 \sum_{r_i \in R} max(0, t_{route}(i) - w) && (5.7)
\end{aligned}
$$

The resulting function $q(p,s)$ can now be used to evaluate each solution generated and report its disutility. As far as actual implementation is concerned, it is advisable to make sure that the function is as fast as possible, as it is called fairly often and is not very fast as it is $O(n \times m)$ with n being the number of jobs and m being the number of vehicles. In the next part we go into more detail about the technicalities of the implementation of our genetic algorithm itself and present our crossover operator.

Part III

Genetic algorithm

Chapter 6

Overview of the GA

In this part we describe the Random Insertion Genetic Algorithm (RIGA) that solves the aforementioned dial-a-ride problem. The general idea of a genetic algorithm is that given some initial population of solutions, the algorithm continuously selects the best solutions and evolves them using crossover and/or mutation operators to generate a new generation of solutions. It is here where the random insertions are performed as we randomly insert jobs into other schedules during crossover. This process is repeated until some termination criteria is met, after that the best solutions that were encountered are returned. This idea of remembering the best solutions that were ever seen is called elitism and is frequently used for genetic algorithms, examples include the algorithm by Jørgensen et al.[14] and the generic genetic algorithm by Deb et al.[8]. The general idea of a genetic algorithm (including elitism) is shown in Algorithm 6.1 on the following page. As we can see from the pseudocode, our genetic algorithm is composed of the following steps:

- **Initialization** step where a set of initial (random) solutions is created. We use a cluster-first-route-second approach because it is a commonly used method (see also [14]) and because clustering is considered to be a more important (but not independent) decision than scheduling[1, 13]. We first create a random cluster and then create a random (but feasible) schedule for each cluster, after that we may or may not apply the local search procedure according to the specified parameter p_{ls}.
- A **selection** procedure that selects the best candidate solutions to produce the next generation. This basically means that the size of the population shrinks by removing the worst elements.
- The remaining elements are then used by the **repopulation** procedure to create new solutions to add to the population until the specified population size is met. The basic idea is that the existing solutions are subjected to crossover and local search operators to try to generate offspring solutions that are better.
- The **termination** condition determines when the algorithm should stop and return its result. Possible criteria may include time limits, reaching some quality value, or convergence.

Algorithm 6.1 Overview of a genetic algorithm

```
1: population = generate_initial_population()
2: elitist_archive = ∅
3: while not termination_criteria_met() do
4:    subset = select_best_subset(population)
5:    elitist_archive = update_elitist_archive(subset)
6:    population = generate_next_population(population)
7: end while
8: return elitist_archive
```

In the following chapters we describe each of the earlier mentioned components in more detail, discuss how we implemented them and which alternatives are available. However, before we can think of how to initialize our first population we need to decide how to represent a solution in the GA.

Chapter 7

Solution representation

A common method used for genetic algorithms is to represent solutions as bitstrings, where a portion of the bitstring denotes how the jobs are clustered and another portion that represents each of the schedules. Based on this idea we could use a Estimation of Distribution Algorithm (EDA) that estimates the distribution of each of the bits and look for the best solution. However, there are two major issues with this method:

1. The most important reason why using a bitstring notation is not appropriate is because there are a lot of constraints on many of the bits. For example if we let bit b_{xy} denote that job x is to be clustered to taxi y, the variables b_{xq} for $q \in [1, m]$ must be 0 if $b_{xy} = 1$, since a job can only be assigned to one taxi. If we remember the precedence and capacity constraints we mentioned in earlier sections it is easy to see that many bitstrings within our search space will be infeasible. To circumvent this issue we can either allow infeasibility (by using some metric to grade solutions based on feasibility or check every solution for feasibility before adding it to the population. It is easy to see that a feasibility metric will not be very helpful as the number of infeasible solutions in our search space greatly outnumbers the feasible portion and there is no way we could use an infeasible solution (for example, capacity constraints are hard). On the other hand, checking the feasibility for every single solution results in a very slow algorithm as all constraints have to be checked many times.

2. Since a solution is represented by both the clustering and the scheduling, it is not trivial to estimate the distribution of bits either. When looking back at the previous example, a distribution for bit b_{xy} only makes sense if all other bits b_{xq} for $q \in [1, m]$ are not set and vice versa. This by itself is not a problem, as there exist algorithms that can handle multivariate interactions such as the Extended Compact Genetic Algorithm (ECGA). However, note that we have not yet considered the scheduling part of the problem, which essentially means that on top of the multivariate factorization we can find for the clustering part another multivariate factorization is needed to handle the scheduling.

We do have to stress that using a bitstring representation of the clustering itself is perfectly possible, as there have been a number of algorithms that encode only the clusters as bits[14, 19]. However, if we are to solve the whole DARP in a GA, we can see that bit strings notations and estimation of distribution algorithms are not a good choice for representing and solving our problem. Alternatively, instead of attempting to estimate the distribution of the various bits we can see that if we can define

a crossover operator that works on solutions as defined in 4.2 on page 11 we do not need to concern ourselves with the distribution of various bits and solutions are much simpler to operate on (especially considering the constraints). In the next chapter we show how generating a solution becomes much easier if we operate on solutions directly.

Chapter 8

Initialization

Most genetic algorithms start by acquiring a initial set of solutions, which are often generated in a random fashion. Considering that we use a cluster-first-route-second approach to solve the problem, using the same approach to generate a random solution seems a fair enough as well. Note that by clustering we mean that jobs are divided into groups (and assigned to a taxi) and routing refers to determining the exact driving schedule of a taxi. Since we are using a random way to generate solutions the general idea of generating a random solution from scratch is as follows:

1. **Clustering**: Assign each job to a taxi, only allowing the assignment if the capacity of the empty taxi is larger or equal to the capacity requirement of the job.
2. **Routing (part one):** For each taxi, repeatedly remove a random job from its list of assigned jobs and append it to its schedule as either a pickup or drop off stop. A stop can only be appended if the capacity and precedence constraints both hold.
3. **Routing (part two):** Once a complete schedule for a taxi is created, a simple routine is used to calculate the arrival times for each of the stops. The most difficult part is determining when a vehicle should leave the depot as it depends on whether a request is inbound or outbound. Even though any pickup time is fine for a inbound job, we obviously would want to perform the pickup before the time window for delivery begins. On the other hand, for the outbound requests we can just make sure to arrive within the time window for the pickup.

The pseudocode of each part of the algorithm can be found in the appendix on page 66, but for the sake of clarity we have omitted a few elements:

- In the pseudocode we assume that a job is guaranteed to be clustered even though a job could theoretically require more capacity than any of the vehicles can provide. In our implementation we added code that checks this property, but as it is not critical to normal operation we choose to omit this code for the sake of clarity.
- Determining the waiting time is simply done by checking whether a vehicle arrives before the beginning of the time window, it is not explicitly shown how this is computed, but it is not very important.
- When deciding when to leave the depot to pickup someone for a outbound request we choose to do so as late as possible, meaning that we calculate the pickup time as if the customer would

be immediately driven to his destination after pickup. This means that if this isn't the case (some other request is serviced after pickup), the customer may arrive too late and/or incur some waiting time because the vehicle left the depot too late.

Furthermore, since we do not need the actual schedule times we do not have to compute them directly after generating them. Schedule times are only needed when computing the quality of the solution, meaning that it is only needed during the selection stage and the local search procedure.

Another important feature of our initialization step is that we maintain a sorting in the population, this is greatly beneficial for the performance as this means we do not need to keep sorting the population again after repopulation. The benefit of doing so is that our selection step is quite simple now.

8.1 Selection

Since we maintain a sorted population the best elements are those at the top. After a population has been created, the selection procedure selects one element from the worst few elements of the population and replaces it by a crossing over two randomly selected parents from the remaining population. The parameter that indicates when a element is bad is based on a ratio that specifies how many elements risk being replaced during each iteration. As long as this ratio does not cover all of the elements, the selection procedure maintains an elitist archive containing the best solutions, as the best solutions will never be replaced. So after selection has occurred the size of the population has decreased by one element, we choose not to replace all elements because repopulation is relatively slow. This procedure is similar if not identical to the one used by Jørgensen et al.[14], which also means that it becomes slightly easier to compare our algorithm to their results as we now also do a similar amount of work during selection.

Chapter 9

Repopulation

After purging some element from the population the algorithm enters the repopulation phase where the population grows back to its defined limit. This part of the algorithm is based on the work of Pereira et al.[19], Jørgensen et al.[14] and Xiang et al.[20]. More specifically, the idea of our crossover operator is based on those of Pereira's and Jørgensen's with some modifications. Our local search (mutation) operator is based on Xiang's ideas of best insertion. In the following sections we elaborate on the particulars of our implementation.

9.1 Crossover

The idea behind our crossover operator is actually quite simple: Our crossover operator is a two-point crossover operation that selects a schedule of one taxi (cluster) from the first parent and all other clusters from the second parent, which are combined into a single child solution, the greatest difference being that two entire solutions are combined and not just the clusters. The pseudocode for the procedure is presented in Algorithm 18.3 on page 67.

Since clustering is considered to be more critical than routing[3, 1], it is important to have more diversity in how clustering is performed. As such we decided to make some changes to the crossover operator that was used by Jørgensen et al. by simplifying the selection procedure for the parents and omitting the usage of a binary template. The reason we chose to do so is because we realized that our local search procedure already takes care of improving solutions by itself, but to do so it would need a more diverse population to be able to find meaningful results. By simpliyfing the crossover the algorithm is left with more time which can be used to call the local search routine more often while creating somewhat a somewhat more diverse population. The idea of moving around clusters is actually reminiscent of the approach of Pereira et al.[19] but has the following differences:

- Pereira's approach is presented for the vehicle routing problem (VRP), and as such does not need to deal with carrying payloads of different customers at the same time. This means that selection of the cluster to be transferred is much easier, as it does not need to consider itself as much with precedence and capacity constraints as we need to do. The problem of dealing with the precedence constraints is also acknowledged by Jørgensen et al.[14]
- Our approach copies the complete schedule of a vehicle, whereas Pereira's method copies only a subset of it. This obviously limits the number of possibilities left for selecting the elements to copy, but since we need to consider more constraints than in the VRP this seems like a good

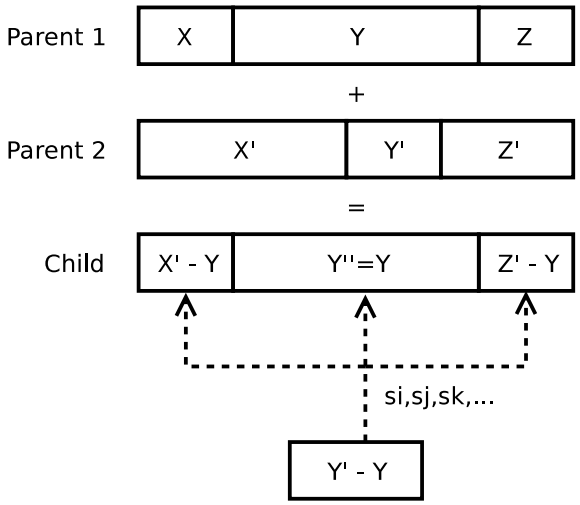
compromise to us. Also, since the jobs that were in the replaced cluster need to be inserted in the solution again, the child solution will be different from its parents besides that one cluster that was copied.

- We select both parents at random, whereas Jørgensen et al. use the roulette wheel method to select one parent based on its quality (see Hypothesis 9.1.1). We choose not to do this as this adds unnecessary computational complexity and we do not want to be too obsessed with solution quality during the crossover phase as the local search procedure will already do so. Furthermore, by selecting parents based on their solution quality one would limit the search space, which is not something we want to do since we cannot say anything about the quality of the child. We do not test this hypothesis as its verification is not important for the overall results.

  Hypothesis 9.1.1. There is no significant difference between selecting parents for crossover randomly and selecting them stochastically.

- Finally, Pereira et al. use the cluster they selected as a bitmask of sorts on the other parent's corresponding cluster. We forgo this approach and copy the whole cluster directly, which means that at least one taxi schedule from the child will be completely identifical to one of one of the parents.

Figure 9.1: **Schematic of the crossover operator** - Note that we first needs to remove all stops in Y from X', Y', Z' before we can randomly insert stops $si, sj, sk, \ldots \in Y' - Y$



A graphical example of how our crossover operation creates a new child solution can be seen in Figure 9.1. As far as computing routes is concerned, the genetic algorithm by Jørgensen et al. only really focuses on the clustering part and uses the space-time-nearest-neighbour by Baugh et al.[3] to solve the routing. Our algorithm is different here because it lets the genetic algorithm work on both the clustering and the routing, but strictly handles the clustering before doing any routing when generating candidate solutions, hence it qualifies as a cluster-first-route-second algorithm.

While Jørgensen et al. also copy a cluster completely, they use it like Pereira et al. do, so the selected cluster only serves as a bitmask. To be more specific, they select only the cluster while we select the driving schedule $\{stop_1, stop_2, \ldots, stop_x\}$ of a taxi t and copy it completely.

It can be easily seen that if we want to copy a schedule into another solution there are three things that must be taken care of:

1. Most importantly, any job that was in the copied schedule must be removed from the other schedules in the target solution.

2. All jobs that were in the replaced schedule and not in the copied schedule must be inserted into the target solution again while taking care not to invalidate precedence and capacity constraints

3. All schedules that were modified must be checked again on their arrival times, as adding and removing stops in schedules will change them

The first problem of removing jobs that are already present is also rather trivial (ignore the time windows other timing issues for now), since it only involves removing stops. It is easy to see that no capacity or precedence constraint can be ever invalidated by removing a job from an existing schedule because vehicle capacity only changes for the better and precedence constraints specify the relation between two stops of the same job. However, the waiting time increases as there are now gaps in the schedule.

The last problem of correcting arrival times can be easily solved by iterating through the schedule again and updating the arrival times, which is not very difficult.

The second problem is the most difficult as we must insert customers into existing schedules, which means we need to keep in mind the vehicle capacity at each stop asnd the precedence constraints.

There is a trivial method to solve this insertion problem, which is simply to append the jobs back into some randomly selected schedule of a vehicle that has enough capacity when it is empty. However, it is also easy to see that this simple method is almost always guaranteed to produce a very bad ride for the unlucky few customers who had to be inserted again.

Instead we randomly select a vehicle to assign the job to and randomly insert the job into its schedule. Denote a stop within a schedule as s_i where i is the index of the stop within the schedule. We can now find a feasible insertion as follows:

- Find some stop s_i where the vehicle can visit the pickup stop without invalidating any capacity constraint if inserted after s_i
- Select the first pickup stop s_j with $j > i$ where the capacity constraint does not hold (negative capacity left) if we would insert the pickup after s_i
- Randomly select one stop s_k with $i < k \leq j$
- The pickup stop can now be inserted after s_i and the drop-off can be inserted before s_k

Since we assume that any result in the population is a feasible one, it is easy to see that no precedence constraints can be broken, as the drop-off always occurs after the pickup stop ($i < k$). To prove that it does not break any capacity constraints we can do the following:

Proof. (by contradiction) Assume the capacity constraints of a feasible schedule are broken after insertion

Capacity constraints are broken if one or more of the following occurs:

(1) *The vehicle attempts to visit the pickup stop while it is already full* - Not possible, s_i is selected such that no capacity constraints are broken.

(2) *The vehicle visits the pickup stop and finds that it cannot visit some future pickup stop anymore due to lack of capacity* - Not possible, all stops are inserted before s_j, which is the first invalidated stop

(3) *The vehicle attempts to drop a customer off but finds that it has (more than) full capacity left (nobody on board)* - Not possible, we assumed a feasible schedule, we know $i < k$ and picking up a customer decreases the capacity only

We conclude that no capacity constraints are violated. □

After insertion of the missing jobs the crossover process is done and the result will be a feasible child solution. In our algorithm we replace the schedule of a vehicle by its corresponding schedule in the other solution, so we do not need to check the capacity requirements as the vehicles are the same. We could choose generalize this by allowing copying from other vehicle schedules, but that would (in general) require us to check whether the copied schedule for feasibility again which does not seem very beneficial to us.

Depending on the parameter for the local search probability, the algorithm may decide to perform a mutation/local search on the resulting solution. This procedure is discussed in the next section.

9.2 Mutation/Local search

From the earlier subsection we can see that routing is given rather little attention during the crossover operation, our GA approach compensates for the lack of a more dedicated routing solution by applying more mutations than the earlier mentioned authors. We used the *remove-one-insert-one* heuristic by Xiang et al.[20], which is similar to the swap strategy by Baugh et al.[3]. The general idea of those strategies is that a random job is selected and moved to another cluster. After it has been moved, the job is inserted in the best way possible using a method similar to the insertion procedure in the previous subsection. However, now we need to evaluate all values for s_i, s_j and s_k, meaning that it takes $O(m^2)$ time for the m stops in the schedule. As stated by Xiang et al., the remove-one-insert-one procedure is $O(n)$ by itself (with n being the number of jobs) meaning that by itself it is reasonably fast. However, care should to taken as insertion takes quite a bit of time.

Our implementation of this *best_insertion*() function is slightly different from the one idea by Xiang et al. as well. Whereas they force the procedure to find a better solution than their starting one, or otherwise optimally insert the job back into its original schedule, we actually let our algorithm find the best insertion itself, so we accept solutions where the quality decreases due to moving the customer. Due to this characteristic this procedure could also be called a mutation operator. There are two reasons why we chose this alternate approach:

- As our crossover operator already modifies the clusters, but does not do anything special regarding the routes. We think local search in this context should be about trying very small changes to see if they are interesting. If they are sufficiently interesting they will not be removed by selection procedure and as such may be able to pass this change on to further generations. The procedure as described by Xiang et al. obviously sounds more sensible, but is also greedy as it rejects anything that shows no direct improvement.
- While we cannot prevent our local search from degrading the original solution due to a unfavorable insertion, we know that our selection procedure will remove the solution eventually if becomes too bad. Also, since we only move one job around, the effect on the solution quality will not be very significant (regardless of whether it is a good or bad effect) as we use the best (or actually, the least worst) insertion.

A more detailed description of the best insertion procedure can be found in the appendix at algorithm 18.4 on page 68. Once the remove-one-insert-one has finished the resulting solution is returned and will be added to the population.

As it is too costly to do local search for every new child solution, we give each crossover operation a small chance to also perform local search. This is also one of the reasons why we think that using the greedy approach is not very appropriate. Even if a better solution is found, the improvement will not be too great as it only moves one customer to another cluster (except for early iterations where solutions are very bad). So instead of trying to find slightly better solutions it may be much more useful to provide the crossover with a more diverse population so that more diverse (and hopefully better) solutions can be found. If we take into consideration that the (greedy) best insertion is slower we suspect that our method should work better in our algorithm:

Hypothesis 9.2.1. The modified best insertion procedure is more suitable for the RIGA than the implementation described by Xiang et al.[20] in the sense that the RIGA produces better solutions

Although it is important to verify whether this hypothesis holds, it is beyond the scope of this project and should be done in future work.

9.3 Termination and return values

Ultimately, the algorithm has to decide when it should stop and return its results. Because of our test environment, we choose to implement a (configurable) fixed time limit for our algorithm. Another alternative is to let the algorithm terminate after a fixed number of iterations, which is done by Jørgensen et al.[14]. As for the return value, since we keep the population sorted, we can immediately return the first element as that is also the best solution.

As it is now clear how each of the major components in our algorithm works, we can now begin with experiments that have to show how our solver performs.

Part IV

Empirical results

Chapter 10

Experimental setup

In this part we discuss the empirical results we obtained after implementing and testing RIGA we presented earlier. The most important difference between our solver and other cluster-first route-second solvers is that our solver uses a GA for both the cluster and the routing phase, whereas other approaches only use a GA to handle the clustering. We compare our algorithm to the implementation by Jørgensen et al.[14] and conclude with a simple comparison to some of the results presented by Parragh et al.[17]

The goal of our experiments is to find out whether specialized routing heuristics (like the one by Baugh et al.[3]) are really necessary when using GAs to solve the static DARP or if GAs by themselves are powerful enough to find good routes. We believe that this is indeed the case and have formulated this as follows:

Hypothesis 10.0.1. Given similar or equal time and computational limits, the RIGA approach produces results that are competitive to the ones found by traditional route-first cluster-second genetic algorithms using specialized routing heuristics.

To be able to evaluate whether this hypothesis holds we need to determine whether our solver can compete with other solvers using comparable time and computational limits. As the experiments by Jørgensen et al. were done quite a while ago, we need to normalize our results accordingly. The basic idea here is to make sure that if we run both solvers on the same machine for the same amount of time, our results should be competitive to theirs. This would mean that our solver could function as a drop-in replacement.

Since the computational power of computers has increased over the years, it is possible to do the same amount of work in less time. On the other hand, if we choose to fix the amount of time available, we can do much more work than previously possible, which should result in better solutions. Consequently, we expect that both GAs will perform better on faster hardware and a fixed amount of time.

Hypothesis 10.0.2. Given a constant time limit, both GA-methods will find better solutions on faster hardware.

Given the nature of our algorithm we suspect that if we have sufficient computational power, using the full-GA method may be preferable to using routing heuristics. The main reason for this is that heuristics inherently make assumptions regarding the objective function, such as choosing the closest stop or a stop with the earliest time window. This means that heuristics do not completely follow

objective functions as the primary method to choose the next stop. On the other hand, we use a method that relies much more on chance and occasionally uses the objective function to determine which stop to select next, which means our algorithm pays more attention to the given objectives which should result in solutions of higher quality.

As our algorithm starts with worse solutions than Jørgensen et al., we can only expect to outperform their algorithm if we are given enough time to overcome this disadvantage of our algorithm.

Hypothesis 10.0.3. Given sufficient time/computational power, the RIGA will outperform genetic algorithms using routing heuristics.

10.1 Setting up the experiments

For our experiments we were allowed access to the TU Delft cluster of the Distributed ASCI Supercomputer 4 (DAS-4) which consists of 32 nodes with each node having a dual quad-core processor (Intel E5620) clocked at 2.4 GHz. Even though our algorithm does not benefit from multiple cores, it does enable us to execute benchmarks in parallel.

Before we go into further detail about the experiments, it is important to note that the implementation by Jørgensen et al. was written in Java, while ours is written in Python. As such, we need to realize that there may be differences between performance due to the different platforms. However, for our work we assume that the difference in performance due to different programming languages is negligible and that there is no significant difference between the technical quality of the implementations. The source code of Java algorithm was taken from the thesis by Bergvinsdottir[4] and modified slightly to allow us to measure convergence as well as modifying the termination criterion to support time limits. As these changes are very small (a few lines of code) and only activated when needed we can assume that they do not negatively affect the performance of their solver.

10.1.1 Problem instances

The 20 problem instances that we use are created by Cordeau et al.[7] using realistic assumptions regarding time window widths, vehicle capacity, route duration and maximum ride times. These instances contain between 24 and 144 jobs. Half of this set of problems uses problems with narrow time windows whereas the other half has wide windows. Furthermore, Cordeau et al. also varied between the job to vehicle ratio, meaning that our solver is tested for its performance on a number of different sorts of problems. Finally, because there are a number of authors who have also used these problems, it makes it somewhat easier for us to compare our results[4, 14, 17].

10.1.2 Algorithm parameters

Before we can commence testing we first need to decide on how to configure both algorithms. Since we are mostly interested in whether the full-GA approach is useful or not, as a result, our initial parameters are nearly the same as those by Jørgensen et al. and are shown displayed in table 10.1 on the next page.

For our algorithm, there are only three actual parameters to be set (excluding the termination criterion, as this depends on the experiment), which are as follows:

- **Population size (p):** This is a rather obvious parameter, for our tests we set it to 50, as this was also done by Jørgensen et al. and we had little reason to modify it for our initial experiments.

We initially used values of over 100 but we found that this resulted in too much slowdown and no improvement due to the algorithm not being able to perform enough iterations given some amount of time.

- **Local search probability (p_{ls}):** The probability of executing the *best_insertion*() function for a newly created child, in our first tests we used $p_{ls} = 0.01$ (or 1%), but after some simple experiments we found $p_{ls} = 0.08$ to be acceptable (refer to Chapter 14 for a more detailed discussion of this parameter). While increasing this value should produce better results for the same amount of iterations, the runtime complexity of $O(m^2)$ for the size of the taxi's schedule m greatly increases the amount of time needed per iteration. So in general we should try to a value such that we can still reach convergence within the given time.
- **Replacement percentile (z):** This value denotes the percentile of solutions that risk being replaced in the current iteration, a value of 0.1 means that the worst 10% of solutions risk being replaced. Note that this does not mean they will all get replaced, as only one solution is replaced every iteration. We kept this value low to make sure only the really bad solutions are rejected while preserving solutions with potential.

Table 10.1: **Algorithm parameters** - The initial settings for both algorithms. Note that the ones for our algorithm are mostly based on those used by Jørgensen et al.

Parameter	Jørg.	JM
Population size	50	50
# Iterations	15000	-[a]
Mutation prob.	0.01	0.08[b]
Replacement percentile	0.10	0.10

[a] Depends on the experiment as it is a termination criterion
[b] Probability of executing *best_insertion*()

For our comparison to make any sense it is important that our objective values are measuring the same units. This means that we have to use the same objective function as Jørgensen et al. The weights for the objective function discussed in Chapter 5 can be found below.

Table 10.2: **Objective value weights** - We use the same weights as Jørgensen et al. for our experiments. Here n denotes the number of jobs, which means that the last three objectives are proportionally weighted.

	Value	Description
w_1	8	Customer transportation time
w_2	3	Excess ride time
w_3	1	Customer waiting time
w_4	1	Work time
w_5	n	Time window violation
w_6	n	Maximum ride time
w_7	n	Maximum work time
m	90	Maximum travel time limit per job (1.5 hours)
w	480	Maximum work time per taxi (8 hours)

In the following sections we first test whether Hypothesis 10.0.1 on page 34 holds. This requires us to run our algorithm using time and computational limits comparable to the ones used by Jørgensen et al. and analyze the fitness values of our solutions.

Chapter 11

Establishing a performance baseline

As stated before, we want to know how our algorithm would perform in the environment used by Jørgensen et al. to ensure a fair comparison is made. However, since it is not feasible (within the scope of the project) to replicate the exact environment used by those authors, we found a compromise by assuming the following conditions:

- Assuming the performance and efficiency of the Java Virtual Machine (JVM) has only increased since previous versions, the results we get from running the reference algorithm will not be significantly worse than those reported.
- Any difference in hardware has no effect on the quality of the solutions, as Jørgensen et al. use a fixed limit of 15000 iterations.
- We assume that the difference in speed between the JVM and the Python interpreter is either insignificant or in the JVM's favor. Which means that our algorithm should be faster if we implemented it again in Java.[1]

Following the conditions stated above, we can see that both the different platforms and newer hardware only influence the running times of the algorithm. Furthermore, since we consider Python to be slower than Java, our test setup is actually biased towards the Java implementation. So if our algorithm would outperform the Java solver, we know that our solver would perform even better if it was implemented in Java.

To establish the performance baseline, we run the Java solver 10 times on each of the benchmark instances using default settings and note the time needed and the value of the objective function. For our own solver we do mostly the same (with parameters from 10.1 on page 36), but we limit the time available for our algorithm to the time needed by the Java code.

Since we only perform 10 runs per benchmark instance, it is important to determine whether any difference in the test results is actually significant or only due to chance. To do so we assume that result of a solver has a normal distribution with unknown μ and σ^2. We would like to know whether the mean fitness of a solution produced by Jørgensen's solver is significantly different than one produced by ours. To find whether this is true we perform a paired t-test with critical value $p_{crit} = 0.05$ and the null-hypothesis $H_{0_i} : \mu_{Jorg_i} = \mu_{JM_i}$ for each benchmark instance i and report the p-values found. The results of this experiment are in Table 11.1 on the following page.

[1] For more about Java vs Python issues, refer to `http://wiki.python.org/moin/LanguageComparisons` (Accessed 14-06-2011)

Table 11.1: **Basic comparison to Jørgensen's solver** - n denotes the number of jobs, m is the number of vehicles. As can be seen, our algorithm seems to have some difficulty to cope with smaller problems ($n = 24$ and once for $n = 48$). But it does make up for this deficiency by finding significantly better results for larger problems. The time listed is the amount of time needed by the Java solver.

				Jørgensen et al.	JM		
Name	n	m	Time (ms)	Objective	Objective	Ratio	p
R1a	24	3	12659	**6122**	6206	1.01	0.85
R2a	48	5	30855	**18285**	19518	1.07	0.51
R3a	72	7	56903	61913	**26329**	0.43	0.00
R4a	96	9	93421	148846	**33739**	0.23	0.00
R5a	120	11	139133	217676	**40228**	0.18	0.00
R6a	144	13	199559	414011	**51766**	0.13	0.00
R7a	36	4	20967	12361	**9641**	0.78	0.01
R8a	72	6	61240	76042	**54143**	0.71	0.00
R9a	108	8	121780	343850	**191923**	0.56	0.00
R10a	144	10	201853	729234	**332969**	0.46	0.00
R1b	24	3	13277	**4388**	4731	1.08	0.19
R2b	48	5	31572	15063	**10512**	0.70	0.01
R3b	72	7	56785	35653	**18455**	0.52	0.00
R4b	96	9	91934	74236	**23000**	0.31	0.00
R5b	120	11	137683	129357	**26829**	0.21	0.00
R6b	144	13	198794	218418	**32464**	0.15	0.00
R7b	36	4	21240	9385	**6993**	0.75	0.01
R8b	72	6	61933	54090	**26519**	0.49	0.00
R9b	108	8	120110	217875	**74304**	0.34	0.00
R10b	144	10	204994	589682	**214800**	0.36	0.00
Overall						0.52	0.00

11.1 Basic performance comparison

The most surprising discovery we make is that our algorithm outperforms the Java solver by a significant margin on most (larger) problems. This is also somewhat confirmed in the corresponding p-values as many of them are very close to 0.00, indicating that it is very improbable to find the results we found if the null-hypothesis holds. As for the objective values themselves, given the great improvement we see for some instances, we believe that this is mostly due to our crossover operator. Since crossover is random, it will move a randomly selected vehicle cluster from one solution to another. However, unlike Jørgensen et al. we do not use a routing heuristic to compute a route by looking at costs of different stops but instead choose to randomly insert the stops while enforcing feasibility. This means that our results after a crossover are more diverse in solution quality. However, to compensate for this we occasionally use our local search procedure to improve on the resulting solution by optimally reinserting a job into some vehicle. This is different from the regular mutation which is mostly a random operation.

Given the p-values for the cases where we perform worse, we can claim that the difference is not significant enough to reject the null-hypothesis of $\mu_{Jorg_i} = \mu_{JM_i}$. On the other hand, for all cases

where we are better the p-values suggest that we can reject the null-hypotheses for those cases and claim $\mu_{JM_i} < \mu_{Jorg_i}$ for the corresponding problem instances.

In any case, we can now argue that Hypothesis 10.0.1 holds partially as we have shown that our algorithm actually produces solutions that are twice as good (on average) given similar bounds. The next step is to check whether Hypothesis 10.0.3 also holds, which requires us to determine our optimal parameters first as we are now allowed to tweak our algorithm as much as possible. Before we can find our parameters we first need to understand where the improvement in our solutions comes from, this analysis is presented in the next chapter.

Chapter 12

In-depth analysis of objective values

As our objective values are actually composed of seven subobjectives, we need to look at our fitness function in more detail if we want to understand where the improved performance comes from.

If we look back at the objective function weights in Table 10.2 on page 37 we see that the most important factor apart from the ones that are proportionally weighted is traveltime, which is the time the customers travel in a taxi and has a weight $w_1 = 8$. Clearly, this is the most important subobjective and small improvements here will benefit the overall objective value greatly. Given that our solver sometimes produces solutions with objective values almost a factor ten smaller leads us to believe that our algorithm somehow manages to find solutions that improve the most on the most important objectives and do worse on the less important ones.

To be able to confirm whether this idea is correct we need to look at the values of each of the subobjectives and see where the greatest difference can be found, we also use the paired t-test here to check the significance of the test results.

12.1 Analyzing quality for each of the subobjectives

If we refer to Table 12.1 on the following page we see that our solver produces solutions with waiting times that are about a hundred times as high as the ones found by Jørgensen et al. However, as the weight corresponding to the waiting time is only 1, the difference does not amount to much in the resulting solution. The three most important subobjectives (excess ride/work times and time window violations) are greatly improved upon, which means that most of the improvement could be found there. Note that while we only performed 10 runs per instance, the p-value of each of the objectives is near zero, indicating that the perceived differences in the average values exist and can be considered statistically significant for $p_{crit} = 0.05$.

Interestingly, both solvers score about the same on transport time and route duration, even though we suggested that the major improvement was to be found in the transportation times. Furthermore, the data also shows that almost all of our improvement comes from the three highest weighted subobjectives (time window violation and excess of max ride/work time). This also shows that our local search procedure is capable of finding good schedules in the long run even though it can theoretically produce a solution that is worse than its input.

If we look at the values of the various subobjectives for our smaller problems (R1a, R2a and R1b) we see that most subobjectives are very competitive to the reference values except for the vehicle waiting time. We suspect that this is due to the way our algorithm begins its computation. As the

Table 12.1: **Breakdown of objective values** - *Transport* denotes the distance a taxi moves. *Ex. Ride* is the comparison between optimal distance and realized distance. *Wait* is the time a vehicle spends waiting at a stop, multiplied by its current load. *Route* is the time a vehicle is active. *TW viol.* stands for the amount of time window violations. *Ex. max ride* and *Ex. work* are the excess of maximum ride time and work time. Our algorithm trades greatly increased waiting time for decreased ride time, time window violations and excess on ride/work times. This exchange does not seem to work for smaller instances.

Weights	8		3		1		1		n		n		n	
Subobjective	Transport		Ex. Ride		Wait		Route		TW viol.		Ex. max ride		Ex. work	
Problem	Jørg.	Ratio	Jørg.	Ratio	Jørg.	Ratio	Jørg.	Ratio	Jørg.	Ratio	Jørg.	Ratio	Jørg.	Ratio
R1a	306.09	**0.97**	511.85	**0.77**	28.68	27.89	1066.58	**0.85**	25.32	1.52	18.10	**0.06**	2.87	**0.00**
R2a	539.53	1.00	1321.39	**0.98**	38.14	81.47	1974.47	1.00	119.20	**0.89**	47.30	**0.51**	13.92	**0.00**
R3a	1036.69	**0.98**	2569.87	**0.63**	81.91	26.24	2680.75	1.04	379.27	**0.29**	217.12	**0.02**	0.00	NaN
R4a	1214.17	**0.99**	3827.92	**0.58**	220.36	16.18	3514.10	1.04	749.78	**0.12**	512.19	**0.03**	27.01	**0.00**
R5a	1365.05	**0.98**	4710.45	**0.57**	293.76	12.89	4129.38	1.04	921.73	**0.11**	632.69	**0.02**	0.00	NaN
R6a	1785.20	1.01	6204.76	**0.55**	309.11	14.50	5200.23	1.03	1465.39	**0.08**	1090.35	**0.01**	0.00	NaN
R7a	473.99	1.00	925.92	**0.76**	9.73	117.16	1322.50	1.04	83.76	**0.39**	40.11	**0.04**	28.81	**0.00**
R8a	910.28	1.00	3044.77	**0.64**	84.47	9.02	2435.26	1.01	426.22	1.13	360.59	**0.12**	52.64	**0.00**
R9a	1307.80	**0.96**	5857.27	**0.52**	93.56	6.62	3514.39	1.00	1788.14	**0.79**	1075.66	**0.14**	6.33	**0.53**
R10a	1748.65	**0.99**	8667.18	**0.52**	262.07	4.43	4776.39	1.00	2327.88	**0.77**	2222.30	**0.10**	201.24	**0.34**
R1b	280.11	**0.99**	387.31	**0.83**	0.52	1326.39	887.88	**0.97**	1.56	**0.00**	2.47	**0.13**	2.87	**0.00**
R2b	522.70	1.05	997.63	**0.92**	17.36	75.50	1597.90	1.07	105.01	**0.06**	25.68	**0.04**	13.92	**0.00**
R3b	1010.91	**0.99**	2147.68	**0.66**	26.03	44.84	2535.84	1.03	104.32	**0.30**	153.47	**0.01**	0.00	NaN
R4b	1202.49	**0.98**	3257.04	**0.61**	58.43	24.02	3228.94	1.02	219.72	**0.09**	313.73	**0.03**	27.01	**0.00**
R5b	1338.59	**0.99**	3972.68	**0.58**	129.56	22.61	4097.96	1.02	412.45	**0.04**	417.75	**0.00**	0.00	NaN
R6b	1770.06	**0.99**	5253.70	**0.51**	151.39	18.30	4902.50	1.03	709.11	**0.02**	553.45	**0.00**	0.00	NaN
R7b	459.05	**0.95**	681.65	**0.80**	3.33	153.30	1221.01	1.01	31.26	**0.09**	36.59	**0.00**	28.81	**0.00**
R8b	932.11	**0.99**	2350.62	**0.72**	45.20	32.08	2521.82	1.02	196.64	**0.66**	314.50	**0.02**	52.64	**0.05**
R9b	1345.52	**0.99**	4714.49	**0.66**	153.99	9.20	3624.45	1.02	724.85	**0.46**	964.05	**0.09**	6.33	5.89
R10b	1729.22	**0.97**	8108.41	**0.53**	151.37	6.02	4702.31	**0.99**	1562.26	**0.64**	2069.47	**0.10**	201.24	**0.35**
Average	1063.91	**0.99**	3475.63	**0.67**	107.95	**101.43**	2996.73	**1.01**	617.69	**0.42**	553.38	**0.07**	33.28	≈ 0.51
p-value	0.00		0.00		0.00		0.00		0.00		0.00		0.00	

first generation consists of randomly generated solutions, it is not strange to see solutions with fitness values that are a factor 100 or 1000 larger than the final ones as all subobjectives have very bad values. However, since the *best_insertion*() routine only cares about reducing the objective value, it does not care where the improvement comes from, this basically means that it is easier to find solutions with lower time window violations and lower excesses of ride/work times.

Note that the routing heuristic by Baugh et al.[3] implicitly assumes several conditions when selecting stops. For example, sorting the stops based on space-time proximity implies that the algorithm tries to keep the traveltimes between stops low. Even though this is a good guideline for the selection process we can imagine that this limits the number of other options we have. Furthermore, focusing so much on the space-time proximity essentially implies a high preference for keeping the transport and waiting times low regardless of the objective function, which is only partly so for Jørgensen et al. as waiting time has a weight of 1.

Since we only call the local search procedure at most once for some selected population member, it does not come as a surprise to us that our algorithm is relatively slow in finding good solutions as it starts with a population of randomly generated solutions. Also, the weights also affect the rate at which each subobjective improves, so for the next experiment we are interested in finding when our solutions start to become competitive and how we can tune the parameters for the best performance.

Chapter 13

Algorithm parameters and convergence

As we now know that the strongest point of our algorithm is its ability to greatly reduce ride time and time window violation at the cost of (mostly) waiting time, we are interested in finding out how much computation is needed for this. As we start out with a population that is completely random (excluding the times when local search is applied), we can be certain that our solutions will be very bad in comparison to algorithms that use routing heuristics when the algorithm is just getting started.

However, since heuristics work by limiting the search space in a smart manner, one cannot escape the inevitability of not finding good solutions because they do not conform to the way the heuristic operates (Baugh's space-time-algorithm[3] seems to miss the possibility of trading waiting time for ride time). This is not a problem when the problem is relatively small, since the heuristic should find (near) optimal solutions as long as it is given enough time and some sort of diversity operator is used. But when problems become even larger, we see that the routing heuristic cannot find better solutions (as far as the fitness function is concerned) while our GA approach is able to.

The goal of this chapter is to discuss how we can to tune the parameters such that we can find better solutions given some amount of time (more improvement per time unit). Since this goal is time related, improving the performance of the written Python code can also lead to improvements. However, optimizing Python code is outside the scope of our project and as such we focus on the parameters.

13.1 Effects of parameters on time

As stated earlier there are four relevant parameters to be optimized, namely: *population size*, *number of iterations*, *local search probability* and the *replacement percentile*. Obviously, each parameter has a different effect on how much more or less time the algorithm needs as the runtime complexity of each of the affected components is not the same. In this section we discuss how each of the parameters affects the runtime and solution quality.

Population size

The effect of the population size on the runtime can be considered to be mostly linear because the amount of work needed per unit population size is more or less fixed (it depends on other parameters). This means that it is relatively cheap to increase the population, but also means that we can expect little improvement from modifying our initial value of 50. This suspicion is also strengthened by the

fact that increasing this value means the algorithm needs more time per iteration, which results in the algorithm being able to run less iterations. Of course we cannot set the population size too small as this will result in a very fast convergence to a very bad solution, so the idea is to find the smallest population size that still results in the good solutions.

As the optimal size of the population also depends on the size of the problem itself, we would rather have the algorithm figure this value out for itself during execution. However, as such procedure would still be a heuristic and would require parameters of its own, we choose to find a fixed value for the population size that works well for all benchmark problems.

Termination criterion/number of iterations

We should note here that the number of iterations is actually a termination criterion, determining when the algorithm should stop. However, we can imagine there are many possible ways to determine when to stop, but because of following reasons we decide to stop after a certain amount of time has passed, which turns this parameter into a time limit:

- One of the important characteristics of the dynamic dial-a-ride problem is the addition of limits on time available for computation. Depending on the policy of the taxi company as well as the load it may be very well necessary to compute a new schedule every t time units. So choosing a time limit as termination criterion will make it easier to adapt the algorithm for the dynamic problem.
- If we use a termination criterion based on the quality of the elite solution, we need a heuristic that determines when further improvement is unlikely. Since we are more concerced with clustering and routing, we feel that finding a good heuristic is outside the scope of this project.
- Even though we could limit our algorithm to a fixed number of iterations like Jørgensen et al. did, we believe that this method is not very useful for us. Since the speed of computer systems is always increasing, newer computers will be able to perform more iterations per time unit. As such, it is more useful (from a practical point of view) to look at the performance given some amount of time on some computer system. Also, using a time limit also has the added advantage that if we have a faster system, we are almost always guaranteed a better solution due to the fact that the faster system could run more iterations while we do not need to modify anything. Obviously, if we are happy with the current results, we can always reduce the time limit when running the solver on a future system.

Considering the usage of the algorithm in a dynamic setting, added complexity by finding good stopping moments and the usage policies of the DAS-4 super computer, we decide to use a 15 minute time limit. We acknowledge that this might be too much time to be of use in the dynamic setting, but we have to note that much of the time our algorithm needs can be saved if we fix certain aspects of a solution for future iterations (which is possible when dealing with the dynamic problem). Finally, if we consider the time we needed for each of the benchmark instances in the previous experiments we think that 15 minutes is more than enough time for both solvers to converge.

Local search probability

As our local search procedure is the most responsible for the quality of our results, the variable governing when this routine is started is very important. We suspect that altering this variable has a

great influence on the running time, as its complexity of $O(m^2)$ with m being the size of a vehicle's schedule can be near $O(n^2)$ as the number of taxis becomes less. On the other hand, increasing this variable should also have a very positive effect on the solution quality, as this means that more iterations will produce a local search improved result.

As stated earlier, we would like to increase this variable as much as possible given our time limit, meaning that we have to find the highest value that still allows convergence. To reduce the amount of work, testing on the largest instance and smallest instance should be sufficient, as the algorithm should perform similarly for the other instances.

Hypothesis 13.1.1. If we modify the local search probability, the results found by the algorithm for the largest and smallest problem instances are representative for the results of the entire population.

We will not test whether hypothesis holds because of obvious reasons, so this should be verified in future work.

Replacement percentile

In each iteration we select one element to be replaced, this element is selected from the worst few solutions in the population. The number of worst solutions we consider depends on the value of the replacement percentile z. This value was set to 0.10 in our previous tests, which means that one of the elements in the worst 10% of the population is replaced.

Note that decreasing or increasing this value has no significant effect on the running time of the algorithm, but will result in the algorithm becoming very protective of the existing population or almost random with respect to choosing which element is replaced. However, we suspect that increasing this parameter results in slightly better solutions due to the increased diversity while decreasing it will make the algorithm converge slightly faster. Our overall hypothesis is that modifying this variable will not yield any significant changes results.

Hypothesis 13.1.2. Modifying z will not result in significant positive results but may result in very bad results

13.2 Convergence for the default settings

In this section we discuss the results we achieved while testing how fast our algorithm converges while using the default settings as described in Table 10.1 on page 36. However, as mentioned in the previous section, the parameter for the number of iterations is changed into a time limit of 15 minutes. Also, in addition to taking the average of our 10 runs per instance, we also select the best result of both solvers.

The goal of this experiment is to provide a baseline to compare further experiments with changed parameters, to determine which algorithm produces the best solution when given more than enough time and hopefully show us why our algorithm performs somewhat worse for small problems.

Given the results from our previous tests, we are fairly certain that our algorithm will still have better solutions as the solver by Jørgensen et al. was already tested using their optimal settings while our solver used somewhat arbitrary parameters. Basically, we hope to show that Hypothesis 10.0.3 on page 35 holds even when we are using sub-optimal settings.

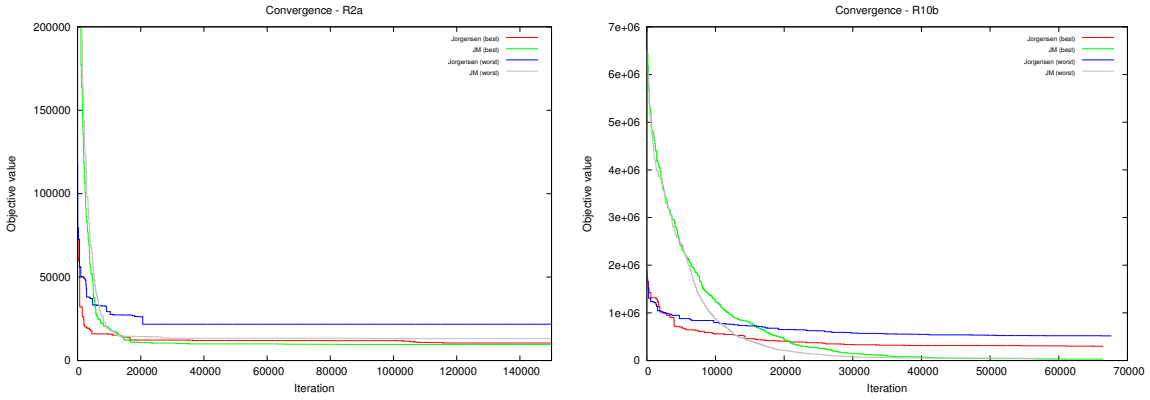
As far as the performance on small problems is concerned, we already expressed in previous sections that this is probably due to lack of time, which means that in the following experiments we should see

improvements comparable to those found for larger problems. The data regarding this experiment is shown below and graphs for the best and worst cases can be found in the Appendix on page 71. The graphs of the two most interesting cases are also shown below to illustrate the effect of problem sizes on the convergence.

Table 13.1: **Convergence after 15 minutes -** After convergence our solver performs better for all instances, although the difference for small problems is much less than larger instances. Note that the overall ratio (average of all ratios) has dropped from 0.52 to 0.42, which means the difference between the two solvers actually increased slightly. The p-value for R2a and R2b are above the critical value, indicating that the null-thesis of $\mu_{Jorg_i} = \mu_{JM_i}$ cannot be rejected. The value of Ratio (best) indicates the ratio between the best solutions of both solvers.

			Jørgensen et al.		JM				
Name	n	m	Average	Best	Average	Best	Ratio (all)	Ratio (best)	p (all)
R1a	24	3	4977	4271	**4229**	**3905**	0.85	0.91	0.00
R2a	48	5	12933	10152	**11165**	**9598**	0.86	0.95	0.11
R3a	72	7	45271	24804	**13620**	**13144**	0.30	0.53	0.00
R4a	96	9	72411	46957	**16464**	**15162**	0.23	0.32	0.00
R5a	120	11	97356	70611	**19515**	**18445**	0.20	0.26	0.00
R6a	144	13	217624	183787	**24740**	**24055**	0.11	0.13	0.00
R7a	36	4	8518	7069	**6364**	**6000**	0.75	0.85	0.00
R8a	72	6	59226	41219	**15324**	**12726**	0.26	0.31	0.00
R9a	108	8	239824	195738	**52309**	**37038**	0.22	0.19	0.00
R10a	144	10	485267	394907	**54542**	**38731**	0.11	0.10	0.00
R1b	24	3	3816	3724	**3662**	**3465**	0.96	0.93	0.08
R2b	48	5	9663	7398	**6606**	**6125**	0.68	0.83	0.00
R3b	72	7	17648	14422	**11593**	**11148**	0.66	0.77	0.00
R4b	96	9	36846	22489	**14205**	**13504**	0.39	0.60	0.00
R5b	120	11	80476	40093	**16940**	**15965**	0.21	0.40	0.00
R6b	144	13	112640	74104	**21005**	**20160**	0.19	0.27	0.00
R7b	36	4	6602	5607	**5527**	**5120**	0.84	0.91	0.00
R8b	72	6	32746	16165	**11704**	**10919**	0.36	0.68	0.00
R9b	108	8	129361	96162	**16650**	**15992**	0.13	0.17	0.00
R10b	144	10	416621	300703	**34839**	**28744**	0.08	0.10	0.00
Overall							0.42	0.51	0.00

Figure 13.1: **Comparing the worst and best results** - In our worst result (left) it can be seen that our algorithm starts of very bad (which is expected), but fails to converge to a better solution whereas the solver by Jørgensen et al. starts off significantly better. In the second example it is more obvious that our algorithm starts with very bad solutions (several orders of magnitude), but starts finding competitive solutions after about 25% of the time available has passed while the remaining time is used to increase the widen the gap before eventually converging.



It can be seen that the results of this experiment are similar to our baseline results, as the overall difference between the two is only about 10%. However, we now outperform the algorithm by Jørgensen et al. on all instances even though the relatively worse performance is still visible. This seems to suggest that both algorithms do not seem to be able to find anything significantly better than they did in the performance baseline experiments. Interestingly, for the instances R2a and R1b we cannot reject the null-hypothesis $\mu_{Jorg_i} = \mu_{JM_i}$, which may mean that the results of the two algorithms are more or less equal in terms of objective value for those two problems.

Another interesting observation is that the average ratio between the best results is higher than the ratio over all results, this means that the results by Jørgensen et al. are more diverse than ours.

Strangely, there are some occasions where our solver actually manages to perform more iterations than the other solver, this is particularly the case for problems R6a and R6b, which are large instances with a higher number of vehicles. We suspect this may be due to the job/vehicle ratio of $\frac{144}{13}$, as our local search is exhaustive whereas the routing strategy by Baugh et al.[3] is a heuristic that only considers a few locations when choosing the next stop.

When we refer to Figure 13.1, we see that our suspicions about very bad starts are confirmed as we start with solutions that are several orders of magnitude worse for large problems. On the other hand, after a small amount of time our algorithm outperforms the other solver (either by a lot or just barely), which means our algorithm seems to be profiting from the extra diversity the crossover operator provides. As can be seen above, the routing heuristic cleary performs best for quite a while (at least a few thousand iterations). When dealing with a small problem the routing heuristic seems quite adequate in finding a good solution fast. However, for the larger problems it is evident that the routing heuristic converges to a rather suboptimal solution in comparison to our algorithm. It has to be noted that we can also see that our method is more computationally intensive, as we can perform less iterations in the same time (refer to the other graphs in the Appendix).

The graph also shows that convergence takes only about two-thirds of the time allocated to the algorithm, this is more or less the case for both the best and worse result. We can imagine that if

convergence always occurs at this place, we could try to spend the time more usefully by performing more local search iterations (more improvements per iteration), larger populations (more diversity, lower point of convergence) or a combination of both.

Another possible way of improving our algorithm is by running it multiple times in succession, as can be seen in the graph for problem R2a, our algorithm converges very fast given the time limit. We could improve its performance by restarting the algorithm (generating a completely new random population) a couple of times to find alternative results and can return the best solution amongst them. However, seeing as that this method is not applicable for all instances we do not implement this idea.

Chapter 14

Balancing local search and population size

Given the effects of population size p and local search probability p_{ls} on the runtime as well as the solution quality it is important to find good values for both. Note that from our previous experiments it shows that the default settings are already quite good, but we cannot rule out that there are possibilities left for further improvement.

In this subsection we perform simple experiments to empirically find better values for the local search probability and the population size. When we have found such values, we continue by testing different values for the replacement percentile. The reason for this setup is because p and p_{ls} are also the most important ones, whereas we consider the replacement percentile z to be not as important.

For the first part of our experiments we define a set $P = \{30, 40, 50, 55, 60, 65\}$ and $P_{ls} = \{0.04, 0.08, 0.12, 0.16\}$. We solve instances R2a and R10a five times for 15 minutes on each combination of the values in P and P_{ls} and determine whether they improve or worsen. The value of z is kept constant to its default value of 0.10 for this part as it is not an important variable. We expect that the settings with higher p_{ls} show the most promise as the local search routine affects the solution quality a lot. However, we suspect that no significantly better results will be found due to the fact that there is not a lot of time left after convergence for larger problems.

Hypothesis 14.0.1. Due to the balancing effect of the local search probability and population size we cannot find significantly better results, although we can find significantly worse ones.

In the second part we test the less important parameter z on any interesting settings to see whether this parameter does make a difference. The reason why we choose to test the settings only on two problems at is stage is due to the great amount of computation needed if we were to test every problem for each setting, instead we try to find the best settings for these particular problems. As one of the tested problems is one of the largest, we do not expect other problems to have problems due to a smaller population if the tested problem has none. Similarily, we do not expect that changing the value of p_{ls} will affect other problems too negatively if it does not do so for the larger problem.

14.1 Initial testing of new parameters

As can be seen in the table below the results show that no significant improvements can be found, confirming Hypothesis 14.0.1. There are only two settings that seem to improve a the results a little bit. So we can conclude this part of the testing with the notion that our values for p and p_{ls} are quite good already. The only two other settings that seem useful are $p = 40, p_{ls} = 0.12$ and $p = 40, p_{ls} = 0.16$,

although more testing is necessary to see whether this difference is statistically significant. However, since the improvement is only a mere 2% it does not really matter for our previous results.

Table 14.1: **Initial testing of new parameters -** Comparing the results of the new settings to the reference setting of 0.08 and 50. Each setting was tested five times on problems R2a and R10a. The values in the table are average ratios. Note that all settings produce comparable results, although there are 2 settings that perform 2% better than the default. The time limit was set to 15 minutes.

$p \backslash p_{ls}$	0.04	0.08	0.12	0.16
20	1.14	1.04	1.12	1.15
30	1.01	1.20	1.08	1.03
40	1.07	1.01	**0.98**	**0.98**
50	1.16	***1.00***	1.02	1.08
55	1.07	1.12	1.10	1.12
60	1.09	1.07	1.04	1.05
65	1.09	1.03	1.10	1.04

14.2 Modifying the z-value

As we have seen that no significant improvements can be found for p and p_{ls} we think that modifying z will not yield any significantly change. The most important reason is that the speed of the algorithm is hardly changed as the algorithm only has to check the fitness value of a few more solutions. In our algorithm this procedure only computes this value once for every solutions as it caches it after computation. Therefore we only need to do very few extra computations if z increases. We tested the initial settings $p = 50, p_{ls} = 0.08$ on $Z = \{0.02, 0.05, 0.10, 0.25, 0.50, 0.98, 1.00\}$ by solving R2a and R10a five times each. This yielded the following results:

Table 14.2: **Experimenting with z -** Comparing the results of the new settings to the reference setting of $p_{ls} = 0.08$ and $p = 50$. Each setting was tested five times on problems R2a and R10a. The values in the table are average ratios. Surprisingly, it seems that decreasing z does not worsen the solution quality but may even improve them (R10a with $z = 0.05$). Furthermore, setting $z = 1$ is a very bad idea as one is practically guaranteed a very bad solution. Time limits were set to 15 minutes. The columns suffixed with *-p are the p-values that were calculated using the reference values.

$z \backslash$ Problem	R2a	R2a-p	R10a	R10a-p
0.02	0.90	0.26	0.90	0.44
0.05	**0.93**	**0.31**	**0.74**	**0.03**
0.10	1.00	-	1.00	-
0.25	1.04	0.54	1.20	0.18
0.50	0.83	0.10	1.02	0.90
0.98	8.83	0.00	43.54	0.00
1.00	25.20	0.00	102.78	0.00

Surprisingly, the results show that $z = 0.05$ may be a very interesting parameter to test as it does not seem to decrease the solution quality but may very well improve it significantly in some

cases (26%). Even more surprisingly, the corresponding p-value is below the critical value, indicating that the difference is actually significant. Note that setting z very high is detrimental to the solution quality as we now have disabled elitism. To test whether the improvement found also happens for other instances we simply run the algorithm again on all problems for setting $p = 50, p_{ls} = 0.08, z = 0.05$. The results of this test can be found below.

Table 14.3: **Experimenting with z -** Running every instance ten times 15 minutes for $p = 50, p_{ls} = 0.08$ and $z = 0.05$ yielded the following results. Unfortunately, the results of the previous experiment were not conclusive and it can be seen that the new z-value is not a good idea as the average ratio is 1.03. Interestingly, the p-values for R3a, R8a and R1b are relatively low. The overall p-value is indicates these results may be insignificant.

Name	n	m	$z = 0.10$	$z = 0.05$	Ratio	p-value
R1a	24	3	4229	4341	1.03	0.31
R2a	48	5	11165	11877	1.06	0.22
R3a	72	7	13620	14273	1.05	0.04
R4a	96	9	16464	16472	1.00	0.98
R5a	120	11	19515	19771	1.01	0.61
R6a	144	13	24740	24848	1.00	0.76
R7a	36	4	6364	6436	1.01	0.48
R8a	72	6	15324	18549	1.21	0.04
R9a	108	8	52309	52803	1.01	0.91
R10a	144	10	54542	53028	0.97	0.77
R1b	24	3	3662	3841	1.05	0.06
R2b	48	5	6606	6572	0.99	0.81
R3b	72	7	11593	11593	1.00	1.00
R4b	96	9	14205	14209	1.00	0.99
R5b	120	11	16940	16873	1.00	0.85
R6b	144	13	21005	20959	1.00	0.75
R7b	36	4	5527	5604	1.01	0.47
R8b	72	6	11704	11644	0.99	0.81
R9b	108	8	16650	16687	1.00	0.87
R10b	144	10	34839	38832	1.11	0.31
Average ratio/p-value					1.03	0.29

14.3 Conclusion

From Table 14.3 it can be seen that the results are both insignificant and worse than the reference values. Furthermore, R10a performs the best here, showing a 3% improvement. We can conclude that Hypothesis 13.1.2 indeed holds with the exception of $z = 1.00$, which basically negates anything the local search/crossover does as any element of the population can be replaced regardless of quality.

Chapter 15

Comparison to variable neighborhood search

As we have now shown that our algorithm indeed outperforms the algorithm by Jørgensen et al.[14] and that we cannot find any significant improvement by tweaking the parameters p, p_{ls} and z, we would now like to see how our algorithm compares to the results by Parragh et al.[17], who claim an improvement of 71%. As we could not find the source code for their algorithm we could not make a detailed comparison as we did for Jørgensen et al. However, we can make a simple threeway comparison between the results available to see how good our improvements really are. The results of the comparison are below:

Table 15.1: **Threeway comparison of results** - Results of Jørgensen et al. and Parragh et al. were averaged over 5 runs and are taking from the paper by Parragh et al.[17] Our results are averaged over 10 runs. All CPU times are listed in minutes. It can be seen that of the results listed, Parragh et al. perform even better than we do. Note that the CPU times were found on different systems. Since time window constraints are considered hard by Parragh et al. their objective values are naturally much lower. The scores of our algorithm after the 15 minute convergence test are also posted as a reference.

	Jørgensen	JM (norm.)	Parragh	JM (15min)	Jørg. CPU[a]	Parr. CPU[b]	JM. CPU[c]
R1a	4696	6206	**3234**	4229	5.57	2.70	0.21
R2a	19426	19518	14640	**11165**	11.43	5.16	0.51
R3a	65306	26329	15969	**13620**	21.58	6.38	0.95
R5a	213420	40228	23852	**19515**	58.23	13.93	2.32
R9a	333283	191923	**13806**	52309	40.78	33.53	2.03
R10a	740890	332969	**25016**	54542	65.98	40.27	3.36
R1b	4762	4731	**2825**	3662	5.46	3.78	0.22
R2b	13580	10512	**5003**	6606	11.72	8.29	0.53
R5b	98111	26829	**12360**	16940	58.93	23.19	2.29
R6b	185169	32464	**16499**	21005	81.23	26.39	3.31
R7b	9169	6993	**4601**	5527	8.29	4.49	0.35
R9b	167709	74304	**13412**	16650	44.66	30.32	2.00
R10b	474758	214800	**16420**	34839	66.41	51.81	3.42
Avg	179252	75984	12895	20046	36.94	19.25	1.65
Ratio[d]	1	0.60	0.28	0.33			

[a] Intel Celeron 2.00 GHz
[b] Intel Pentium D 3.20 GHz
[c] Intel Xeon E5620 (1 core at 2.40GHz), normalized to Jørgensen et al. and used as termination criterion
[d] Average of the average ratios in comparison to Jørgensen et al.

From the results we can see that our solver does not even come close to the results by Parragh et al. if we used the normalized time limits as termination condition. However, for three problem instances we are able to produce a better solution if we use a 15 minute time limit. As we do not know the relative performance between our system and the one used by Parragh et al. it is very difficult to say which algorithm is better. This is further complicated by the Hyper-Threading technology used in both the Pentium D and the Xeon processor, which improves performance by doing more computations in parallel, for example, the dual Xeon setup reports 16 logical processors, indicating that Hyper-Threading is used. To be able to say something more about the performance difference we compare the reported scores for two different benchmarks, which can be seen below. If we were to look at CPU performance instead of single core performance we can see that our system outperforms a generic Celeron processor by a factor 18 if we are using the Geekbench scores. A very useful feature of these scores is that they are linear, so a processor with score $2x$ is twice as fast as another processor with score x. However, this seems to conflict with our results because one core of the Xeon processor is about 22 times faster than the Celeron system according to our experiments. Depending on the way we calculate the performance of one core we can estimate that our system is somewhere between 1.4 (Geekbench) to 3 (CPU Mark) times faster than the Pentium D system.

Table 15.2: **Performance comparison of the systems used** - Comparing CPU core performance becomes even more difficult as we need to decide how to handle threading.

System	Clock	#Cores	#Threads	CPU Mark[a]	Geekbench[b]
Intel Celeron	2.00	1	1	233	600
Intel Pentium D (assuming 940)	3.20	2	4	892	1947
2x Intel Xeon E5620	2.40	8	16	$\approx 2 \times 5113$[c]	11081[d]

[a] Refer to `http://www.cpubenchmark.net/cpu_list.php` (Accessed on 19-07-2011)

[b] Refer to `http://www.primatelabs.ca/geekbench/pc-benchmarks/` and `http://browse.geekbench.ca/` for more information on Geekbench scores (Both accessed 19-07-2011).

[c] There was only data available for one Xeon processor, we estimate a dual CPU setup to be about twice as fast

[d] Real value, calculated on the DAS-4 cluster, average of 10 runs

As far as the objective values are concerned, Parragh et al. are able to find solutions where none of the time windows, maximum ride times and maximum work times are violated. Interestingly, Table 12.1 on page 42 shows that we also manage to reduce the amount of violations close to zero at the cost of increased customer waiting time. Regardless of the time used, our results are actually quite competitive to the ones of Parragh et al. in many occasions even though they seem to outperform our solver significantly on the largest instances. Obviously, since they only posted their results on a subset of all available problem instances we cannot say what the results would be for all problems.

Part V

Conclusion and future work

Chapter 16

Conclusion

In this part we briefly summarize the work we have done and discuss some topics for future work. This chapter begins with a summary of characteristics of the RIGA, followed by a discussion on how it performs in comparison to other solvers.

In this thesis we have presented the Random Insertion Genetic Algorithm (RIGA) for the static dial-a-ride problem with time windows, its main characteristics are as follows:

- The RIGA does not use routing heuristics to determine routes, but relies on the GA to find good solutions, this greatly simplifies the routing step.
- Even though the RIGA uses the cluster-first-route-second strategy, it deviates from other implementations as it lets the GA solve the whole dial-a-ride problem and not just the clustering part.
- There are very few parameters that need to be set, making it easy to run the algorithm on a problem. Furthermore, the few parameters that need to be set have no significant effect on the results, so as long as they are not set very badly, setting them suboptimally will not influence the results significantly.
- The algorithm of the RIGA is very simple as it is mostly a general genetic algorithm with one local search procedure, this makes it very easy to adapt it to other situations or variants of the DARP.

The RIGA significantly outperforms the solver by Jørgensen et al.[14] on almost all problem instances introduced by Cordeau et al.[7] We have argued that the space-time routing heuristic by Baugh et al.[3] is not always a good choice due to its bias towards minimizing travel and waiting times (hence, space-time) and that a randomized approach that focuses on the objective value can actually find much better results (even more so if the problem is large). However, the routing heuristic is still a better choice if one wants solutions very fast (within 5 minutes or so) as the RIGA starts with worse solutions and needs some time to catch up. This means that for situations where less than 5 minutes are available the algorithm by Jørgensen et al. may be a better choice. However, in other situations our algorithm is a good replacement for the algorithm by Jørgensen et al.

Upon inspection of the solution quality we saw that our algorithm is especially beneficial if time window violations and excess ride/work times are important, as they are significantly improved upon. It also showed that our solver improves (lower objective values) on Jørgensen's results by a factor 0.52

when using normalized time limits and by factor 0.42 when using a time limit of 15 minutes on the DAS4 supercomputer.

Finally, we compared our results to the ones found by Parragh et al.[17] and saw that (with some exceptions) the results are comparable, although we cannot say anything about whether the comparison is completely fair or not.

Chapter 17

Future work

Many interesting elements of our algorithm and solver have been left unexplored. This chapter lists some ideas that could be used to further improve upon the RIGA.

Firstly, a better comparison to the algorithm by Parragh et al. would be very educational, as their idea of temporarily allowing violated time constraints as long as the final solution does not have them is very interesting and we would like to see if we could implement that feature as well. Obviously it is also interesting to see how the RIGA fares against other algorithms under fair circumstances as this helps us identifying other possible ways of improvement. However, if we want to do so it will be necessary to have access to the source code of their algorithm.

Secondly, we introduced a number of unverified hypotheses in our thesis regarding some design choices of our algorithm such as the implementation of the best insertion and effects of parameters on solution quality. We are interested in confirming or rejecting these claims as they allow us to optimize our algorithm even more. However, this requires us to run a significant amount of experiments so we could not perform them in this thesis.

As stated in the problem definition, our ultimate goal is to solve the dynamic dial-a-ride problem. If we want to do so by using the algorithm we presented we need to account for several more issues:

- **Pareto optimality** - Even though we deal with multiple objectives in our work, we are of the opinion that using Pareto optimality is a more elegant way of solving the DARPTW and especially the DDARPTW. Given the nature of Pareto optimality it is very appropriate to use genetic algorithms as they can easily produce the multiple solutions needed to populate the Pareto frontier.

- **Incremental solutions** - As a schedule is executed by the drivers, parts of the solution will become locked in the sense that they cannot be modified or removed.

- **Event handling** - As explained in Section 4.5, we need to react on each of the events that were discussed by modifying the problem and/or the solution.

- **Performance** - Since we do not know when the next event will come in, we must have a new solution ready whenever one is needed, which gets increasingly more difficult as problems become larger.

In the following subsections we will go into more detail about each of these issues and suggest possible solutions.

17.1 Pareto optimality

As mentioned in our thesis, there is not much existing work on the combination of Pareto optimality, genetic algorithms (that encode the entire solution) and the dynamic dial-a-ride problem. We believe that Pareto optimality is essential to sole the dial-a-ride problem elegantly, as it provides a better way to handle the non-commensurable objectives. Because quality is very hard to measure and is usually composed of several non-commensurable subobjectives, it is difficult to define a good objective function that uses a weighted sum approach. Another argument for using genetic algorithms is that they are very suited for problems that use Pareto optimality as they already maintain a whole population of solutions during each iteration. This makes it much easier to select elements from the Pareto front.

As we have performed some undocumented experiments on adding Pareto-optimality to RIGA using parts from the NSGA-II algorithm by Deb et al.[8], we saw that performance significantly degrades as non-dominated-sorting and maintaining Pareto fronts is a very computationally intensive task. This means that the speed of the algorithm becomes even more important if we are to solve the dynamic variant which places more time constraints on the solver. A less complicated subject for future work would be to test our algorith on different (weighted sum) objective functions to see how it performs when time window violations and excess ride/work times are not as important.

17.2 Incremental solutions

Since events cause the problem itself to change, the solution to the problem has to change accordingly. However, if we assume that the change is small it is not very efficient to compute a new solution from scratch. Insertion heuristics are a logical way to build solutions incrementally as they insert the new information into some previous solution. Even though we can simply restart the whole computation when new information is available, we think that that time could be spent in better ways instead of redoing mostly old computations. Therefore, investigating how to add some insertion-heuristic-like qualities to our algorithm is certainly interesting.

Some additional background information on the dynamic dial-a-ride problem is discussed by Larsen et al.[2], who discuss dynamism in more detail and provide a metric to measure how dynamic a problem is. We imagine that this may be used to solve certain parts of a problem instance using other algorithms depending on the level of dynamism.

We do have to note that insertion heuristics may not even be necessary, as the insertion algorithm presented in Subsection 9.1 can be easily adapted to lock a portion of the solution by letting s_i be the last stop of the locked portion of the solution. This means that the insertion itself becomes much faster as less values of s_i (and by extension s_k) need to be evaluated. This is a rather simple modification, but may provide just enough performance improvement to make the RIGA competitive for dynamic problems. However, deciding how to implement the procedure to mark part of a solution as immutable will require much thought, as the algorithm needs to decide when a stop within a schedule may not be changed anymore.

17.3 Event handling

Xiang et al.[20] provide a framework for the dynamic dial-a-ride problem that uses an event-driven approach. We also believe that an event-driven approach is appropriate given the nature of the dynamic variant. The idea of dealing with events is quite simple. For example, if a customer cancels his job, the algorithm can remove the job from its schedules and recompute schedules. Similarly, if a vehicle breaks

down all customers who were on board will have their jobs cancelled and new jobs will be created with the break down location as the pickup stop.

17.4 Performance

We suspect that by carefully applying unbiased (insertion) heuristics it should be possible to create a genetic algorithm that can handle events more quickly. Furthermore, another way to determine the initial population may be necessary, as the solutions in the initial population are often many orders of magnitude worse than the final solution, so much time can be saved if we can start with a more reasonable population. These improvements are necessary as we have seen that improving the solution quality using parameter tweaking alone is very difficult.

However, as we have also discussed in Chapter 10, we are already assured of improved solution quality given a constant time limit as hardware improves, so if we can develop a faster algorithm we have good hopes of being able to solve all the issues (dynamism, Pareto optimality) that were presented.

Bibliography

[1] Alberto Colorni and Giovanni Righini. Modeling and optimizing dynamic dial-a-ride problems. *International Transactions in Operational Research*, 8(2):155–166, 30 June 2000.

[2] Allan Larsen, Oli B.G. Madsen, and Marius M. Solomon. *Dynamic Fleet Management*, chapter Classification Of Dynamic Vehicle Routing Systems, pages 19–40. Springer US, 5 October 2007.

[3] John W. Baugh, Gopala Krishna Reddy Kakivaya, and John R. Stone. Intractability of the Dial-a-Ride Problem and a Multiobjective Solution using Simulated Annealing. *Engineering Optimization*, 1998.

[4] K.B. Bergvinsdottir. The Genetic Algorithm for solving the Dial-a-Ride Problem. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, April 2004.

[5] Ralf Borndörfer, Martin Grötschel, Fridolin Klostermeier, and Christian Küttner. Telebus Berlin: Vehicle Scheduling in a Dial-a-Ride System. In Nigel H. M. Wilson, editor, *Computer-aided transit scheduling. Proceedings, Cambridge, MA, USA, August 1997*, volume 471, pages 391–422. Springer, 1999.

[6] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Jean-François Cordeau and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*, 37(6):579 – 594, 2003.

[8] Kalyanmoy D. Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[9] Dusan Teodorovic and Gordana Radivojevic. A fuzzy logic approach to dynamic dial-a-ride problem. *Fuzzy Sets Systems*, 116(1):23–33, 2000.

[10] Elian Swinkels; Jurgen Visser; Matthijs de Gier. De kwaliteit in het contractvervoer. Technical report, TNS Consult, 22 February 2010.

[11] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Vehicle dispatching with time-dependent travel times. *European Journal of Operational Research*, 144(2):379 – 396, 2003.

[12] Jang-Jei Jaw, Amedeo R. Odoni, Harilaos N. Psaraftis, and Nigel H. M. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B: Methodological*, 20(3):243 – 257, 1986.

[13] Jean-François Cordeau and Gilbert Laporte. The Dial-a-Ride Problem (DARP): Variants, modeling issues and algorithms. *4OR: A Quarterly Journal of Operations Research*, 1(2):89–101, 1 August 2002.

[14] R. M. Jørgensen, J. Larsen, and K. B. Bergvinsdottir. Solving the dial-a-ride problem using genetic algorithms. *The journal of the Operational Research Society*, 58:1321–1331, 2007.

[15] Oli B.G. Madsen, Hans F. Ravn, and Jens Moberg Rygaard. A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives. *Annals of Operations Research*, 60(1):193–208, 1995.

[16] Julie Paquette, Jean-François Cordeau, and Gilbert Laporte. Quality of service in dial-a-ride operations. *Computers and Industrial Engineering*, 56(4):1721 – 1734, 2009.

[17] Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl. Variable neighborhood search for the dial-a-ride problem. *Computers and Operations Research*, 37(6):1129 – 1138, 2010.

[18] Sophie N. Parragh, Karl F. Doerner, Richard F. Hartl, and Xavier Gandibleux. A heuristic two-phase solution approach for the multi-objective dial-a-ride problem. *Netw.*, 54:227–242, December 2009.

[19] Francisco B. Pereira, Jorge Tavares, Penousal Machado, and Ernesto Costa. Gvr: a new genetic representation for the vehicle routing problem. In *Problem, Proceedings of the 13th Irish Conference on Artificial Intelligence and Cognitive Science*, pages 95–102. Springer-Verlag, 2002.

[20] Zhihai Xiang, Chengbin Chu, and Haoxun Chen. The study of a dynamic dial-a-ride problem under time-dependent and stochastic environments. *European Journal of Operational Research*, 185(2):534 – 551, 2008.

Part VI

Appendix

Chapter 18

Pseudo-code of the GA

18.1 Pseudo-code of the initialization procedure

Algorithm 18.1 (Initialization) Generate a random solution

Require: The set of all taxis M
Require: The set of all jobs is N
Require: Let p_j denote the pickup location of job j
Require: Let d_j denote the drop off location of job j
Require: Let d_0 denote the depot
Require: $feasible(l, s_t) = True$ iff appending stop l to schedule s_t still results in a schedule that does not break the precedence and capacity constraints
Require: The function $retime(schedule)$ that computes and adds the arrival times for every stop in every schedule.
Ensure: All jobs fit into at least one taxi
{First determine the clustering}
for all $t \in M$ **do**
  $c_t = \{\}$
  $s_t = \{\}$
end for
for all $j \in N$ **do**
  while j is not assigned **do**
    $t = random(M)$
    if j fits into t **then**
      $c_t = c_t \cup \{j\}$
    end if
  end while
end for
{Second part, find routes}
for all $t \in M$ **do**
  $s_t = \{d_0\}$
  while $|c_t| > 0$ **do**
    $j = random(c_t)$
    if $p_j \in s_t$ **then**
      $s_t = s_t \cup d_j$ {Add as drop off location}
      $c_t = c_t - j$
    else if $feasible(p_j, s_t)$ **then**
      $s_t = s_t \cup p_j$ {Add as pickup off location}
    end if
  end while
  $s_t = s_t \cup d_0$
end for
return $retime(\{s_t : t \in M\})$

18.2 Pseudo-code of the repopulation procedure

Algorithm 18.2 (Repopulation) Generate the next population

Require: Let pop denote the current population
Require: Let P denote the require population size
Ensure: $|pop| < P$
while $|pop| < P$ **do**
  $p_1 = random(pop)$
  $p_2 = random(pop)$ such that $p_1 \neq p_2$
  $child = crossover(p_1, p_2)$ {The crossover returns either: feasible schedule or 'not possible'}
  if $child$ is possible **then**
    if local search is allowed **then**
      $child = local\_search(child)$
      $pop = pop \cup \{child\}$
    else
      $pop = pop \cup \{child\}$
    end if
  else
    continue {We repeat the while-loop again}
  end if
end while

18.3 Pseudo-code of the crossover procedure

Algorithm 18.3 Pseudo-code of the crossover procedure

Require: Let p_1 and p_2 denote the two parents
Require: Let c' denote the cluster corresponding to c (the schedule of the same taxi in the other parent)
Require: Let $random\_taxi()$ return a random taxi (not particularly related to any solution)
$child$ = Nothing
c_t = random schedule belonging to taxi t from p_1
c'_t = schedule of t in p_2
if $|c_t| = 0$ **then**
  return NotPossible
end if
$b = \{job \in c_t\}$ {Remember which jobs are in the selected cluster}
for all $(taxi, schedule) \in p_2$ **do**
  $child = child \cup (taxi, \{s : s \in schedule$ **and** $s \notin b\})$ {Copy schedules while excluding jobs in c_t}
end for
for all $job \in c'_t$ **where** $job \notin b$ **do**
  randomly insert job into a schedule s of $child$ while preserving feasibility
end for
return $retime(child)$

18.4 Pseudo-code of the best_insertion() function

Algorithm 18.4 Pseudo-code of *best_insertion*()

Require: Let *solution* denote a complete solution for the DARP
Require: Let t denote the target taxi
Require: Let s_t denote the route s_t of taxi t
Require: Let j denote the target job
Require: Let p_j denote the pickup location of job j
Require: Let d_j denote the drop off location of job j
Require: Let d_0 denote the depot
if j does not fit into t even if it were empty **then**
  return NotPossible
end if
$best$ = None
if s_t is empty **then**
  return *solution* where $s_t = \{d_0, p_j, d_j, d_0\}$
else
  for all $s'_t = \{d_0, \ldots, p_j, \ldots, d_j, \ldots\ d_0\}$ given s_t **do**
    if s'_t satisfies capacity and precedence constraints **then**
      if s'_t is better than $best$ **then**
        $best = retime(s'_t)$
      end if
    end if
  end for
  return *solution* where $s_t = best$
end if

18.5 Pseudo-code of the retime() function

Algorithm 18.5 Pseudo-code of *retime*(*solution*)

Require: The set of all taxis M
Require: Let *solution* denote a solution for some problem instance
Require: Let $s_t \in$ *solution* denote the route s_t of taxi tPseudo-code
Require: Let s_{t_i} denote the i^{th} stop in route s_t
Require: Let p_i denote the pickup stop corresponding to the job which has a stop at i
Require: Let $t_{arr}(s_{t_i})$ denote the getter/setter for the arrival time at stop s_{t_i}
Require: Let $t_s(s_{t_i})$ denote the getter/setter for the beginning of the time window of stop s_{t_i}
Require: Let sv_i denote the service time at stop i
Require: Let $t(l_1, l_2, t_1, t_{now})$ denote the traveltime

```
1:  for all t ∈ M do
2:    t = 0
3:    for i ∈ [0, |s_t|] do
4:      if i = 0 then
5:        if next stop belongs to an inbound request then
6:          t_arr(s_{t_i}) = latest time to arrive at the beginning of the window of s_{t_{i+1}}
7:        else
8:          t_arr(s_{t_i}) = latest time to arrive at s_{t_{i+1}} s.t. the taxi is exactly on time if the next stop
            t_{i+2} happens to be the drop off stop
            {Outbound request}
9:        end if
10:     else
11:       t_arr(s_{t_i}) = t_arr(s_{t_{i-1}}) + t(s_{t_{i-1}}, s_{t_i}, t_arr(s_{t_{i-1}}), t_arr(s_{t_{i-1}})) {As soon as possible, arriving too
          early incurs waiting time, which is allowed}
12:     end if
13:   end for
14: end for
```

Chapter 19

Additional empirical data

19.1 Convergence for each of the problem instances

Figure 19.1: **Convergence graphs for instances R1a to R6a -** The best and worst objective values found by both solvers after 15 minutes

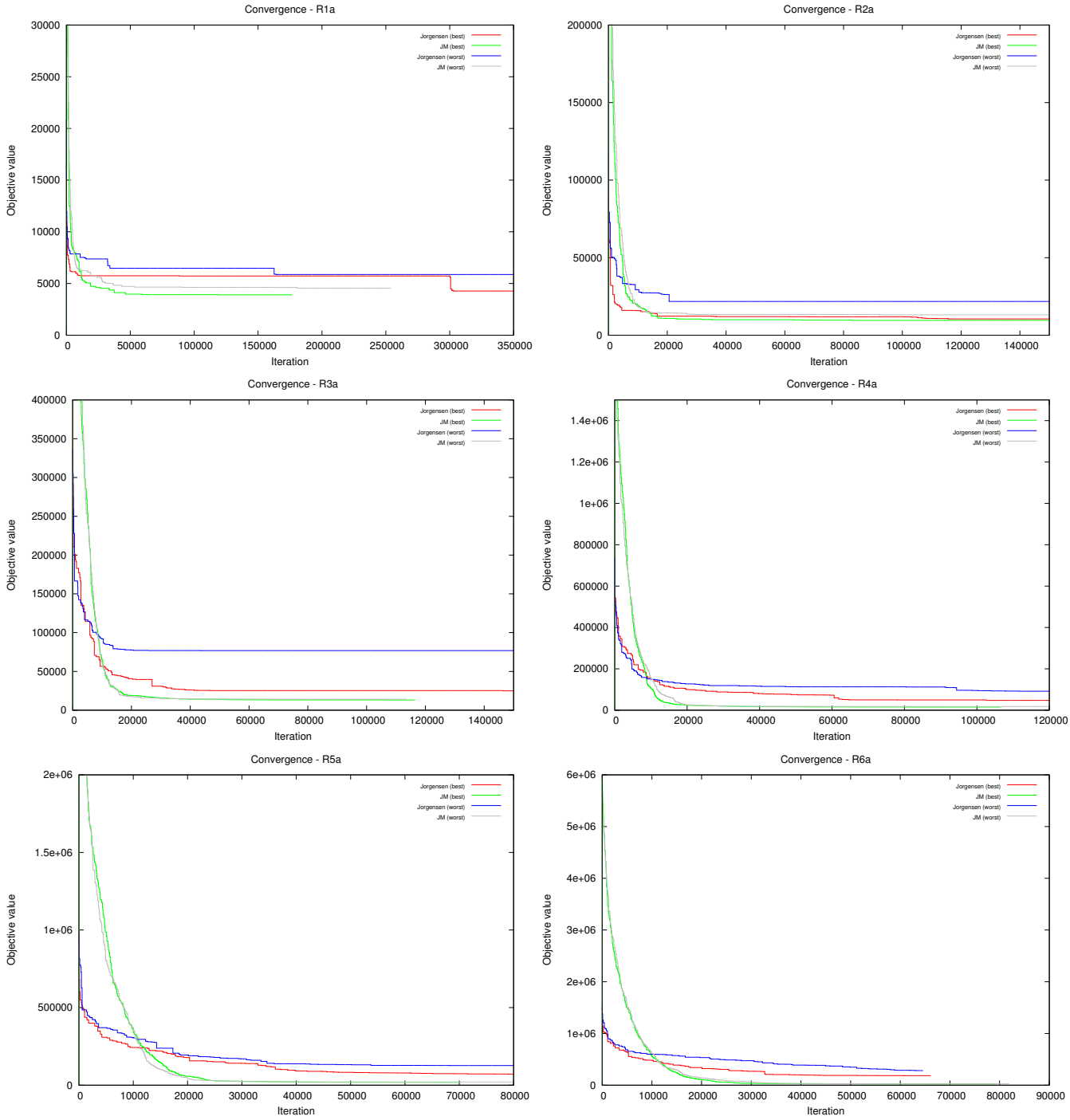


Figure 19.2: **Convergence graphs for instances R7a to R2b -** The best and worst objective values found by both solvers after 15 minutes

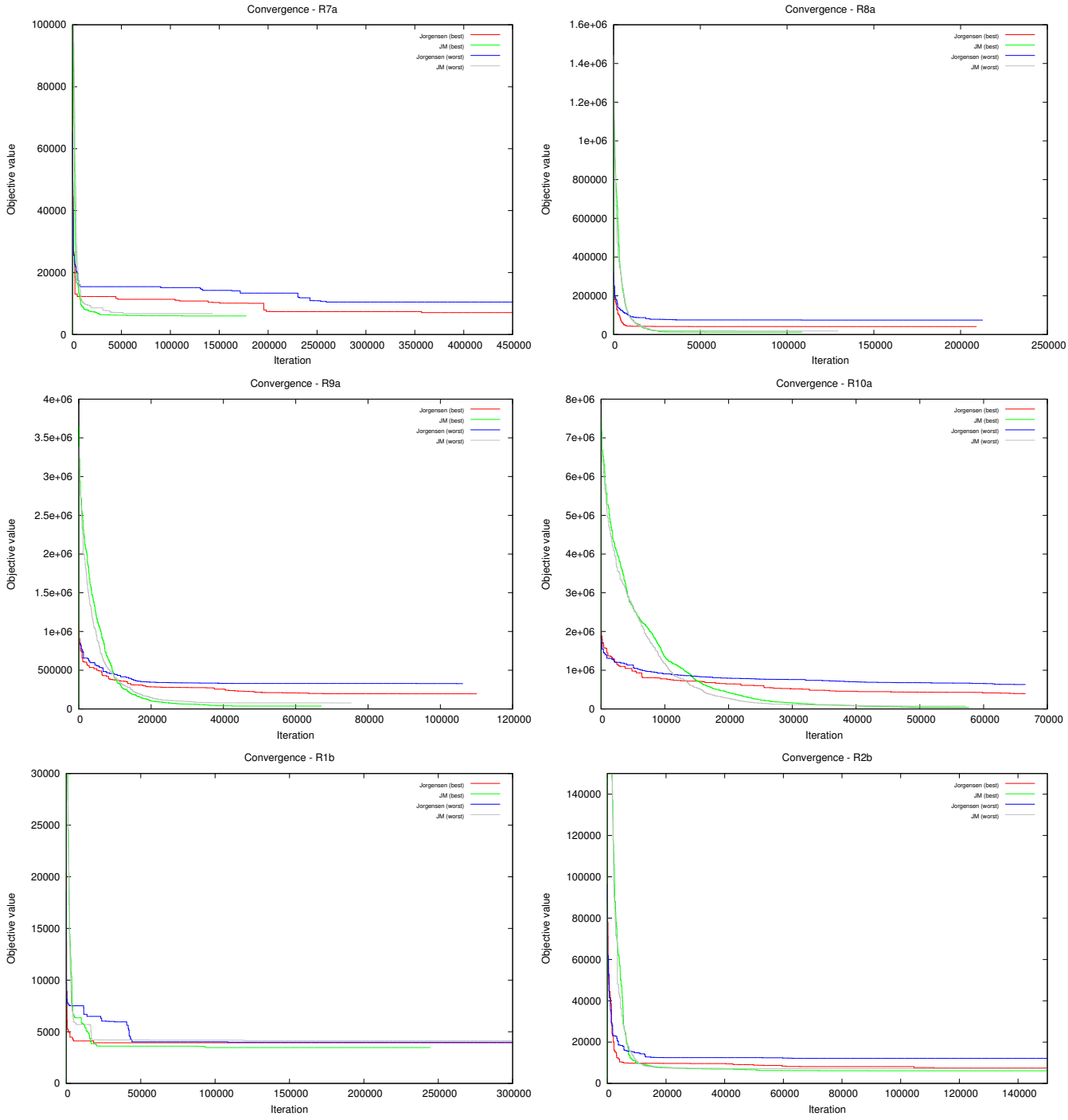


Figure 19.3: **Convergence graphs for instances R3b to R8b -** The best and worst objective values found by both solvers after 15 minutes

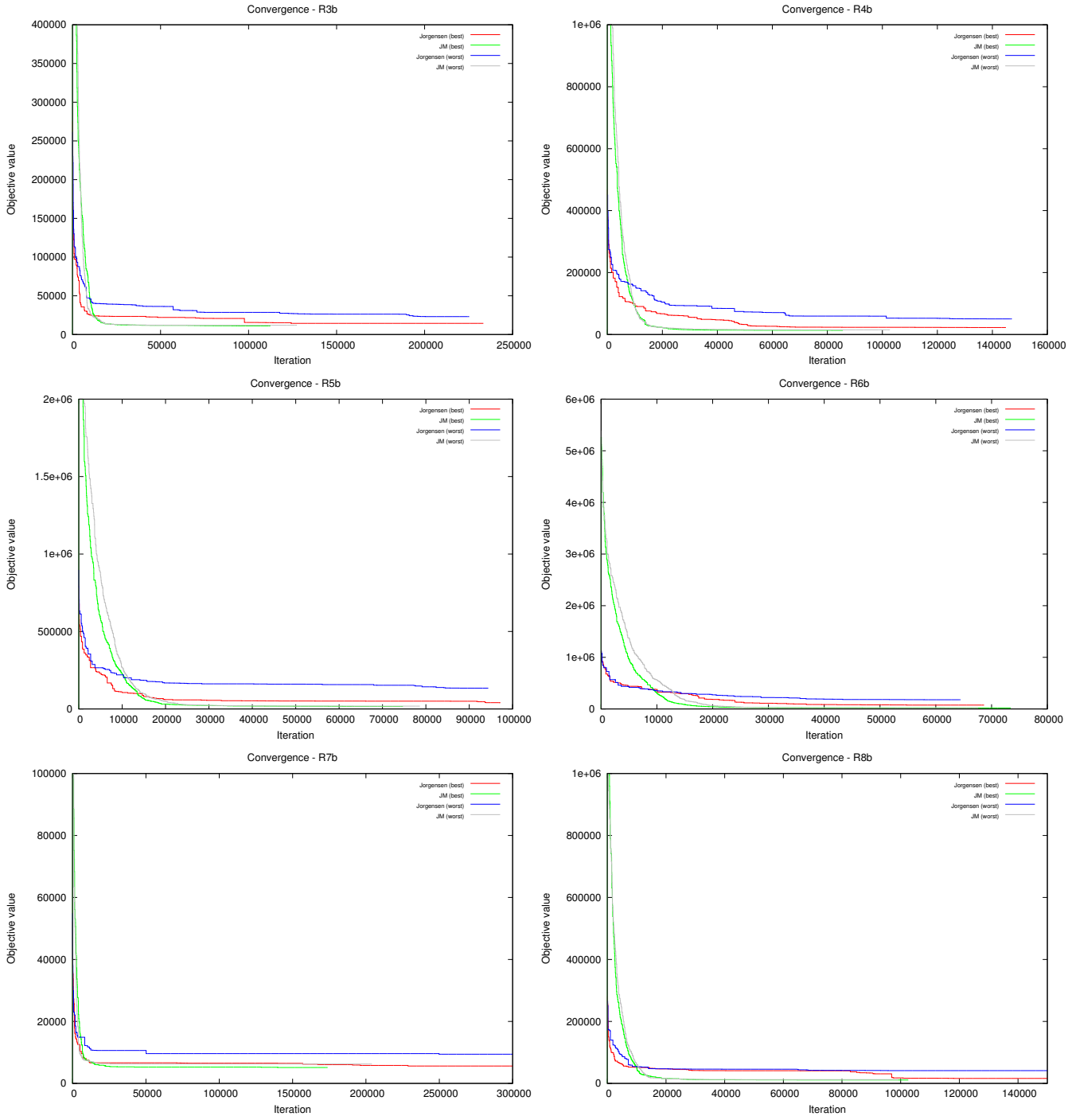


Figure 19.4: **Convergence graphs for instances R9b and R10b -** The best and worst objective values found by both solvers after 15 minutes