Taxi ride scheduling and pricing using historical data

Martijn den Hoedt

TUDelft

gogido

Taxi ride scheduling and pricing using historical data

by

Martijn den Hoedt

born in Rotterdam, The Netherlands

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Thursday August 25, 2016 at 9:30 AM.

Student number:	4143760	
Project duration:	November 9, 2015 – July 1, 2016	
Thesis committee:	dr. M.M. de Weerdt,	TU Delft, supervisor
	prof. dr. C. Witteveen,	TU Delft
	dr. ir. A. Bozzon,	TU Delft
	drs. P. Pietersen,	Gogido

Preface

Taxis often bring people to airports and drive back to their area of operation without a passenger. This is considered as a gap in a taxi driver's schedule and could be filled up by giving customers extra incentive to book a taxi ride by asking a reduced price. A reduced price is justified when the costs of when the rides are combined is lower than when the rides are carried out separately. This thesis work describes two steps towards computing the cost of performing a taxi ride while taking into account the time and fuel needed for driving empty between rides. The first method responds to a request of a customer and composes offers that are related to the customer's taxi ride request from different companies, while taking the already booked rides into account. A good offer has both the price and the offset to the requested departure time minimized. The second method learns the cost for driving empty by taking into account the probability of taxi rides that will be booked in the future.

This thesis has been submitted to obtain the degree of Master of Science in Computer Science at Delft University of Technology. I would like to thank Mathijs de Weerdt and Paul Pietersen, which are the university supervisor and company supervisor respectively, for their ideas and supervision. Additionally, I would like to thank both Cees Witteveen and Alessandro Bozzon for taking part as chair and member of the thesis committee, respectively. Finally I would like to mention that I am thankful for the support of family and friends.

Martijn den Hoedt
Delft, August 16, 2016

Contents

1 Introduction

In The Netherlands, transport by taxi can be divided into two categories, namely street taxis and contracted transportation. The latter includes the transportation of school students, disabled people and elderly people who cannot use public transportation or drive a car by themselves, this is regulated by the government. Traditionally, street taxis can be hailed on the street or requested via telephone by anyone. The latter falls under regulations for 'hired car with driver', but in contrast to other countries Dutch laws make as little distinction as possible between a street taxi and a 'hired car with driver' [63]. Therefore this thesis will use 'taxi' while referring to a street taxi or a 'hired car with driver'. Another characteristic of the Dutch taxi market is that it is uncommon for taxis to browse around a city looking for new customers and uncommon for a customer to hail an empty driving taxi, though this is not prohibited. In The Netherlands, most people order a taxi by phone or pick up a taxi on a taxi stand, which roughly corresponds with a two-thirds and one-third of the total revenue of the whole taxi market [51]. Taxi companies and drivers determine fares by themselves, but they should meet with the maximum fares determined by the government [56]. The maximum fares are not obligatory when customer and driver agree on a pre-arranged price. Extra fees can be agreed upon for extra services, such as a bottle of water or WiFi access. Recently new initiatives were launched that allow people to request a taxi via a smartphone application or a website, but this is essentially the same as by phone call.

In the year 2008 roughly a third of The Netherlands did not travel with a taxi in the year 2008 and roughly a quarter of the surveyed used a taxi only one to three times that year [51]. Taxis in The Netherlands have one of the most expensive tariffs in the world and 76% of the same surveyed think that travelling by taxi is expensive or very expensive in The Netherlands [69]. A poor availability of public transport and alcohol usage are the two most important reasons for people to use a taxi [51]. Gogido, a platform to compare prices between taxi companies and book taxi rides, is interested in reducing the prices of taxi rides. This will enlarge the taxi market and increase the profit of both taxi companies and Gogido. One way of reducing the average price for a taxi ride is to let customers share a taxi when they have an overlapping part during their trip. However this has some disadvantages as well, it will cost the customer more time and he will be less flexible to reschedule the ride, because he is now dependent on other customers. Another way of reducing the price for a taxi ride is to help companies to operate more efficient by driving less with an empty taxi. Nowadays taxis often drive empty to and from their usual area of operation to bring and pick up customers to locations outside this area. These gaps in the schedule could be offered for a reduced price to customers. Thus combining taxi rides means, in this thesis, that shortly after dropping of a customer a new customer is picked up.

In order to be able to offer a reduced price to a customer the cost of executing the customer's taxi ride for the taxi company should be determined. This can be done by comparing the total cost of a schedule with and without this new ride. Note that most taxi companies do not have a schedule such that every taxi ride is assigned to a taxi, but just a list of rides that should be done, because a lot can happen during the day that will make the schedule invalid or suboptimal. Examples of this kind of events are car breakdowns, new customers who require a taxi as soon as possible, customers that cancel a ride and customers with a meeting that runs in overtime. Sometimes a company accepts more taxi rides than it has capacity, these rides are then sold to a friendly competitor. When a new ride perfectly matches the other rides, the cost of this new ride could be near zero. However this will happen with very low probability and there can also be multiple 'good' ways of scheduling the new ride. Generally either of two possible scenarios can occur, namely the taxi driver has to wait for the customer, or the other way around. In the first case the cost for the ride increases and in the second case the customer inconvenience will rise. Thus the matching is a multi-objective optimization problem, which will be formally defined in chapter 2. Other factors that might influence the customer's choice are the reputation of the company and the type of vehicle that will be used. The reputation of a company can be measured by rating given by customers, but this is not done for all companies in The Netherlands. However Gogido enables their customers to give the company a rating based on punctuality, kindness of the drivers and other evaluation criteria. The types of vehicles a taxi company can include, but are not limited to a standard car, a luxurious car and a small bus for eight passengers. In this thesis the multi-objective optimization problem does only consider the taxi's cost and the waiting time for the customer and not vehicle type and rating, because these are fixed parameters.

Multiple problems arise when computing the cost of a taxi ride. The first one is a scalability problem, because an enormous number of taxi rides are driven nationally it seems infeasible to recompute an optimal schedule every time whenever a customer would like an offer on his taxi ride request. A second problem will arise when a customer reschedules or cancels his taxi ride after that ride was combined with another taxi ride. Because this could mean that taxi companies have to ride below cost price. This could be compensated with a fixed price increase to cover the risk, but this could also be a variable price increase to cover the risk for a specific situation. For the latter a model for the risk could be learned by using historical data from taxi companies as training set. A third problem will arise when multiple customer would like to book a similar ride, at the same moment and with the same company. Because this will influence the cost of both rides, but at the time the offer is made no information is known about a second customer requesting a similar ride.

When a company is having trouble to acquire taxi ride requests, because the prices are too high compared to competitors, the company should lower its price where possible. This could be done by not only offering customers a discount on a taxi ride that have already a matching ride, but also by offering discounts when it is likely that, in the future, a matching taxi ride request will come. Although past performance does not guarantee future results, historical data of the company could be very useful while reasoning about the likelihood of future taxi rides. The cost of a taxi ride matched with likely taxi ride requests will be called the expected cost. The formal problem definition regarding the expected cost of a taxi ride is stated in the next chapter.

The rest of this thesis is structured in the following way. Firstly, the next chapter describes the problem in a formal and more detailed way. Then in chapter 3 the related work is divided in seven sections, each describing different topics that are relevant for solving the problems this report treats. These sections give an overview on related scheduling problems, vehicle routing scheduling algorithms, multi-objective optimization algorithms and methods evaluating them, literature about taxi demand prediction, methods for clustering datasets, regression analysis and interpolation methods. Then in chapter 4 three methods for exploring the solution space are introduced. In addition a method that discards similar solutions, which can be used when one does not want to overload the customer with many possibilities, is described in chapter 4. Chapter 5 describes methods to calculate the expected cost of a taxi ride based on historical data from a taxi company. The methods described in chapters 4 and 5 are evaluated in chapter 6. Finally in chapter 7, a discussion of the assumptions is stated, possible future work is listed and conclusions are drawn.

2

Problem definition & research questions

In this chapter, a formal problem description and the research questions are stated. In addition the constraints and the scope of this research is defined.

Customers come to the Gogido website with a certain taxi ride R_i in mind. Each taxi ride R_i has a pickup time $t_i^p \in \mathbb{N}$, a drop off time $t_i^d \in \mathbb{N}$, a pickup location $p_i^p = (\phi_i^p, \lambda_i^p)$ and a drop off location $p_i^d = (\phi_i^d, \lambda_i^d)$. A location is denoted by a pair (ϕ, λ), with $\phi \in \mathbb{R}$ the latitude and $\lambda \in \mathbb{R}$ the longitude. The customer can book this ride for a normal fare or look into the situational offers. An offer can be made to the customer when we know what the increase in cost is for a taxi company C, where $C = \{T_1, T_2, ..., T_{|C|}\}$ and every taxi $T_j = \langle R_1, R_2, ..., R_{|T_j|}\rangle$. Besides, every taxi that drives for company C costs C_d per kilometre and C_t per minute. This offer $x = (C, R_i, t_w)$ is a possible solution the customer can choose for, where C is the selected taxi company, R_i the initially requested taxi ride and $t_w \in \mathbb{N}$ the offset with the requested departure time. The offer x as reaction to a taxi ride request R_i always has the same pickup and drop off location. A negative offset means that the customer has to depart earlier than the requested time, and thus a positive offset means that the customer has to depart later than the requested time. From now on the offset to the requested departure time will be called 'waiting time'. The cost of solution x is denoted by $f_c(x)$ and the waiting time t_w as $f_w(x)$. A solution x is strictly better than x' if x dominates x', which is denoted by $x \succ x'$. A solution can only dominate another solution when they both point out that the customer departs either later or earlier than the requested time. In addition one of the two objectives of x is strictly than x' and the other objective is for not worse. This means if one of the following two statements holds, the solution x dominates x':

- $f_c(x) \leq f_c(x')$ and $|f_w(x)| < |f_w(x')|$ and $\text{sign}(f_w(x)) = \text{sign}(f_w(x'))$
- $f_c(x) < f_c(x')$ and $|f_w(x)| \leq |f_w(x')|$ and $\text{sign}(f_w(x)) = \text{sign}(f_w(x'))$

Since taxi companies only exchange rides when a taxi ride is infeasible to do by them, every company can be considered separately. A taxi ride is infeasible when no taxi is available to do this ride. Since taxi companies do not use a schedule for days in advance, we do not need the computed schedule, but only the computed cost and assigned waiting time. The cost a taxi ride is not only dependent on the route the customer wants to travel, but also the time and fuel needed to drive to the pickup location from the taxi's previous location. Therefore a schedule has to be made to be able to calculate the cost and present an offer to the customer.

The aforementioned cost increase for a company C, caused by a taxi ride R requested by a customer, can be computed by solving a Static Multi-Vehicle Dial-A-Ride Problem with Time Windows (MVDARPTW) twice. Once with and once without the new ride R and in both cases with all previously booked rides of company C. The optimal solution to a MVDARPTW is an assignment for every ride to a vehicle such that the cost function $f_{cost}(C)$ is minimized. In our model we assume that every taxi company has an infinite number of taxis available, because a taxi company can sell infeasible rides to friendly competitors. The time window of a ride defines the flexibility of the departure time, since a customer wants his taxi right on time the time windows are very tight. However the time window for taxi ride R is very loose allowing solutions with a different departure time a customer might be interested in. The cost of a company's schedule $f_{cost}(C)$ defined in equation 2.2 sums up the cost of the routes of the taxis will drive. The cost of a taxi's route consists of two parts the distance of the complete route $f_{route}(T)$ and the time the taxi is needed. The time a taxi is needed includes the waiting time for a taxi driver before every ride. The route $f_{route}(T)$ always starts and ends at the home base of the taxi company and is denoted by $p_C = (\phi_C, \lambda_C)$. Note that in practice not every taxi from the same company will use the same location. The functions $f_d(L)$ and $f_t(L)$ return the distance in kilometres and time in minutes required to visit all locations $(\phi, \lambda) \in L = \langle p_1, p_2, \dots \rangle$ in the specified order. This can be implemented using the Google Maps Routing API [17] or the OpenStreetMap variant called OpenStreetMap Routing Machine (OSRM) [35]. Only OSRM is a viable option to use for this thesis, because Google limits the number of requests on their API.

$$f_{route}(T) = \langle p_C \rangle \cup \left(\bigcup_{R_i \in T} \left\langle p_i^p, p_i^d \right\rangle \right) \cup \langle p_C \rangle \tag{2.1}$$

$$f_{cost}(C) = \sum_{T \in C} \left(C_t \cdot \left(f_t(\langle p_C, p_1^p \rangle) + f_t(\langle p_{|T|}^d, p_C \rangle) + t_{|T|}^d - t_1^p \right) + C_d \cdot f_d(f_{route}(T)) \right) \tag{2.2}$$

Thus the remaining problem is to match customer requests with one or multiple matching taxi rides in the list of scheduled taxi rides. Based on these best matching taxi rides an offer can be presented to the customer. An offer includes a price and an actual departure time which can differ with the requested departure time. Therefore the first question we define is:

How can an online algorithm efficiently compute a non-dominated set of offers in reaction to a taxi ride request, given a large list of already accepted taxi rides, within a small enough period of time?

A small enough period of time is defined to be one second, since the list of offers will be made available to the customer via a web interface and literature states that humans will notice a one second interruption, but the user's flow of thought stays uninterrupted [41]. Additionally the number of solutions presented on this web interface should not be boundless. The following subquestions can be identified:

1. *How can the cost of a taxi ride be defined such that it can be efficiently computed?*
2. *How well does the algorithm scale? In other words how many already accepted taxi rides can be evaluated?*
3. *How can the list of non-dominated solutions be reduced to a specific size, such that similar solutions are discarded?*

However, if a company only offers discounts based on already booked taxi rides, it will never offer a discount to the first customer. On popular routes a discount can be given, because in the future a matching taxi ride request is likely to come in. This will increase the volume of rides and revenue even more. Therefore the second question we define is:

How can an online algorithm efficiently compute the expected cost of a taxi ride, given the historical data of a taxi company, within a small enough period of time?

Again the small enough period of time is defined to be one second. To answer this question we also need to know how to define the expected cost of a taxi ride. When no good match can be found in the future the company will suffer a loss. Therefore it is interesting to see what could be done to reduce the risk of a loss. For the second research question we have identified the following subquestions:

1. *Which features can be extracted from the historical data that influence the expected cost of a taxi ride?*
2. *How could the expected cost of a taxi ride be defined?*
3. *What are good methods to reduce the probability of a loss?*

This thesis only discusses the cost of a taxi and not the price of a taxi that will be presented to the customer, because the latter is a strategic decision and competitors' and customers' behaviour should be taken into account. During analysis only one type of vehicle is considered, but other types of vehicles could be taken into account without major adjustments of the methods proposed. However when considering more than one type of vehicle could introduce issues this is discussed.

3
Related work

This chapter, which describes the related literature, is divided into seven sections. The first section briefly explains some related problems and variants of the Dial-A-Ride Problem (DARP). In the second section different algorithms that can solve these variants of DARP. The third section describes problems that arise when dealing with multi-objective optimization problems. Section 3.4 describes the techniques used to predict taxi demand in various cities. Section 3.5 surveys clustering methods. Sections 3.6 and 3.7 describe regression and interpolation methods.

3.1. Vehicle Routing Problems

The class of Vehicle Routing Problems (VRP) consists of many different problems with many different applications. However they all share a common goal, namely given a set of transportation requests and a fleet of vehicles, determine for all vehicles a route such that all (or some) requests are served at minimum cost [66]. This section describes some subclasses and problems that are contained in these subclasses and concludes with a formal problem description found in literature accompanied with the different variants, which are closely related to the problem relevant for this thesis.

The Vehicle Routing Problems with Pickups and Deliveries (VRPPD) class consist of problems where goods or people have to be transported from a pickup locations to a delivery or drop off location. This class can be divided in problems that have paired or unpaired pickup and delivery locations. An example of the unpaired case is the Pickup and Delivery Vehicle Routing Problem (PDVRP). With PDVRP every customer at a delivery location can be served with the goods from any pickup location [47]. In the paired case every pickup location is paired with one delivery location. Two examples of this are the Pickup and Delivery Problem and the Dial-A-Ride Problem, which is about goods and people respectively. The scheduling of taxi rides or another form of transportation of people is a special case, because not only the cost should be minimized, but the customer convenience should be maximized as well. The cost is mostly measured with the fleet size and the distance travelled, while the customer inconvenience is mostly expressed as the time travelled or the deviation from the desired drop off time. DARP is proven to be $\mathcal{NP}$-hard [6], and generally all VRPs can be generalized to the well-known $\mathcal{NP}$-hard Traveling Salesman Problem (TSP) [22].

The problems mentioned in this section come in different flavours, a number of flavours that are applicable to DARP consists of, but are not limited to: static versus dynamic, single-vehicle versus multi-vehicle and with time windows versus no time windows. In the static variant of DARP the requests are known beforehand and in the dynamic variant the requests come in by an online fashion. The multi-vehicle variant of DARP has much larger solution space than the single-vehicle variant and the algorithms for the former a usually easier to understand and implement. Finally, there is a time window (DARPTW) variant, which allows customers to specify a desired pick-up time window.

The problem most related to the problem this thesis is trying to solve is the multi-vehicle DARPTW problem and can be defined as follows: Construct one route for every vehicle in the fleet of m vehicles, which together serve n customers. The road network is defined by a complete graph $G = (V, E)$, where $V = \{v_0, v_1, ..., v_{2n}\}$

and $E = \{(v_i, v_j) : v_i, v_j \in V\}$ and for every $(v_i, v_j) \in E$ the cost is defined by $0 \le c_{ij}$ and the travel time by $0 \le t_{ij}$. Vertex v_0 represents the depot of all m vehicles. Vertices v_i and v_{i+n} represent the pick and drop off location of customer i respectively and must be visited by the same vehicle in the order of mentioning. Note that multiple pickup locations can be visited consecutively, which means that customers may share a vehicle on (parts of) their ride. Additionally for each customer a time window $[e_i, l_i]$ is defined, which means that both the pickup and the delivery should fall within this interval. Every route should start and end at depot v_0 and the total cost of these routes is minimized. Depending on the application various extra constraints can be defined, for example to ensure that the capacity of the vehicle is satisfied, that the convenience of the customer is above a certain threshold or that the length of one ride does not exceed a predefined upper bound.

The problem we consider in this thesis is only slightly different compared to the problem just defined. After a customer has booked the ride, the time window for this ride is such that $e_i = l_i$, which means there is no flexibility. However while computing the offer there is a time window, which could be a few hours.

3.2. Scheduling algorithms for DARP

This section describes algorithms and their performance that solve variants of DARP, which is defined at the end of the previous section. These algorithms are divided into three categories, namely exact methods, heuristics and metaheuristics. These state of the art methods could possibly also be used to solve our problem.

3.2.1. Exact methods

An exact dynamic programming solution for the Single-Vehicle DARP which runs in $\mathcal{O}(n^2 \cdot 3^n)$ time and needs $\mathcal{O}(n \cdot 3^n)$ space, was introduced by Psaraftis [52]. The algorithm minimizes a single-objective function, which is the weighted sum of total ride time and customer dissatisfactions, which is expressed as the weighted sum of customer waiting and riding time. This algorithm uses a state vector that keeps track of the vehicle's position and for each customer one of the three states, namely not been pickup yet, currently in the vehicle and already dropped off. Before a state's objective function is evaluated constraints like vehicle capacity are checked. Experimental results show that the runtime increased with the capacity of the vehicle. Only problems up to nine customers were solved, which took about ten minutes. Later Psaraftis [54] proposed an algorithm for Single-Vehicle DARPTW that is based on the previously mentioned algorithm and has the same time and space complexity.

Cordeau [13] proposed a Branch-and-Cut algorithm for Multi-Vehicle DARPTW that outperformed CPLEX [3] and solved problems with up to 30 customers. The authors claim that the algorithm is fast enough to optimize routes found by heuristic methods containing hundreds or thousands of customers. The Branch-and-Cut algorithm used several preprocessing techniques and new inequalities. During preprocessing the time windows are tightened, directed edges from the complete graph are removed when they can't be used in an optimal solution and some variables are fixed to reduce the search space. Examples of directed edges that can be removed include, but are not limited to: edges from pickup locations to the base location and edges from drop off locations to their corresponding pickup location. Some customers are assigned to specific vehicles after incompatible customer pairs are identified.

3.2.2. Heuristics

For the Single-Vehicle DARP Psaraftis introduced a 4-approximation algorithm [53] that runs in $\mathcal{O}(n^2)$ time and assumes an undirected graph as input. This means that the algorithm always finds a solution with a total cost not more than four times away of the optimal solution. The algorithm starts with computing a Travelling Salesman tour on the graph with $2n$ vertices without the depot vertex. For computing a Travelling Salesman tour T_0 any heuristic [24, 31] can be used, therefore the algorithm could be easily changed to a 3-approximation algorithm that runs in $\mathcal{O}(n^3)$ time. In the second step of the algorithm, a DARP solution T_1 is constructed by starting at any of the n pickup locations p_i and going clockwise over T_0 and adding a vertex v_j if v_j is not added to T_1 and in the case v_j is a drop off location its corresponding pickup location should already be in T_1. The next three operations can find a better T_1 by one swapping any two locations in T_1 that does not violate the constraints defined by DARP, by two constructing a new T_1 by going counter-clockwise over T_0 and three starting with a different vertex p_i Experiments show that the average performance is well within the performance guarantee, no experiments have been conducted on the version with an approximation ratio of 3.

For the Static Multi-Vehicle DARPTW an insertion algorithm is proposed by Jaw [28], which sorts the rides either on earliest pickup time or latest arrival time. Another insertion algorithm was proposed by Madsen [37], which first sorts all jobs according to expected difficulty to schedule. This difficulty is based on the size of the time window, maximal allowable travel time and special wishes (e.g. number of seats or accessible with a wheelchair). Moreover the algorithm of Madsen is capable of inserting the drivers' breaks into the schedule and solving the dynamic version of the problem, such that it can be used in an online setting. Coslovich [15] introduced another insertion algorithm for the Dynamic Multi-Vehicle DARPTW. The algorithm starts with scheduling all requests known in advance. Whenever a new request comes in during the online phase the algorithm checks if it is possible to add this request to every vehicle. Only if it is possible for a vehicle to do this ride the request is accepted. After this new request is inserted in the schedule, the algorithm tries to improve the schedule until a new request comes in by using a variable neighbourhood search. The neighbourhood is explored by removing two or more edges from the tour and reconnecting the parts in the best way. After a better tour is found the specified constraints, such as customer dissatisfaction, are checked. The experimental results show that the online runtime with up to 50 customers is negligible.

3.2.3. Metaheuristics

Metaheuristics are a popular field of research and have been applied to a wide variety of optimization problems which includes variants of DARP. The term metaheuristic has been defined in various ways by different authors [8, 46, 62, 68]. In short, a metaheuristic is a high level strategy for exploring the solution space, that is not problem specific.

For the Static Multi-Vehicle DARPTW a Tabu Search algorithm (TS) is described by Cordeau [14]. This method is flexible in the sense that it can be easily adapted to deal with more sophisticated objective functions and can be adapted to handle multiple vehicle types and depots. The TS is allowed to explore infeasible solutions and uses a diversification strategy to prevent getting stuck in local optima.

Other metaheuristics have been combined with a heuristic called cluster-first route-second, which has been applied to many routing problems [9, 45]. This approach first clusters jobs based on location and time window and creates for each cluster a route. Once the jobs have been clustered the routing problem has become significantly smaller and easier to solve. Every cluster of jobs is assigned to one vehicle. A similar and less popular approach called route-first cluster-second creates one route and breaks this route down to satisfy constraints like time windows and capacities. Baugh et al. used a Simulated Annealing Algorithm for the clustering part and a space-time nearest neighbour heuristic for the routing part. Other better routing heuristic exists, such as the Lin-Kernighan algorithm, but cost more computation time which is not worth for routing 10 or 20 jobs [6]. Jorgensen et al. used a Genetic Algorithm (GA) for the clustering part and a modified version of the space-time nearest-neighbour heuristic [55]. Every individual in the population is a boolean matrix, with a column for every customer and depot and a row for every vehicle. The typical phases for a GA are defined is such a way that feasible solutions can be found. The steps of a GA include the selection phase, the recombination phase with a crossover operator, the mutation phase and fitness evaluation.

Only the methods for the dynamic setting seem to be applicable for our problem. These methods only include an insertion heuristic combined or not combined with an offline improvement phase in between incoming requests.

3.3. Multi-objective optimization

The solutions of our problem defined in chapter 2 have two features, namely the cost of the taxi ride and the offset to the requested departure time. These two features need to be minimized in dementedly, so it is a multi-objective optimization (MOO) problem. With MOO the goal is to minimize or maximize multiple conflicting objective functions. Problems with these conflicting objective functions appear natural in the decision making and design processes. In this section we assume, without loss of generality, to minimize all

$m \geq 2$ objective functions $f_i(x)$ with $i \in \{1,2,...,m\}$. Since there are multiple competing objective functions there is often no single best solution but a set of optimal solutions or non-dominated set. A solution x is said to dominate another solution x' (denoted by $x \succ x'$), when $f_i(x) \leq f_i(x')$ holds for all $i \in \{1,2,...,m\}$ and $f_i(x) < f_i(x')$ holds at least once. A solution x is called optimal or non-dominated, when there is no other solution x' such that $x' \succ x$. Goldberg [23] introduced the notion of non-dominated levels, to sort n solutions and group them in 1 to n sets of solutions. For each two levels i and j $(i < j)$ the corresponding sets of solutions L_i and L_j have the following property: $\forall x \in L_i, x' \in L_j : \ x \succ x'$. The remainder of this section describes algorithms for solving MOO problems and methods of comparing the quality of solutions of MOO problems.

3.3.1. Algorithms

MOO problems are often solved with population based metaheuristics, such as evolutionary algorithms [21, 72] and particle swarm optimization algorithms [4, 12], but can also be solved with other types of metaheuristics, for example tabu search [25]. With population based algorithms it is important to maintain diversity in the population. For genetic algorithms this can be done with parameter-space and function-space niching, depending on the problem one performs better than the other [16]. Parameter-space niching results in a lot of different parameters, but can have similar objective values, because solutions with similar parameters are more likely to get removed from the population. On the contrary, function-space niching results in a lot of different objective values, but can have similar parameters. The latter seems more promising in the use case of selecting solutions to present to the customer.

3.3.2. Metrics for performance evaluation

In contrast to the evaluation of single-objective optimization (SOO) algorithms, it is not straightforward to evaluate the performance of MOO algorithms. Moreover, literature does not describe one single best method to do such an evaluation. This evaluation is not an easy task, because we have to measure the quality of a set of solutions instead of just one solution. A non-dominated set itself has multiple characteristics that should be optimized. Namely, the distance to the optimal non-dominated set should be minimized, the solutions should be well distributed (for example uniform) and for each objective function a wide variety of values should be covered.

Esben and Kuh [19] defined a metric that computes a single value for any given set of non-dominated solutions, which makes it just as easy to compare algorithms as it would be for an SOO problem. Note that this only captures the distance to the optimal non-dominated set and does nothing with the other two characteristics described earlier. The quality metric $\mathcal{Q}(X)$ for non-dominated set X is formally defined in equation 3.1, where $f'_i(x)$ is a normalized objective function $f_i(x)$, $w_i \in [0,1]$ is the weighting factor, E is the expected value over all possible user preference configurations $w \in W$. Note that computing this metric could become computationally intensive for MOO problems with a high number of objective functions and that in theory W should be infinitely large.

$$\mathcal{Q}(X) = \underset{w \in W}{E}\left[\min_{x \in X}\left\{\sum_{i=1}^{m} w_i \cdot f'_i(x)\right\}\right] \tag{3.1}$$

In order to capture all three characteristics into metrics Zitzler et al. defined three metrics $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$, that measure the average distance to the optimal solution with a distance metric d, the distribution and the extent of set X respectively and are defined in equation 3.2 [72]. Where $\bar{X}$ is the non-dominated set found by an optimal algorithm, σ the size of a neighbourhood. For $\mathcal{M}_1 \geq 0$ it holds that smaller is better, for $\mathcal{M}_2 \in [0, |X|]$ it holds that larger is better distributed and $\mathcal{M}_3$ should also be as large as possible.

$$\mathcal{M}_1(X, \bar{X}) = \frac{1}{|X|}\sum_{x \in X}\min_{\bar{x} \in \bar{X}}\{d(x, \bar{x})\} \tag{3.2a}$$

$$\mathcal{M}_2(X, \sigma) = \frac{1}{|X|-1}\sum_{x \in X}\left|\left\{x' \in X \mid d(x, x') > \sigma\right\}\right| \tag{3.2b}$$

$$\mathcal{M}_3(X) = \sqrt{\sum_{i=1}^{m}\max_{x, x' \in X}\left\{d(x_i, x'_i)\right\}} \tag{3.2c}$$

Zitzler et al. [72] also defined two other metrics to compare the performance of MOO algorithms for the multi-objective knapsack problem. Note that with this problem all objective functions are maximized and all $f_i(x) \geq 0$. The first metric $\mathcal{S}(X)$ measures the space covered by a non-dominated set X. For the 2-dimensional case one solution x covers the rectangular area between points $(0,0)$ and $(f_1(x), f_2(x))$, this works well because of the structure of the problem. However this metric can easily be adapted to be useful for minimization problems. The second metric $\mathcal{C}(X, X') \in [0,1]$ compares two non-dominated sets X and X' by counting the number of times X' gets dominated by a solution in X, which is defined in equation 3.3. When all solutions in X' are dominated by a solution in X, then $\mathcal{C}(X, X') = 1$. Note that in general the statement $\mathcal{C}(X, X') = 1 - \mathcal{C}(X', X)$ is false, therefore both values are interesting while comparing two algorithms.

$$\mathcal{C}(X, X') = \frac{\left|\{x' \in X' \mid \exists x \in X : x \succeq x'\}\right|}{|X'|} \tag{3.3}$$

The multi-objective algorithms described in literature are mainly evolutionary algorithms which are not useful for a real-time setting. On the contrary the metrics used to evaluate these algorithms are useful to evaluate the non-exact algorithm we propose in the next chapter.

3.4. Taxi demand prediction

This section gives an overview of the working of methods developed for taxi demand prediction and the use cases for these methods. The authors of multiple papers had a common goal, namely to prevent taxi divers from roaming for new customers to save cost. Thus this thesis and these authors have as goal to increase the effective utilization of taxis.

Phithakkitnukoon et al. [49] described a method to predict the number of vacant taxis based on historical data for 1×1 kilometre cells in the city of Lisbon, Portugal. The predictive model uses the time of the day, the day of the week and the weather condition as help to predict more accurately. The conditional probability of y vacant taxis given the time of the day T, the day of the week D and the weather condition W is stated in equation 3.4.

$$P(Y = y \mid T, D, W) = \frac{P(Y = y) \cdot P(T, D, W \mid Y = y)}{P(T, D, W)} \tag{3.4}$$

Also Chang et al. [11] used time of the day, the day of the week and the weather condition in their algorithm to predict taxi demand hotspots. However they concluded that these features are not enough, because events like a musical performance, affect the distribution of taxi requests significantly. The authors also compared three clustering algorithms, namely k-means, agglomerative hierarchical clustering and DBSCAN *(see section 3.5)*, to cluster the customer's pickup and drop off locations in six cities in the northern part of Taiwan. They cluster these locations, such that they can map coordinates to buildings or city blocks. However the authors did not draw a conclusion, because none of them were the absolute best and all three algorithms had their advantages and disadvantages.

Moreira-Matias discussed several models [43] to predict taxi demand with an online algorithm. One of them assumed the probability to have n taxi requests within a determined time period to be a Poisson distribution with a time dependent rate $\lambda(t)$. The calculation of $\lambda(t)$ was based on historical data and recent information gathered during the execution of the algorithm. The weighted variant of this model attached more weight to recent information, in this case $\lambda(t)$ is calculated by the exponential moving average of the last α observations. Another model Moreira-Matias proposed for taxi demand prediction is the ARIMA model, which is able to update itself and makes less assumptions such as periodicity.

The same features discussed in this section can also be used with the calculation of the expected cost of a taxi ride. However the methods to predict taxi demand are not applicable to predict taxi demand.

3.5. Methods for clustering data

Since both scheduling and taxi demand prediction algorithms heavily depend on various clustering methods, this section describes a few of them. In the world of data analysis clustering algorithms are used to do unsupervised classification of observations into groups (clusters) [27]. In this section n denotes the number of observations that will be divided in clusters. Each observation is represented as p dimensional feature vector x. There are different kinds of cluster algorithms used for different kinds of datasets. To be able to learn how taxi rides are distributed over the year a cluster algorithm is needed. The available features of a taxi ride are: pick up location, a drop off location, a pick-up time and a drop off time. Note that a location itself consists of two features namely the longitude and the latitude.

Some algorithms are only capable of dealing with two dimensional data, e.g. a location. Other algorithms are capable of dealing with higher dimensional data, e.g. multiple locations with a time and a date. The survey of Xu and Wunsch [70] lists for different kinds of clustering algorithms some examples. For numerous algorithms the number of clusters need to be defined before hand, e.g. k-means. This section describes clustering algorithms that have been used to predict taxi demand in the past or seems promising. Methods for choosing a useful number of clusters are described in subsection 3.5.6. Methods for computing the similarity between two objects are described in subsection 3.5.1. Clustering sequential data, e.g. sound waves, is not discussed since this method cannot be applied to taxi rides.

3.5.1. Methods for computing similarity

Before we can divide data data in clusters we should be able to say something about the similarity of observations. Different metrics have been defined for computing the distance or similarity of observations, the most popular metric for numerical features is the Euclidean distance, which is defined in equation 3.5. Other well-known metrics include the Manhatten distance and cosine similarity, which are defined in equation 3.6 and 3.7 respectively. The Manhatten distance, also known as the city block distance, and Euclidean distance are sensitive to overweight the largest-scaled feature [27]. To overcome this drawback the data could be normalized first. The cosine similarity naturally has the property that it ignores the length of the feature vectors, because it computes the angle between feature vectors. Cosine similarity is useful when one would like to classify texts while ignoring the length of the text. When dealing with coordinates the haversine distance could be used. This metric accounts for the spherical property of the earth and is defined in equation 3.8, where ϕ_a the latitude of location a, λ_a the longitude of location a and r the radius of the earth.

$$d(a,b) = \sqrt{\textstyle\sum_{i=1}^{p} (a_i - b_i)^2} \tag{3.5}$$

$$d(a,b) = \sum_{i=1}^{p} |a_i - b_i| \tag{3.6}$$

$$d(a,b) = \cos(\theta) = \frac{a \cdot b}{\|a\| \, \|b\|} = \frac{\sum_{i=1}^{p} a_i b_i}{\sqrt{\sum_{i=1}^{p} a_i^2}\sqrt{\sum_{i=1}^{p} b_i^2}} \tag{3.7}$$

$$d(a,b) = 2r \cdot \arcsin\sqrt{\sin^2\left(\frac{\phi_a - \phi_b}{2}\right) + \cos(\phi_a)\cos(\phi_b)\sin^2\left(\frac{\lambda_a - \lambda_b}{2}\right)} \tag{3.8}$$

3.5.2. Hierarchical clustering

Hierarchical cluster algorithms organize the data in a tree structure, called a dendrogram, where the nodes represents subsets of the whole dataset and the leafs represent an observation. This tree structure can be build bottom-up (agglomerative) or top-down (divisive). Different clustering assignments can be retrieved from this tree by cutting it at a certain height. The divisive approach is not commonly used in practice [70], therefore only the agglomerative approach is described in more detail. A simple agglomerative hierarchical clustering algorithm can be described with the pseudo code in algorithm 1. This algorithm cost $\mathcal{O}(n^2 \log(n))$ time and $\mathcal{O}(n^2)$ space, if a sorted data structure for the distances is used.

All agglomerative hierarchical cluster algorithms require to compute distance between two clusters. Computing the distance between a newly formed cluster $C_1 \cup C_2$ and another cluster C_3. This could be done in several ways [44], e.g. the single link method which uses equation 3.9.

$$d(C_1 \cup C_2, C_3) = \min(d(C_1, C_3), d(C_2, C_3)) \tag{3.9}$$

Algorithm 1: General agglomerative hierarchical clustering algorithm [44].

```
Create $n$ clusters with each one observation
Compute all pair-wise distances between the clusters
repeat
    Merge the two closest clusters
    Recompute distances between clusters
until one cluster remains
```

3.5.3. Squared error-based clustering

The goal of the k-means clustering problem is find a clustering assignment $\{C_1, C_2, ..., C_k\}$ such that each x_j for $1 \le j \le n$ is in exactly one cluster and the sum of the squared error over all $1 < k < n$ clusters is minimal [26]. In equation 3.10 the objective function for the k-means clustering problem is stated formally. In this equation μ_i is a p dimensional vector that represents the mean of cluster C_i. This clustering problem has been proven to be $\mathcal{NP}$-hard [18, 38].

$$f(\{C_1, C_2, ..., C_k\}, \mu) = \sum_{i=1}^{k} \sum_{x \in C_i} \left\| x - \mu_i \right\|^2 \tag{3.10}$$

The k-means heuristic algorithm [36] is a well-known and easy to understand algorithm with a time complexity of $\mathcal{O}(nkp)$. The algorithm starts with clustering the data in k clusters based on randomly chosen means. Then the centroid of each cluster is computed, which becomes the new mean. Finally clustering based on the new means and computing new means is repeated a few times until the algorithm converged to a local optimum and the data points do not switch cluster. A major disadvantage of the k-means algorithm is that it is sensitive to outliers.

3.5.4. Fuzzy clustering

With fuzzy clustering, in contrast to hierarchical clustering and squared error-based clustering, can an observation be partially assigned to multiple clusters. When $J(U, \mu)$ in equation 3.11 is minimal an optimal fuzzy clustering with c clusters is found. A fuzzy clustering can be described by a membership matrix U of size $c \times n$ where $u_{ij} \in [0, 1]$ describes the degree of membership of x_j in C_i. The distance measurement function d, the number of clusters $1 < c < n$ and the weighting exponent $1 \le m$ should be selected beforehand. Selecting m can be done experimentally, but for most data $1.5 \le m \le 3.0$ gives good results [7].

$$J(U, \mu) = \sum_{i=1}^{c} \sum_{j=1}^{n} (u_{ij})^m d(x_j, \mu_i) \tag{3.11}$$

The fuzzy c-means (FCM) algorithm [7] starts by fixing c, m, $d(\cdot)$ and a small positive number ε. In the first round the means μ^0 are selected randomly. In each next round $t \ge 0$ of the algorithm the membership matrix $U^{(t)}$ is computed with 3.12 and new means $\mu^{(t+1)}$ are computed with equation 3.13 and the stop condition $\left\| \mu^{(t+1)} - \mu^{(t)} \right\| < \varepsilon$ is checked [70]. Just as k-means is FCM sensitive to outliers.

$$u_{ij} = \left(\sum_{k=1}^{c} \left(\frac{d(x_j, \mu_i)}{d(x_j, \mu_k)} \right)^{2/(m-1)} \right)^{-1} \tag{3.12}$$

$$\mu_i = \sum_{k=1}^{n} (u_{ik})^m \cdot x_k \Bigg/ \sum_{k=1}^{n} (u_{ik})^m \tag{3.13}$$

3.5.5. Density based clustering

Density based clustering algorithms tend to ignore outliers and are therefore able to cope well with data sets with outliers. DBSCAN [20] is able to find clusters of arbitrary shape and relies on the density based notion of clusters. This algorithm iterates over a list of unclassified points and tries to find neighbours that are reachable without violating the density constraint. Efficiently finding neighbours is done by using a R^*-tree. DBSCAN costs $\mathcal{O}(n\log(n))$ time and space and can produce clusters of any shape, which is depending on the application a advantage or a disadvantage.

3.5.6. Choosing the number of clusters

Numerous algorithms have as parameter the desired number of clusters k, which greatly influences the meaningfulness of the outcome. For some applications a person can, because of his domain knowledge, manually determine a k. If this is not an option we should look for methods that can automatically determine a useful number of clusters. A simple rule of thumb $k \approx \sqrt{n/2}$ [39] could be used, but more advanced methods usually work better. This section describes a few methods that find a *good* balance between a complicated model and loss of information, which corresponds with too many and too few clusters respectively.

Salvador and Chan proposed an efficient algorithm that is called the L Method which finds the point of maximum curvature in the graph of 'number of clusters' versus 'a clustering evaluation metric' [57]. The L Method tries to fit two straight lines on this graph. Using only a part of this graph usually works better, the best part is found in an iterative process by applying the L Method. This method is can be efficiently be combined with a hierarchical clustering algorithm, because with the dendrogram different clusterings with different number of clusters can be easily generated.

It turned out that clustering algorihms were not needed to calculate the expected cost of a taxi ride. However the similarity functions defined in this section are needed with regression analysis and some interpolation methods which are needed to compute this expected cost in chapter 5.

3.6. Regression Analysis

The purpose of regression analysis is to construct mathematical models which describe the relationship between variables [58]. This is exactly what is needed to compute the expected cost of a taxi ride, because this based on the relationship between the various features, such as time and location, and the cost of the ride. This section first describes the most simple form of regression analysis, continues with more advanced methods and concludes with the favourable and unfavourable properties of the methods described.

A simple linear regression model fits a straight line through a set of data points. To define the best fit the mean squared error (MSE) is often used, but also other loss functions can be used. When the relation between the predictor, also called independent variable, and the predictive, also called dependent variable, does not describe a linear relation other methods should be used. Polynomial regression, a special case of multiple linear regression which allows more predictive variables, considers the terms of the polynomial independent. Such a polynomial is stated in 3.14, where the problem is to find the parameters α, α_1, α_2 and α_3 that describes the relation between the dependent variable y and the predictors x, x^2 and x^3 best. The best parameters can be found by minimizing the loss function with an optimization algorithm. Examples of popular optimization algorithms used for regression analysis include several non-exact algorithms, such as Gradient descent and Levenberg-Marquardt algorithm [32]. Most of these algorithms need to be initialized with a guess such that it can find a nearby local minimum.

$$\hat{y} = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \alpha_3 x^3 \tag{3.14}$$

A new problem arises while using polynomial regression, namely what degree of polynomial is used to fit the data. A too low degree polynomial can cause underfitting and a too high degree polynomial can cause overfitting. Both underfitting and overfitting is undesirable, because the true relation between the variables remains unclear. With underfitting the regression function is not able to fit the data and the loss value computed by the loss function stays relatively high. With overfitting the loss value is extremely low and fits the data well, but an overfitted function fails to generalize to other data sets. Cross-validation is a popular way to check if the model generalizes well to other data sets. With cross-validation the data set is divided in a training and a validation set. The training set is used to learn the parameters and the validation set is used to validate the selected hyperparameters. The degree of the regression function is an example of a hyperparameter. Different forms of cross-validation have different ways of splitting up the data. For example leave-one-out cross-validation uses only one observation for validation and the rest of the data is used for training. The average over all possible ways of splitting up the data can be evaluated. Another well-known form is k-fold cross-validation which randomly partitions the data in k subsets, then 1 subset is used for evaluation and the remaining $k-1$ subsets are used for training. Another approach to prevent overfitting is to use Tikhonov regularization, also known as ridge regression, which favours solutions with smaller values for α_j. This is done by using another loss function, for example the one stated in equation 3.15, where each measurement y_i is compared with the predicted value $\hat{y}_i$. The parameter λ can be selected empirically in combination with cross-validation. Note that with $\lambda = 0$ the loss function in equation 3.15 is the same as MSE.

$$\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda ||\alpha|| \tag{3.15}$$

When its hard to select a regression function one can use a Multilayer Perceptron (MLP), which is directed graph where each layer is fully connected with the next. The first layer is fed the features, the last layer outputs the predicted value(s) and in between one or more hidden layers connect the two. Each node receives a number of weighted values, sums them up, feed that to an activation function and passes on the value to all connected nodes. Learning a neural network comes down to learning the weights which minimizes a loss function. This is done by backpropogation in combination with an optimization algorithm which is usually Stochastic Gradient Descent, Adam [30] or L-BFGS [33]. The so-called hyperparameters of the MLP are the number of hidden layers, the number of nodes within each layer, the loss function, the activation function, the initialization of the weights and the learning rate of the algorithm. Selecting the hyperparameters is a non-trivial task and can be done manually or with coordinate descent, grid search, random search and Bayesian optimization [61].

Nonparametric regression can also be used when its hard to select a regression function, this type of regression does not use a predefined global model. Kernel regression is a nonparametric regression which estimates a continuous dependent variable by convolving a kernel function for each of the data points' locations. Another example is nonparametric multiplicative regression (NPMR). NPMR comes in different flavours, but they have in common that they all use a predefined local model and a kernel function which is used to define how local is local, i.e. it sets a weight for the measurements before optimizing the local model. This local model can be a weighted average, linear regression or logistic regression [40].

Although some methods are easier to use than other, all regression analysis methods described in this section are potentially methods useful for the cost prediction of rides. In addition cross-validation is used to validate and evaluate the methods in chapter 6.

3.7. Interpolation Methods

Regression analysis makes it possible to construct a model that describes the relationship between variables, which generalizes well over similar data sets. Most methods require a predefined model and the algorithm must find the parameters or weights of this model. As seen in the previous section, selecting a model is sometimes difficult. With interpolation one tries to accurately predict the value of an unsampled point by a set of sampled points in d-dimensional space. Most interpolation methods are easier to apply than regression analysis, but can still give a prediction of the complete feature space if enough data is provided. Interpolation for multiple dimensions becomes harder especially for an irregular grid, but methods do exist. Since only spatial interpolation methods are used in this thesis only this kind of interpolation methods are discussed in the rest of this section. These are nearest neighbours, natural neighbours, inverse distance weighting, linear interpolation, polynomial interpolation. More interpolation methods used for spatial analysis, such as Kriging and spline interpolation, are discussed with examples and applications by Mitas and Mitasova [42].

Nearest neighbours interpolation is arguably the most simple interpolation method. When the value at some point needs to be predicted it simply predicts the value for the nearest point the value is known. This has a worst case time complexity of $\mathcal{O}(n)$, but an average time complexity of $\mathcal{O}(\log n)$ for n known points with for example a k-d tree. The nearest neighbours interpolation method results in Voronoi tessellation. A related method is the natural neighbours interpolation [60], which make use of Voronoi tessellation of the observations. The prediction $u(x)$ for the unsampled point x is a weighted average of m neighbouring observations. The weights w_i is the area stolen from observation x_i after inserting x in the Voronoi tessellation. Note that

u_i is the value at point x_i.

$$u(x) = \sum_{i=1}^{m} w_i u_i \Bigg/ \sum_{i=1}^{m} w_i \tag{3.16}$$

Inverse Distance Weighting (IDW) [59], defined in equation 3.17, is a weighted average over all N observations to predict the value $u(x)$ at an unsampled point x. It is assumed that all points have an influence on the value of unsampled points, but this decays with an increasing distance. When x is being predicted, the weight $w_i(x)$ of observation x_i is defined by equation 3.18, which makes use of distance metric $d(\cdot,\cdot)$ and a decaying parameter p. A higher value for p weights nearby samples even more, than distant observations. The parameter p is usually set to a value between 1 and 5, but this depends on the application of IDW and should be selected empirically. The runtime complexity for predicting one value is $\mathcal{O}(n)$, which can be computationally to intensive for some applications. This can be reduced to an average time complexity of $\mathcal{O}(m \log n)$ if only m nearby observations are considered and a efficient spatial search algorithm like k-d tree is used.

$$u(x) = \sum_{i=1}^{n} w_i(x) u_i \Bigg/ \sum_{i=1}^{n} w_i(x) \tag{3.17}$$

$$w_i(x) = \frac{1}{d(x, x_i)^p} \tag{3.18}$$

More recently, Lu and Wong [34] proposed the Adaptive Inverse Distance Weighting (AIDW) method. This method, in contrast to IDW, does not select a value for p a priori, but computes one based on the density of samples near the unsampled point. The authors show that their method performs better on two examples compared to IDW.

Linear interpolation for multiple dimensions uses a triangulated irregular network and computes for each triangle a bivariate function [42]. This method will clearly not produce a smooth landscape, but this is sometimes desirable. Polynomial interpolation methods for multiple dimensions use first-order or both first- and second-order derivatives. Akima [5] proposed a method to use a fifth-degree polynomial as interpolating function for each triangle.

All interpolation methods described in this section are applicable for cost prediction. However the question is whether or not the smoothing some methods apply is favourable, which is answered in chapter 6.

4 Finding the best offers

The algorithms in this chapter provide an answer to the question: “How can an online algorithm efficiently compute a non-dominated set of offers in reaction to a taxi ride request, given a large list of already accepted taxi rides, within a small enough period of time?” By combining the newly requested taxi ride with one or multiple already accepted taxi rides we can reduce the cost of the new taxi ride request. This cost reduction can be passed on to the customer in a nice offer. A list of the best offers could, for example, be presented to the customer via a website or mobile phone application.

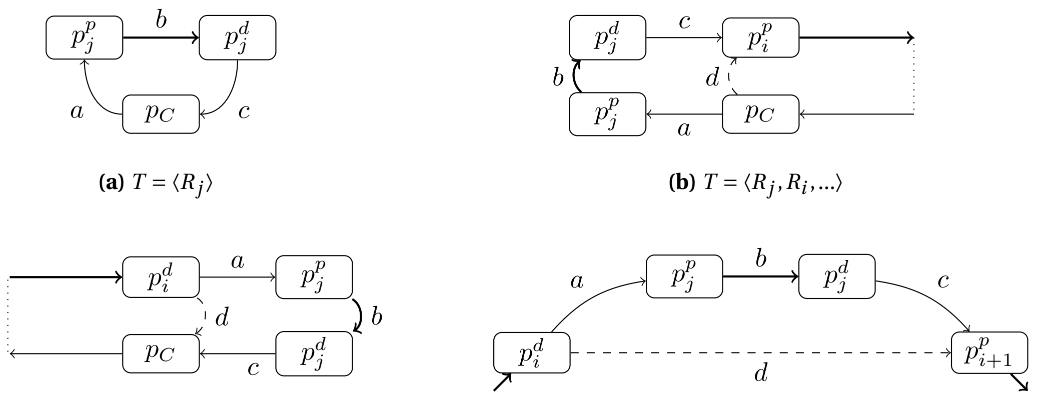
Recall from chapter 2, that the cost for a new incoming taxi ride request is defined by the cost for performing all taxi rides with the new taxi ride minus the cost for doing all taxi rides without the new taxi ride. An optimal algorithm should compute an optimal schedule for both scenarios. As stated in the problem description the algorithm should be able to produce an answer within a second for as large as possible problem sizes and an even quicker result will be favourable, because of the application of the algorithm. As stated in chapter 3, solving a MVDARPTW is proven to be $\mathcal{NP}$-hard. Due to this $\mathcal{NP}$-hardness of the problem, the available computation time and the way companies do their scheduling it is justified to limit ourselves to an insertion heuristic. Generally these kind of heuristics give poor results on scheduling problems, especially when applied on many jobs in succession. However the scheduling is used for an estimation of the costs and a conservative cost estimation is even a positive property. Additionally it is inconvenient for the customer to constantly reschedule the ride therefore these rides may only be exchanged between taxis from the same company. Both the real world and the insertion heuristic imply that whenever a taxi ride is sold to a customer it is fixed in the schedule.

The first section gives a mathematical explanation how to obtain the objective values, that is the cost and the waiting time, when a taxi ride is inserted in a certain place in the schedule. The second section describes a naive algorithm that evaluates all possibilities. The two subsequent sections each describe an algorithm that is an improvement upon the naive algorithm with regard to the runtime. This chapter concludes with a section about niching methods, which can be used to reduce the number of solutions.

4.1. Cost for inserting a taxi ride

This section describes how to compute the cost for a taxi ride and the waiting time for the customer. Recall from chapter 2 that the waiting time is defined as the offset to the requested departure time. Since we limit ourselves to an insertion heuristic, it is not needed to evaluate the cost of the complete schedule and only evaluating this local change is a lot more efficient. This section describes examples and formulas to evaluate the local change in the schedule. This provides good insight into what is required to implement any algorithm to find the list of best offers.

Recall from chapter 2 the notation to denote the location of the home depot p_C of a taxi company C and the pickup location p_i^p, the drop off location p_i^d, the pickup time t_i^p and the drop off time t_i^d for a taxi ride R_i. In figure 4.1 all four possible scenarios are visualized, namely the new ride is either the first for a taxi in figure 4.1a, scheduled before all other rides in figure 4.1b, scheduled after all other rides in figure 4.1c or scheduled between two other rides in 4.1d. In these figures a thick line indicates a part of a route where the taxi seats a passenger, a dashed line indicates a part of the route the taxi would have driven when ride R_j was not inserted and a normal line indicates a part of the route the taxi is empty. In the example illustrated by figure 4.1a the cost of R_j is the cost of driving parts a, b and c. In the example illustrated by figures 4.1b, 4.1c and 4.1d the cost of R_j is the cost of driving parts a, b and c minus part d. Note that if the taxi driver has to wait at location p_j^p the cost for R_j does not increase, because the taxi driver is already paid for this time by the customer of ride R_{i+1}. However a customer must pay for the waiting time when this ride is appended at the beginning or end of an existing schedule.



(a) $T = \langle R_j \rangle$ **(b)** $T = \langle R_j, R_i, ... \rangle$ **(c)** $T = \langle ..., R_i, R_j \rangle$ **(d)** $T = \langle ..., R_i, R_j, R_{i+1}, ... \rangle$

Figure 4.1: Insertion of a taxi ride R_j in an existing schedule $T' = T \setminus \{R_j\}$

Equation 4.1 defines $f'_{cost}(C, T, R_j)$ and expresses the cost of a new taxi ride R_j driven by taxi T working for company C with T the taxi's new schedule already including R_j. The equation evaluates the increase in time and the increase in distance separately. The increase of distance, defined by $f'^{d}_{cost}(C, T, R_j)$, subtracts the length of the old route from the length of the new route. The increase of time, defined by $f'^{t}_{cost}(C, T, R_j)$, subtracts the time needed for the old route from the time needed for the new route and adds the waiting time for the driver. Recall that C_d denotes the cost per driven kilometre and C_t denotes the cost per minute. Note that T is sorted on ascending order of pickup time. Given a schedule $T = \langle R_1, R_2, ..., R_{|T|} \rangle$ the inequality $t_1^p < t_1^d < t_2^p < t_2^d < ... < t_{|T|}^p < t_{|T|}^d$ should hold, else the schedule is infeasible. The schedule is also infeasible when a taxi cannot drive from a drop off location to the next pickup location in time, thus there exists an $1 \leq i < |T|$ such that the inequality $t_i^d + f_t(\langle p_i^d, p_{i+1}^p \rangle) > t_{i+1}^p$ holds.

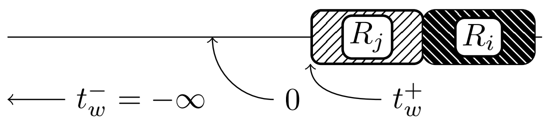
$$f'_{cost}(C, T, R_j) = \begin{cases} \infty & \text{if } T \text{ is infeasible} \\ C_d \cdot f'^{d}_{cost}(C, T, R_j) + C_t \cdot f'^{t}_{cost}(C, T, R_j) & \text{otherwise} \end{cases} \tag{4.1}$$

$$f'^{d}_{cost}(C, T, R_j) = \begin{cases} f_d(\langle p_C, p_j^p, p_j^d, p_C \rangle) & \text{if } T = \langle R_j \rangle \\ f_d(\langle p_C, p_j^p, p_j^d, p_i^p \rangle) - f_d(\langle p_C, p_i^p \rangle) & \text{if } T = \langle R_j, R_i, ... \rangle \\ f_d(\langle p_i^d, p_j^p, p_j^d, p_C \rangle) - f_d(\langle p_i^d, p_C \rangle) & \text{if } T = \langle ..., R_i, R_j \rangle \\ f_d(\langle p_i^d, p_j^p, p_j^d, p_{i+1}^p \rangle) - f_d(\langle p_i^d, p_{i+1}^p \rangle) & \text{if } T = \langle ..., R_i, R_j, R_{i+1}, ... \rangle \end{cases} \tag{4.2}$$

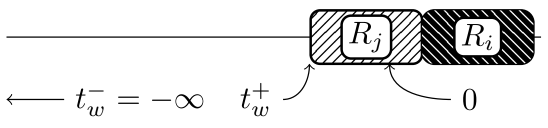
$$f'^{t}_{cost}(C, T, R_j) = \begin{cases} f_t(\langle p_C, p_j^p, p_j^d, p_C \rangle) & \text{if } T = \langle R_j \rangle \\ f_t(\langle p_C, p_j^p \rangle - f_t(\langle p_C, p_i^p \rangle) + t_i^p - t_j^p & \text{if } T = \langle R_j, R_i, ... \rangle \\ f_t(\langle p_j^d, p_C \rangle) - f_t(\langle p_i^d, p_C \rangle) + t_j^d - t_i^d & \text{if } T = \langle ..., R_i, R_j \rangle \\ 0 & \text{if } T = \langle ..., R_i, R_j, R_{i+1}, ... \rangle \end{cases} \tag{4.3}$$

Sometimes an infeasible schedule can be made feasible again by changing the departure time of the newly requested taxi ride. This relaxation of the departure time constraint leads to more possible solutions, and thus more offers a customer can choose from. In some scenarios a lower cost can also be achieved by changing the departure time of a ride. For example it might be that if the same ride, but a year later is less costly than it would be today. Thus if we completely remove the departure time constraint we must evaluate possibilities in the distant future. The chance a customer is interested in an offer, with a large waiting time, is virtually nil. Thus we should define some time window to limit the search space.

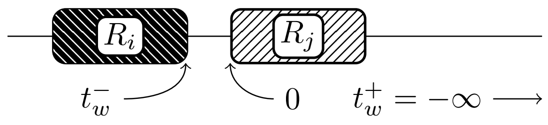
We need to allow the customer to change the departure time of his taxi ride to influence the cost of it. A time window in which this departure time can be moved is denoted by $\hat{t}_w = [t_w^-, t_w^+]$. With t_w^- and t_w^+ the offset to the requested departure time. Since customers are not interested in offers where both the cost and the time between the requested and actual departure time is higher than another offer we also introduce $\hat{t}'_w \subseteq \hat{t}_w$. The interval $\hat{t}'_w$ contains all waiting times for which one offer is not strictly better, i.e. there exists no $t_1, t_2 \in t'_w$ such that $(C, R_j, t_1) \succ (C, R_j, t_2)$. Figure 4.2 shows all possible scenarios when a ride R_j is assigned to a certain taxi T. In all the listed examples $t_p(R_j) = 0$, which is not feasible in figures 4.2b, 4.2d, 4.2f and 4.2h. The interval $\hat{t}_w$ for a taxi ride R_j when assigned to a taxi T, is denoted by $f_{wait}(T, R_j)$ and is defined in equation 4.4. The function $f'_{wait}(T, R_j)$ is defined in equation 4.5 and computes $\hat{t}'_w$.



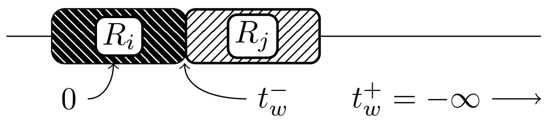
(a) $T = \langle R_j, R_i, \ldots\rangle$ and $f'_{wait}(T, R_j) = [0, t_w^+]$



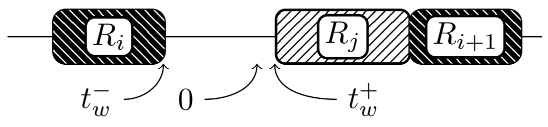
(b) $T = \langle R_j, R_i, \ldots\rangle$ and $f'_{wait}(T, R_j) = [t_w^+, t_w^+]$



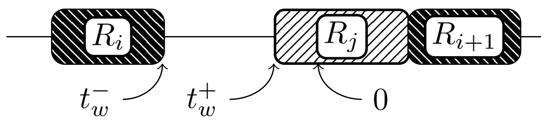
(c) $T = \langle \ldots, R_i, R_j\rangle$ and $f'_{wait}(T, R_j) = [t_w^-, 0]$



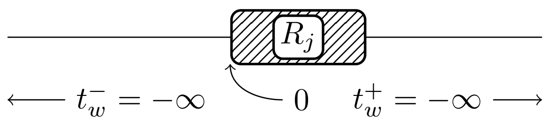
(d) $T = \langle \ldots, R_i, R\rangle$ and $f'_{wait}(T, R_j) = [t_w^-, t_w^-]$



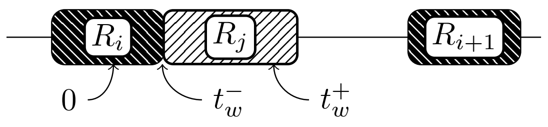
(e) $T = \langle \ldots, R_i, R_j, R_{i+1}, \ldots\rangle$ and $f'_{wait}(T, R_j) = [0, 0]$



(f) $T = \langle \ldots, R_i, R_j, R_{i+1}, \ldots\rangle$ and $f'_{wait}(T, R_j) = [t_w^+, t_w^+]$


R_j
$\longleftarrow t_w^- = -\infty$
0
$t_w^+ = -\infty \longrightarrow$

(g) $T = \langle R_j\rangle$ and $f'_{wait}(T, R_j) = [0, 0]$



(h) $T = \langle \ldots, R_i, R_j, R_{i+1}, \ldots\rangle$ and $f'_{wait}(T, R_j) = [t_w^-, t_w^-]$

Figure 4.2: All possible scenarios to schedule R_j in T

$$f_{wait}(T, R_j) = \begin{cases} (-\infty; +\infty) & \text{if } T = \langle R_j\rangle \\ (-\infty; t_i^p - f_t(\langle p_j^p, p_j^d, p_i^p\rangle)] & \text{if } T = \langle R_j, R_i, \ldots\rangle \\ [t_i^d + f_t(\langle p_i^d, p_j^p\rangle); +\infty) & \text{if } T = \langle \ldots, R_i, R_j\rangle \\ [t_i^d + f_t(\langle p_i^d, p_j^p\rangle); t_{i+1}^p - f_t(\langle p_j^p, p_j^d, p_{i+1}^p\rangle)] & \text{if } T = \langle \ldots, R_i, R_j, R_{i+1}, \ldots\rangle \end{cases} \tag{4.4}$$

$$f'_{wait}(T, R_j) = \begin{cases} \emptyset & \text{if } f_{wait}(T, R) = \emptyset \text{ (when } T \text{ infeasible)} \\ [0, 0] & \text{if } T = \langle R_j\rangle \\ [\min(0, t_w^+); t_w^+] & \text{if } T = \langle R_j, R_i, \ldots\rangle \wedge f_{wait}(T, R_j) = [t_w^-, t_w^+] \\ [t_w^-; \max(0, t_w^-)] & \text{if } T = \langle \ldots, R_i, R_j\rangle \wedge f_{wait}(T, R_j) = [t_w^-, t_w^+] \\ [\max(0, t_w^-); \max(0, t_w^-)] & \text{if } T = \langle \ldots, R_i, R_j, R_{i+1}, \ldots\rangle \wedge f_{wait}(T, R_j) = [t_w^-, t_w^+] \wedge 0 < t_w^+ \\ [\min(0, t_w^+); \min(0, t_w^+)] & \text{if } T = \langle \ldots, R_i, R_j, R_{i+1}, \ldots\rangle \wedge f_{wait}(T, R_j) = [t_w^-, t_w^+] \wedge 0 > t_w^- \end{cases} \tag{4.5}$$

4.2. Naive algorithm

This section describes a straightforward algorithm, which is a good starting point and useful when comparing runtime and the quality of solutions of other algorithms. This naive algorithm uses an insertion heuristic that uses the objective functions f'_{wait} and f'_{cost} defined in section 4.1.

This naive algorithm tries to add the new taxi ride to every taxi possible and computes the non-dominated set. An example of such a non-dominated set is illustrated in figure 4.3, where the non-dominated solutions are marked blue and the rest of the feasible solutions are marked in grey.

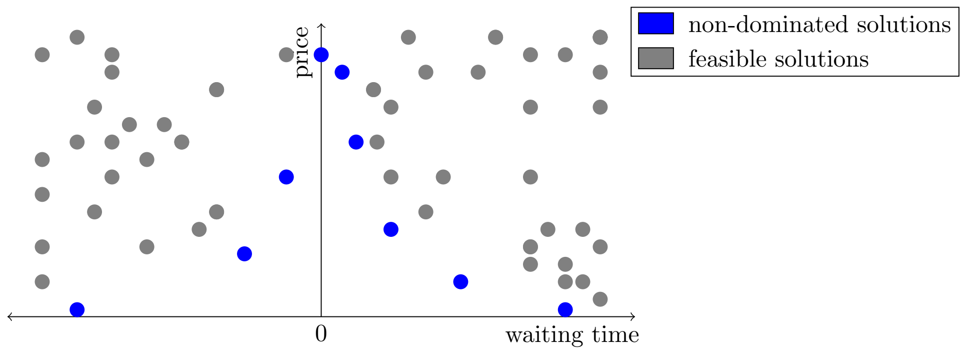


Figure 4.3: Representation of search space

The pseudocode is stated in algorithm 2, where S is the current schedule and R is the taxi ride for which a set of solutions is needed. Schedule S is nothing more than a set of companies that are registered to compete for the best offer. Recall from chapter 2, that a company $C = \{T_1, T_2, ..., T_{|C|}\}$ is a set of taxis and a taxi $T_i = \langle R_1, R_2, ..., R_{|T|}\rangle$ is a list of rides in order of pickup time. When R is scheduled to a certain taxi T it can result in an infinite number of solutions, because if $f'_{wait}(T, R) = [t^-_w, t^+_w]$ with $t^-_w \neq t^+_w$ a customer could be offered every waiting time $t_w \in [t^-_w, t^+_w]$. We assume that the customer is only interested in one of the extremes. In this way every gap in the schedule can result in zero, one or two feasible solutions. In the pseudocode the set of feasible solutions is denoted with X, where $0 \leq |X| \leq 2(|T|+1)$. Then every $x \in X$ is added to the non-dominated set Σ if and only if x is not being dominated by one of the solutions in Σ. Next, all solutions $x^* \in \Sigma$ that are dominated by x are removed from Σ. Note that solutions with a waiting time $t_w < 0$ cannot dominate solutions with a waiting time $t'_w > 0$ and vice versa. If n is the number of taxi rides, the time complexity of the FINDOFFERS procedure is $\mathcal{O}(n^2)$, because in the worst-case scenario all n solutions are non-dominated and $\mathcal{O}(n^2)$ comparisons are needed to verify this. However in practice, the algorithm's runtime scales linear with the number of taxi rides, because computing the routes takes far out the most time. In section 6.2 the runtime of the algorithm is evaluated empirically.

Algorithm 2: Pseudo code of naive algorithm

```
input  : A schedule $S$ and a taxi ride $R$
output: A non-dominated set $\Sigma$
procedure FindOffers($S$, $R$)
    $\Sigma \leftarrow \emptyset$                                   /* non-dominated set */
    foreach $C_i \in S$ do
        foreach $T_j \in C_i$ do
            $X \leftarrow$ schedule $R$ in $T_j$
            foreach $x \in X$ do
                if $\forall x^* \in \Sigma : x \succeq x^*$ then
                    $\Sigma \leftarrow \{x' \mid x' \in \Sigma \wedge x \not\succ x'\}$
                    $\Sigma \leftarrow \Sigma \cup \{x\}$
                end
            end
        end
    end
    return $\Sigma$
end
```

4.3. Parallel algorithm

Suppose the number of taxi companies and taxis is too large for the naive algorithm to handle in a given amount of time. One way to decrease the computation time is to use a faster computer. However a sequential algorithm like the one described in the previous section cannot benefit from multiple cores, but a parallel algorithm like the one described in this section can. The algorithm introduced in this section is based on the naive algorithm.

The parallel algorithm consists of four main steps. First all n solutions must be computed and stored, which can be done in parallel, because these solutions are completely independent. The second step is to initialize a list D, which stores for every solution whether or not it is dominated by another. In the third step, for every pair of solutions X_i and X_j it is checked if X_j is dominated by X_i and if so D_j is set to 1. In the last step the output list can be composed, because all non-dominated solutions $X_i \in \Sigma$ have $D_i = 0$. This algorithm requires $W(n) = \mathcal{O}(n^2)$ work and $T(n) = \mathcal{O}(n^2/p)$ time for p processors. Thus we need $p = n^2$ processors for a constant time algorithm.

Algorithm 3: Pseudo code of naive parallel algorithm

```
input  : A schedule $S$ and a taxi ride $R$
output: A non-dominated set $\Sigma$
procedure FindOffers($S$, $R$)
    $\mathcal{T} \leftarrow \{T \mid T \in C \wedge C \in S\}$
    parallel foreach $T_i \in \mathcal{T}$ do
        $x_i \leftarrow$ schedule $R$ in $T_i$
    end
    initialize $D$ with $D_i \leftarrow 0$ for each $1 \le i \le |x|$
    parallel foreach $1 \le i, j \le |x|$ do
        if $x_i \succ x_j$ then $D_j \leftarrow 1$
    end
    $\Sigma \leftarrow \{x_i \mid D_i = 0\}$
end
```

4.4. Algorithm with precomputed routes

The previous section showed a constant time algorithm exists if enough processors are available. It is not practical to spend a fortune on a powerful computer and use the parallel algorithm to solve the problem within the desired amount of time. This section introduces an algorithm that tries to achieve good performance on a regular household computer.

The computation of a route between two locations with a routing API such as OSRM [35] is relatively time consuming in the naive method. It roughly costs 10 milliseconds per API call, depending on the computer, API interface and type of call. To overcome this bottleneck the distance and travel time between many locations can be precomputed and stored in a giant lookup table. However this comes with a downside, because when using the precomputed routes the cost for travelling between two locations can only be approximated. To compensate for this the algorithm of section 4.2 is slightly adjusted and the pseudo code of the new version is stated in algorithm 4. Just as the naive algorithm one should provide a schedule and a taxi ride, and the algorithm will give a set of non-dominated solutions. Please note that with only little adjustments a parallel version of the algorithm with precomputed routes can be made.

The algorithm first computes for every possibility an estimate for $f_w(x)$ and $f_c(x)$ and stores this in non-dominated levels. The UpdateLevels procedure is used to update this data structure. For two solutions x and x', the operator $x \succ_{(\delta_c,\delta_w)} x'$ is defined in the same way as $x \succ x'$, thus one of the following must hold:

- $f_c(x) + \delta_c \le f_c(x')$ and $|f_w(x)| + \delta_w < |f_w(x')|$ and $\text{sign}(f_w(x)) = \text{sign}(f_w(x'))$
- $f_c(x) + \delta_c < f_c(x')$ and $|f_w(x)| + \delta_w \le |f_w(x')|$ and $\text{sign}(f_w(x)) = \text{sign}(f_w(x'))$

Thus when two solutions are not more than δ_c and δ_w away from each other, they are considered equally good and therefore they should be in the same non-dominated level. This buffer is needed, because the $f_c(x)$ and $f_w(x)$ initially computed are inaccurate and should be recomputed with the routing API, this is only done for the best non-dominated level. However if it turns out that the first non-dominated level does not contain any feasible solution the next non-dominated level is used, and so on. This will happen more often for a low δ_c and δ_w, because in that case less solutions are considered equally good and less solutions are in the same non-dominated level. The algorithm is more efficient when only the non-dominated solutions are stored and all other solutions are discarded completely. However this will decrease the performance, because more often the solutions kept for reevaluation turn out to be infeasible. Note that no performance guarantee for this algorithm can be given, because a solution that should be in the non-dominated set could be marked infeasible when computing an approximation of the objective functions.

Algorithm 4: Pseudo code of the algorithm with precomputed routes

```
input  : A schedule $S$ and a taxi ride $R$
output: A non-dominated set $\Sigma$
procedure FindOffers($S, R$)
    $\Gamma \leftarrow \{\emptyset\}$; $\Sigma \leftarrow \emptyset$
    foreach $C_i \in S$ do
        foreach $T_j \in C_i$ do
            $X \leftarrow$ schedule $R$ in $T_j$ using cached routes
            foreach $x \in X$ do  UpdateLevels($\Gamma, x, 0$)
        end
    end
    foreach $\Gamma_l \in \Gamma$ do
        foreach $x \in \Gamma_l$ do
            recompute $x$ with routing API
            if $x$ is infeasible then continue
            if $\forall x^* \in \Sigma : x \succeq x^*$ then
                $\Sigma \leftarrow \{x' \mid x' \in \Sigma \wedge x \not\succ x'\}$
                $\Sigma \leftarrow \Sigma \cup \{x\}$
            end
        end
        if $|\Sigma| > 0$ then  break
    end
    return $\Sigma$
end
```

```
input  : Non-dominated levels $\Gamma$, solution $x$
         and level $l$
output: and updated $\Gamma$
procedure UpdateLevels($\Gamma, x, l$)
    if $|\Gamma| = l$ then
        $\Gamma \leftarrow \Gamma \cup \{\{x\}\}$
        return
    end
    dominated $\leftarrow$ false
    foreach $x' \in \Gamma_l$ do
        if $x \succ_{(\delta_c, \delta_w)} x'$ then
            $\Gamma_l \leftarrow \Gamma_l \setminus \{x'\}$
            UpdateLevels($\Gamma, x', l+1$)
        else if $x' \succ_{(\delta_c, \delta_w)} x$ then
            dominated $\leftarrow$ true
            break
        end
    end
    if dominated then  UpdateLevels($\Gamma_l, x$)
    else  $\Gamma_l \leftarrow \{x\}$
end
```

Figure 4.4 shows a simplified image of the working of the approximation of the distance and travel time. The solid line indicates the route stored in the lookup table. The dotted lines indicate the error introduced by the approximation and the two rectangles indicate the size of the cell used for the selection of locations. This figure shows that the larger the cell is, the more the queried location could be away from the reference location, the larger the error in approximation is. The image also shows the position of the reference location within the cell is important.

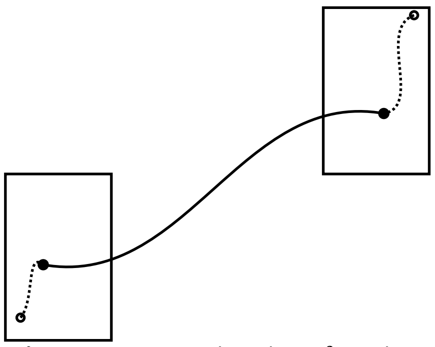


Figure 4.4: Approximation of routing

The lookup table should give an accurate approximation of the distance and travel time for a route, but it must also fit into the memory of the computer. Another constraint is the time required to collect the data to fill the lookup table. Several methods for selecting the locations that will be used as reference can be used. For every pair of locations the route is computed by a routing API and stored on disk, such that our algorithm can load this file into memory in the initialization phase. The first selection method loops over the longitude and latitude with predefined steps. However this results in many useless locations that are for example located in the sea. The second one uses a list of ZIP codes accompanied with latitude and longitude. Because the complete list of ZIP codes is too large a few methods of reducing the size of this list are explored. These methods are listed in table 4.1 together with the number of locations and the time to compute the distance/duration matrix. The abbreviations 'PC4', 'PC5' and 'PC6' indicate that only the first four, five and six characters are considered of the Dutch ZIP code, which starts with four digits and ends with two letters.

In figure 4.5 an estimation of the computation time for the distance/duration matrix given the number of locations is plotted. The storage required for the distance/duration matrix grows just as for the computation time quadratically with the number of locations, because for n locations there are $\mathcal{O}(n^2)$ pairs of locations. Only half of the distance/duration matrix is filled, thus for every pair of locations (p_1, p_2) either the distance and travel time is computed from p_1 to p_2 or from p_2 to p_1. This reduces the time to collect the data and the memory to store it significantly.

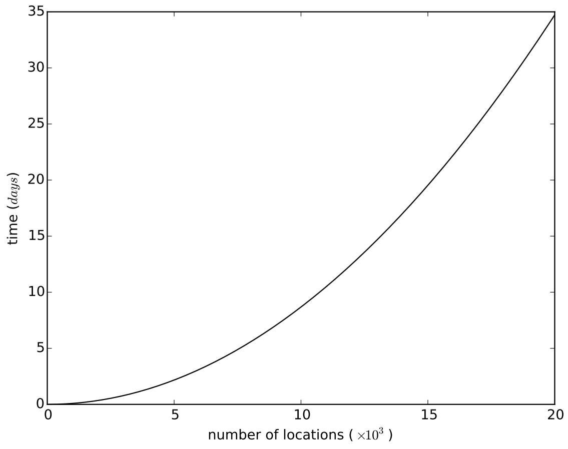


Figure 4.5: Estimated computation time for distance matrix

method	# locs.	est. time	cell dia. (m)
PC6	471993	53 years	–
PC5	35138	107 days	–
PC4	4766	47 hours	–
PC6⊞2	11073	11 days	2624
PC6⊞3	5407	62.1 hours	3926
PC6⊞4	3185	21.6 hours	5248
PC6⊞8	900	1.7 hours	10495

Table 4.1: Estimated computation time for distance matrix with different selection methods

For the method called "PC6⊞2" a grid with cells of size 0.02 × 0.02 degrees longitude/latitude is used. This roughly corresponds with cells of size 2.2 × 1.4 kilometres. For each of these cells one of the selected locations is randomly selected. This resulted in a list of evenly distributed locations, this is plotted in figure 4.6. For every group of ZIP codes with the same numbers one is selected randomly and plotted in figure 4.7. For the selection methods called "PC6⊞3", "PC6⊞4" and "PC6⊞8" larger cells are used and the approximations will have a larger error. In table 4.1 the cell diameters are listed, which will gives an indication of the error. The error with different selection methods is evaluated empirically in section 6.4.

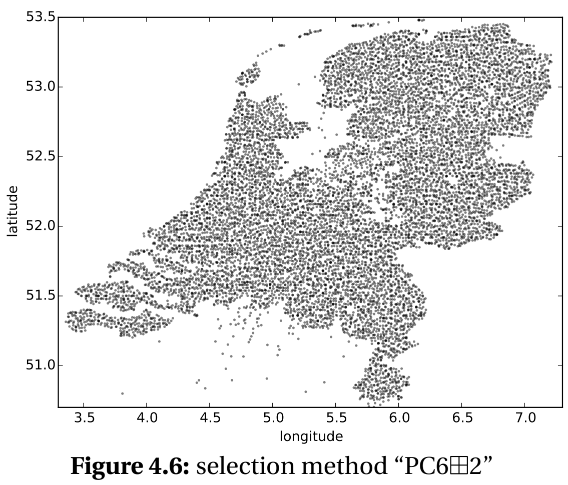


Figure 4.6: selection method "PC6⊞2"

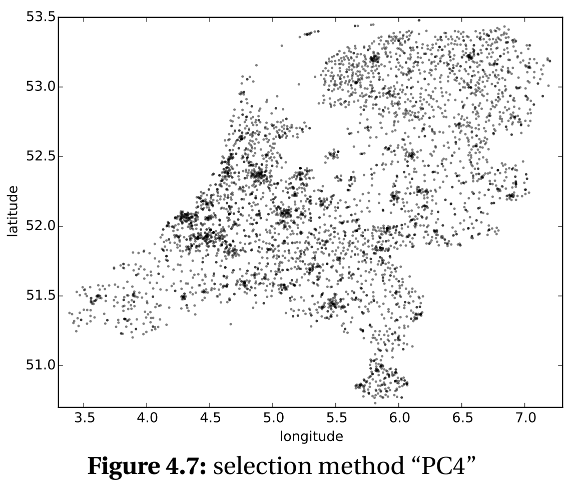


Figure 4.7: selection method "PC4"

4.5. Niching methods

Preliminary experiments with randomly generated problem instances often showed that solutions in the non-dominated set are similar to other solutions in the same non-dominated set. This section will introduce methods to reduce the number of solutions by omitting similar solutions, which answers the following subquestion: “How can the list of non-dominated solutions be reduced to a specific size, such that similar solutions are discarded?” This is useful when for example only limited space is available on a website.

After applying the niching method to the non-dominated set, the remaining solutions should still be distributed over a large extent, because only then a customer has large variety of options he or she could be interested in. The metrics for distribution and extent are defined as $\mathcal{M}_2$ and $\mathcal{M}_3$ in section 3.3.2. The extent can be kept the same simply by keeping the two solutions with the largest distance. A straightforward niching approach is to use a grid and whenever multiple solutions are in the same cell all but one must be discarded. However two solutions in neighbouring cells can still be very close together, so this does not ensure that no solution is in another’s neighbourhood. Moreover selecting the grid size, such that enough solutions are discarded, is a non-trivial task.

One arguably better approach is to keep discarding solutions until $\mathcal{M}_2(X') = |X'|$, with $X' \subseteq X$ and X the original set. This could be done by evaluating the non-dominated set sorted on x-coordinate, and discard the next solution if it is less than σ away from the previous solution and σ could be chosen depending on the number of solutions someone needs in the resulting set. Note that it might be needed to tweak σ to distinguish the quality of two possible resulting sets. Therefore we might conclude that $\mathcal{M}_2$ is not the best metric to measure the distribution of the solutions in the non-dominated set.

Therefore we propose a new metric $\mathcal{M}_4$ for the distribution of a set of solutions, which is defined in equation 4.6. The advantage of this metric is that it needs no additional parameters. It is also more expressive in the sense that in general with two different subsets $X', X'' \subseteq X$ where $|X'| = |X''|$ the inequality $\mathcal{M}_4(X') \neq \mathcal{M}_4(X'')$ holds. In contrast to the other metric where $\mathcal{M}_2(X') = \mathcal{M}_2(X'')$ is often the case. Therefore $\mathcal{M}_4$ is likely to work better in combination with a local search algorithm.

$$\mathcal{M}_4(X) = \frac{1}{|X|} \sum_{x \in X} \left(\min_{x' \in X, x' \neq x} \{d(x, x')\} \right) \tag{4.6}$$

The problem introduced in this section could also be formulated as: Find $X' \subseteq X$, such that $|X'| = m$, $\mathcal{M}_4(X')$ is maximal and $\mathcal{M}_1(X) = \mathcal{M}_1(X')$. Again the latter can only be done by keeping the outer two solutions. In any case someone might find it unfair to discard solutions solely on metric $\mathcal{M}_4$, because a lot of solutions could be discarded from the same company. This could be fixed by considering the solutions of companies separately and give each company C_i its own number of solutions in the resulting set $m_i < m$. A method for choosing a value for each m_i is to compute the ratio of the number of solutions of company C_i in X with respect to the other companies.

5 Expected cost of a taxi ride

In the previous chapter methods for finding offers were described, which can give large discounts to customers. However the first customer will never get a discount, because no previously booked rides exist. This chapter tries to define a method that can also give a discount to the first customer. Therefore the question we try to answer is: “How can an online algorithm efficiently compute the expected cost of a taxi ride given the historical data of a taxi company, within a small enough period of time?” As described in chapter 2 and 4, companies combine taxi rides to reduce cost and this reduction could be forwarded to the customer in a discount. This chapter tries to introduce a method to evenly distribute this discount over all customers. By combining the newly ordered ride with expected future taxi ride requests also the first customer can profit from a discount. This chapter continues with section 5.1, which describes the information needed to calculate the expected cost of a taxi ride, and concludes with section 5.2, which explains two methods to compute this expected cost.

5.1. Cost evaluation

The method to calculate the expected cost described in this section assumes that one knows the cost of operating a taxi. That means one should know what the cost is to drive a certain distance with a vehicle and what does it cost to use a vehicle for a certain amount of time. With this information you can easily obtain the constant cost of a taxi ride, that is the time needed to load and unload baggage, the time needed by a dispatcher to make sure the customer is served, etc. This startup cost can also reflect other types of fixed costs, but the customer should always pay the startup cost and is therefore ignored in the rest of this chapter. The cost per kilometre and cost per minute should cover for the cost for hiring the driver, insurance, fuel, maintaining the vehicle, maintaining the headquarters building, marketing, etc. With this information the total cost of a schedule with a list of rides for one day can be computed. The expected cost of ride R_j given expected rides $\mathcal{R}$ is defined by the difference between the total cost of $\mathcal{R}$ and $\mathcal{R} \cup \{R_j\}$. However computing this is very computationally intensive, because of computing these schedules. The rest of this section will describe an alternative approach that is feasible in a real-time setting.

To compute the expected cost of each taxi ride we split up the route in three segments and argue that the cost of these parts should be computed in a different way. The first segment is driving from the company’s base or current location to the pickup location of the customer. Secondly, the taxi drives with the customer from the pickup location to the drop off location. Lastly, the taxi has to drive from the drop off location back to its base or another important location. For the cost of the segment the customer is in the taxi the standard fares should be used, because this segment cannot be sold to another customer. The first and last segment could have the same fares as the second segment, unless another customer can be served during this time. Therefore we should be able to say something about the average size of the part of the segment which can be filled up by other taxi rides. The relative size of the part of the segment that is not filled up will be called the ratio to pay. If for example half of a segment can be filled up by another taxi ride, the cost for this segment is half the normal cost and thus the ratio to pay is 0.5.

To come up with the expected ratio to pay for each possible segment, a number of historic days will be evaluated. How the historic data is used is discussed in the next section in more detail. It is important that all customers are served at the appointed time, this will cause a waiting time for taxi drivers. Therefore the ratio to pay can differ between cost per minute and cost per kilometre. When the volume of rides is higher the probability a good match can be found will be higher, and the ratio to pay lower. The number of rides can be influenced by many causes, for example the pickup location, the drop off location, the time of the day, the day of the week, the season, the weather, holidays and local events, such as concerts. Thus we try to find a function that maps all these features to the ratio to pay for both minutes and kilometres.

Since there are many features, we limit ourselves to the three most important ones, namely pickup location, drop off location and time of the day. Thus we need to learn a function $f_{fare} : (\mathbb{R}, \mathbb{R}, \mathbb{N}, \{0,1\}) \mapsto (\mathbb{R}, \mathbb{R})$. Which maps a quadruple with latitude, longitude, time of the day and a boolean to a pair that indicates the expected cost per kilometre and the expected cost per minute. The boolean indicates if false the first segment and if true the last segment of the route. Computing the cost per minute for the first segment works slightly different than for the last segment. Note that we assume one fixed base location, which is therefore not a parameter of this function.

5.2. Expected taxi ride cost algorithm

As the previous section describes we are looking for an algorithm that can find the function that maps the location and time to the expected cost per kilometre and minute given a base location and a history of taxi rides.

The algorithm is given a large list of locations L, it will learn how much of the route between the company's base p_C and each location, in both directions, can be filled up by other rides. These other rides come from a history of taxi rides H. The list L should be constructed such that it covers the area customers will book rides to. The route segments from p_C to $p_i \in L$ and $p_i \in L$ to p_C are learned separately, because the latter route could be more popular than the former. The function *days*(H) gives a set of all days a customer is picked up for taxi ride for all rides $R \in H$. The function *rides*(H, d) denotes the set of all rides $R \in H$ performed on day d.

If the first segment of ride R_j can be partially filled up with ride R_i, then the cost of the first segment of R_j is defined by $C_d \cdot f^d_{segment}(p_C, p^p_j, R_i) + C_t \cdot f^t_{segment}(p_C, p^p_j, R_i, t^p_j, \text{FORTH})$. If the last segment of ride R_j can be partially filled up with R_i, then the cost of the first segment of R_j is defined by $C_d \cdot f^d_{segment}(p^d_j, p_C, R_i) + C_t \cdot f^t_{segment}(p^d_j, p_C, R_i, t^d_j, \text{BACK})$. Please recall that C_d and C_t denote the cost per kilometre and cost per minute. See equations 5.1 and 5.2 for the definition of function $f^d_{segment}$ and $f^t_{segment}$. The functions f_d and f_t denote the travel distance in kilometres and the travel time in minutes, which were first defined in chapter 2.

$$f^d_{segment}(p_1, p_2, R_i) = f_d(\langle p_1, p^p_i \rangle) + f_d(\langle p^d_i, p_2 \rangle) \tag{5.1}$$

$$f^t_{segment}(p_1, p_2, R_i, t, dir) = \begin{cases} \infty & \text{if infeasible} \\ f_t(\langle p^d_i, p_2 \rangle) + t^p_i - t & \text{if } dir = \text{BACK} \\ f_t(\langle p_1, p^p_i \rangle) + t - t^d_i & \text{if } dir = \text{FORTH} \end{cases} \tag{5.2}$$

The LEARNROUTE procedure, stated in algorithm 5, simply iterates over all locations and stores for each location and direction two polynomials, one for the minute fare and one for the kilometre fare, these functions map the time of the day to the ratio to pay. These polynomials are computed by the LEARNSEGMENT procedure, stated in algorithm 6. A high degree polynomial to model the fare for each time of the day costs significant less memory than storing a value for every minute of the day. The two methods are compared with each other in section 6.6.1.For each historic day the best match is found by exhaustive search. The best match is the one with the lowest cost, therefore the standard cost per minute and kilometre is needed. This is used to compute the part the taxi drives empty, which results in the ratio to pay. Only the ratio to pay is stored and not the match itself and the average of all ratios over all days is computed. This is done for every minute of the day which results in a lot of data. To reduce the amount of data the relation between time and cost to pay is expressed with a high degree polynomial, that is fitted with the Levenberg-Marquardt algorithm (LMA). This polynomial also expresses the factor to pay in a much smoother way, which gives less surprising results for customers. Note that all of this is done separately for travelling time and distance, because in the travelling time also the waiting time of the taxi is incorporated. By adding the waiting time to the travelling time the two ratios are different. For each location and direction of driving the two polynomials are stored in model M.

Algorithm 5: The main expected cost learn procedure

```
input  : a base location $p_C$ of taxi company $C$
         a list of interesting locations $L$
         a history of taxi rides $H$ normal cost per minute $c_m$ and per kilometre $c_d$
output: a model $M$ company $C$ can use
procedure LearnRoute($p_C, L, H$)
    foreach $p_i \in L$ do
        $M[i,\text{FORTH}] \leftarrow$ LearnSegment($p_C, p_i$, FORTH, $c_d, c_m$)
        $M[i,\text{BACK}] \leftarrow$ LearnSegment($p_i, p_C$, BACK, $c_d, c_m$)
    end
end
```

When a customer requests the price for a taxi ride R the procedure Price, stated in algorithm 7, can compute the cost for this ride by making use of model M. For the first and last segment the factor to pay is calculated by interpolating the polynomials of the nearby locations in list L. The interpolation needs to be done separately for the kilometre and minute factor to pay. The number of polynomials considered during the interpolation depends on the chosen interpolation method. In section 3.7 different interpolations methods that can handle spatial data in an irregular grid are discussed. A few of them are compared in chapter 6. Another approach to this problem is to make four functions from all the data collected by procedure LearnSegment, instead of storing the polynomials for each location separately. We need four functions, because we need one for the minutes and one for the kilometres for both directions. This could be done with a Multilayer Perceptron, kernel regression or multiple regression, both discussed in section 3.6. These methods are compared in chapter 6.

Algorithm 6: The expected cost learn procedure for one route segment

```
input  : two locations $p_1$ and $p_2$
         a direction of the route $dir$ indicating if the departure time or arrival time is a hard constraint
         normal cost per minute $c_m$ and per kilometre $c_d$
output: two functions $f : \mathbb{N} \mapsto \mathbb{R}$ that maps the time to the cost factor for time and distance
procedure LearnSegment($p_1, p_2, dir \in \{\text{BACK},\text{FORTH}\}, c_d, c_m$)
    normal $\leftarrow f_{d/m}(\langle p_1, p_2\rangle)$        /* (distance,duration) for traveling from $p_1$ to $p_2$ */
    sum $\leftarrow \{(0,0) \mid t \in \{1,2,3,\ldots,24\cdot 60\}$
    foreach $d \in days(H)$ do
        best $\leftarrow \{(1,1) \mid t \in \{1,2,3,\ldots,24\cdot 60\}$
        foreach $R_i \in rides(H,d)$ do
            foreach minute $t \in \{1,2,3,\ldots,24\cdot 60\}$ do
                other $\leftarrow (c_d \cdot f^d_{segment}(p_1,p_2,R_i), c_m \cdot f^m_{segment}(p_1,p_2,R_i,t,dir))$
                alt $\leftarrow$ other / normal                        /* element wise operations */
                if $c_d \cdot \text{alt}_d + c_m \cdot \text{alt}_m < c_d \cdot \text{best}(t)_d + c_m \cdot \text{best}(t)_m$ then
                    best($t$) $\leftarrow$ alt
                end
            end
        end
        sum $\leftarrow$ sum + best                                    /* element wise addition */
    end
    avg $\leftarrow$ sum / $|days(H)|$                                 /* convert sum to average */
    $f_d \leftarrow$ LMA($\{1,2,3,\ldots,24\cdot 60\}$, avg$_d$)
    $f_m \leftarrow$ LMA($\{1,2,3,\ldots,24\cdot 60\}$, avg$_m$)
    return $(f_d, f_m)$
end
```

Algorithm 7: The compute expected price procedure

```
input  : a list of locations $L$ used to train model $M$
         a taxi ride $R_i$
         normal cost per minute $c_m$ and per kilometre $c_d$
         base location of taxi company
output: the expected cost of the taxi ride $c \in \mathbb{R}$
procedure PRICE($L, M, R_i, c_m, c_d, p_C$)
    $c \leftarrow$ cost for driving from $p_i^p$ to $p_i^d$
    interpolate $M$ for locations $p_i^p$ and $p_i^d$
    part1 $\leftarrow M[p_i^p,\text{FORTH}]_d(t_i^p) \cdot f_d(\langle p_c, p_i^p\rangle) + M[p_i^p,\text{FORTH}]_m(t_i^p) \cdot f_d(\langle p_c, p_i^p\rangle)$
    part3 $\leftarrow M[p_i^d,\text{BACK}]_d(t_i^d) \cdot f_d(\langle p_i^d, p_C\rangle) + M[p_i^d,\text{BACK}]_m(t_i^d) \cdot f_m(\langle p_i^d, p_C\rangle)$
    $c \leftarrow c$ + part1 + part3
    return $c$
end
```

5.3. Risk aversion

The previous section describes a method for a risk neutral cost assignment of taxi rides. Risk aversion is human behaviour studied by both economists and psychologists and three types of attitudes towards risk can be identified. Firstly, with risk neutral behaviour one is insensitive to a guaranteed payment or an expected payment of the same amount. Secondly, with risk seeking behaviour a person, e.g. a lottery participant, is willing to accept an expected loss. Lastly, a risk averse or risk avoiding behaviour a person is willing to pay money to reduce risk. The latter is the reason why insurance companies exist. This section proposes four possible modifications to the method described in the previous section, such that a risk averse setting is possible.

Only one possible way to reduce the risk of loss on a taxi ride is to increase the price, but the real question is: "By how much should the price be increased such that the risk on a loss is acceptable for the company?" Note that one can favour a large probability on a small loss over a small probability on a large loss. If we increase every price by a fixed amount or a fixed percentage of the price, the whole idea of doing something with the likelihood of future taxi rides is lost. This is because the price could exceed the initial way of computing the cost of a taxi ride. Besides too little or too much risk could be avoided in different scenarios. Therefore we need a function that can compute a price that is not lower than the expected cost and not higher than the standard price, i.e. without any discount. It is favourable that the function also has a parameter α, a taxi company can use to scale the risk it is willing to take. The price should be computed with the best matches on the evaluated historic days, which will be referred to as observations. Assume we have n observations x_i sorted in ascending order, that is $x_1 \le x_2 \le ... \le x_n$, where each x_i indicates the amount of money needed to break even divided by the standard price on a given route segment, historic day and time of the day. Note with a more risk averse behaviour the probability a customer will look for a competitor increases. This lowers the number of future taxi rides, increase the expected cost and therefore also the risk of having a loss. Modelling customer behaviour and finding an equilibrium between avoiding risk and attracting many customers with a low price is outside the scope of this thesis. However the following methods are easy to comprehend for the company's owner or sales manager.

Arguably the most simple possible method is to use the weighted average of the mean and the worst case scenario. The function $g_1(x, \alpha_1)$ is equation 5.3 describes this method, where $\bar{x}$ is the unweighted average over all n observations and $\alpha_1 \in [0, 1]$ models risk neutral behaviour with $\alpha_1 = 0$ and extremely risk averse behaviour with $\alpha_1 = 1$. Another option is to limit a maximum allowable loss with $\alpha_2 \ge 0$. Function $g_2(x, \alpha_2)$ defined in equation 5.4 assures the price is at least the expected cost. The parameter α_2 could be set with a percentage of the standard price. Note that $\alpha_2 = 0$ models the most risk averse behaviour.

$$g_1(x, \alpha_1) = (1 - \alpha_1) \cdot \bar{x} + \alpha_1 \cdot x_n \tag{5.3}$$

$$g_2(x, \alpha_2) = \max(\bar{x}, x_n - \alpha_2) \tag{5.4}$$

Though the functions $g_1(x, \alpha_1)$ and $g_2(x, \alpha_2)$ are easy to understand it does not fully exploit knowledge about the distribution of observations. The $(100 \cdot \alpha)^{\text{th}}$ percentile could used as price but this method has the unwanted property that it can give values below the expected cost. For example if a company uses the 60th percentile, then with a probability of only 40% a taxi ride costs money, but the expected revenue could still be below zero. However if first all observations below the mean are discarded this will work. This method is defined by function $g_3(x, \alpha_3)$ in equation 5.5, where $\alpha_3 \in [0,1]$ models the risk attitude. Where the probability of a loss is reduced by a factor of α_3, so a (near) risk neutral behaviour is modelled with $\alpha_3 = 0$ and the most risk averse behaviour is modelled with $\alpha_3 = 1$.

$$g_3(x, \alpha_3) = \{x_i \mid x_i \geq \bar{x}\}_{\lfloor \alpha_3 \cdot (|\{x_i \mid x_i \geq \bar{x}\}|-1) \rfloor + 1} \tag{5.5}$$

Using a weighted average of all observations instead is another possible method, which uses the complete distribution of observations. The observation where more money is required to make break even on a ride should be weighted more. The weighted average $\bar{x}_w$ of all x_i, where each observation x_i has weight w_i is defined in equation 5.6. This only works if the weights are non-negative and in a non-decreasing order $0 \leq w_1 \leq w_2 \leq \ldots \leq w_n$, because then the weighted average exceeds the unweighted average. Note that with weights $w_1 = w_2 = \ldots = w_n$ an unweighted average and risk neutral behaviour is simulated. The weighted average is easy to compute and only one parameter is needed namely a non-decreasing weighting function, for example $w_i = i$ could be used. A weighted average is, with a non-decreasing weighting function, higher for a set of observations with a high variance than for one with the same mean, but with a lower variance. For example sets $X_1 = \{4,5,6\}$ and $X_2 = \{1,5,9\}$ both have a mean of 5, but the weighted averages with $w_i = i$ are $5\frac{1}{3}$ and $6\frac{1}{3}$, respectively. If the weights $w_i = \alpha_4 \frac{i}{n} + 1 - \alpha_4$ are used, then with parameter $\alpha_4 \in [0,1]$ the magnitude of risk aversion can be scaled. For $\alpha_4 = 1$ the maximum risk aversion is obtained and with $\alpha_4 = 0$ a risk neutral behaviour is simulated. Note that with a non-linear weighting function more risk aversion can be achieved.

$$\bar{x}_w = \sum_{i=1}^{n} w_i \cdot x_i \Bigg/ \sum_{i=1}^{n} w_i \tag{5.6}$$

The last method is possibly less suitable compared to the other methods prosed in this section, because it is harder to explain to the director of a taxi company and for example the linear weighting function does not allow to model a completely risk averse behaviour, but some companies could find this unnecessary. Various weighting functions are experimented with in section 6.7 as well as the methods described by the functions g_1, g_2 and g_3.

6 Empirical evaluation

This chapter describes the empirical evaluation of the algorithms described in chapter 4. This section finds for each algorithm the maximum number of taxi rides that can be considered while respecting the one second computation time constraint. For this evaluation a dataset is constructed in section 6.1. This dataset makes it possible to analyse the performance of the algorithms in a real-life setting. For the naive algorithm we evaluate the runtime, in section 6.2, to create a baseline for the other algorithms. In section 6.3 the speedup achieved by the parallel algorithm is analysed with various number of processor cores. The quality of approximation of distance and travel time with the precomputed routes is analysed separately in section 6.4, before the solution quality and runtime of the algorithm with precomputed routes is evaluated in section 6.5. Section 6.6 describes a comparison of the methods needed for the algorithm proposed in chapter 5. The methods to avoid risk proposed in section 5.3 are compared in section 6.7. For these comparisons another dataset is used, which is also considered in section 6.1.

6.1. Taxi ride datasets

This section describes the datasets used for the experiments, namely two large datasets from New York City described in section 6.1.1 and an artificial dataset for The Netherlands described in section 6.1.2. The latter is artificial, because no real-world datasets exist that include all taxi rides driven in The Netherlands. The Dutch dataset is used for the empirical evaluation of the algorithms described in chapter 4. The New York City dataset is used for the empirical evaluations of the algorithms described in section 6.6. This evaluation required a real-world dataset of at least two successive months. Other datasets exists for Porto in Portugal [67], Rome in Italy [10] and San Francisco in the USA [50] do exists, but they only include a subset of the vehicles or are hard to parse.

6.1.1. New York City

In New York City, USA the taxis come into two flavours, namely canary yellow and apple green. Both flavours are coordinated by the NYC Taxi & Limousine Commission and drive for the same fares. The canary yellow taxis, also called medallion taxis, mainly pickup passengers in Lower Manhattan, Midtown Manhattan, LaGuardia Airport and JFK airport, because in these parts of the city the most money can be earned by the taxi drivers. The apple green taxis, normally called Boro Taxis, drive in all 5 boroughs, except they are legally not allowed to pickup passengers in Lower Manhattan and at the airports. The datasets are from now on called 'yellow taxi data' [65] and 'green taxi data' [64]. The yellow taxi data contain over 173 million taxi rides that were driven in 2013. The green taxi data contain over 8 million taxi rides that were driven between August 2013 and June 2014. The pickup and drop off locations of a subset of both datasets are plotted in figure 6.1 and 6.2. These subsets were randomly selected and both contain 10,000 taxi rides. Note that in figure 6.2 the drop off location of the Boro taxis are partially covered by the medallion taxis.

Figure 6.3 shows preliminary research on the yellow taxi data. The plot shows the distribution of taxi rides over the time of the day for all days in the week. Time windows of 5 minutes are used. One can see that

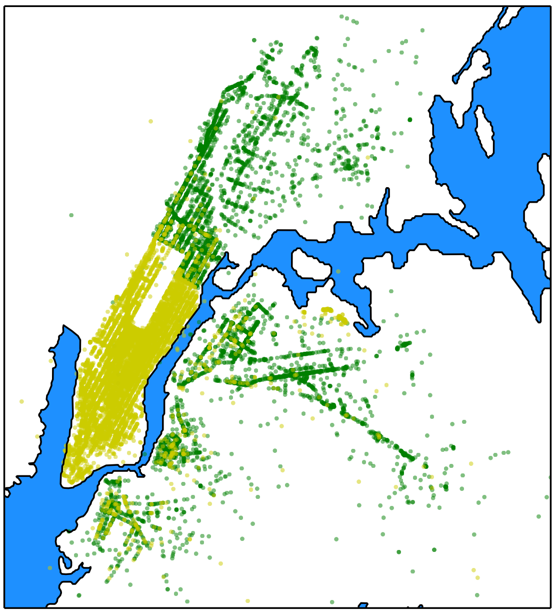


Figure 6.1: Pick up locations of yellow and green taxis in NYC

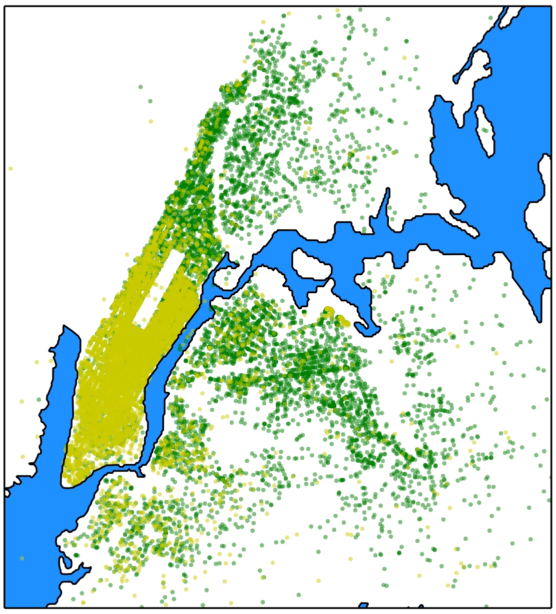


Figure 6.2: Drop off locations of yellow and green taxis in NYC

Fridays and Saturdays are the most popular and Mondays and Sundays are the least popular days of the week to travel with a taxi. One can also see two dips per day this is caused by driver switches [71] and caused by the fact that most people sleep at night.

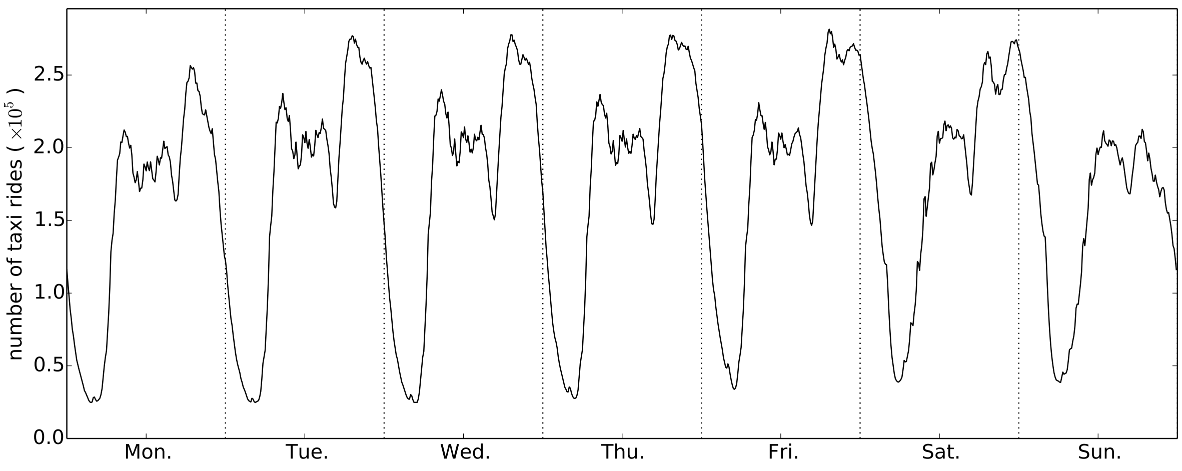


Figure 6.3: Distribution of taxi rides over 5 minute time windows

6.1.2. The Netherlands

There is no significantly sized dataset available for The Netherlands, therefore one is generated by a model of the population density of The Netherlands. In order to generate taxi rides based upon the population density the area and number of residents for every Dutch municipality is needed. This information is extracted from Wikipedia [1]. A taxi ride can be generated to randomly select two locations and two date and a random pickup time. The day of the week and the pickup time are drawn independently with the same distribution as the New York City dataset has. To generate a random location one of the 393 municipalities is randomly selected. The municipalities with more residents have a higher probability to be selected. Then based on the area A the values Δx and Δy are computed and added to the location of the municipality.

$$\alpha = 2\pi \cdot X \tag{6.1a}$$
$$r = \sqrt{A/\pi} \cdot Y \tag{6.1b}$$
$$\Delta x = r \cdot \cos(\alpha) \tag{6.1c}$$
$$\Delta y = r \cdot \sin(\alpha) \tag{6.1d}$$

For figure 6.4 10,000,000 locations were randomly generated. In figure 6.5 the density of zip codes [2] in The Netherlands are shown, which also indicates the more populated areas in The Netherlands. The list of zip codes in not used for generating the dataset, because the number of resident per zip code is unknown and the model would be more memory consuming.

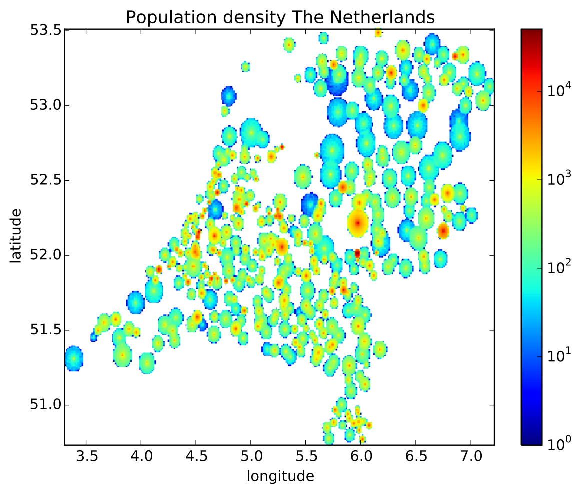


Figure 6.4: The simulated population density in The Netherlands

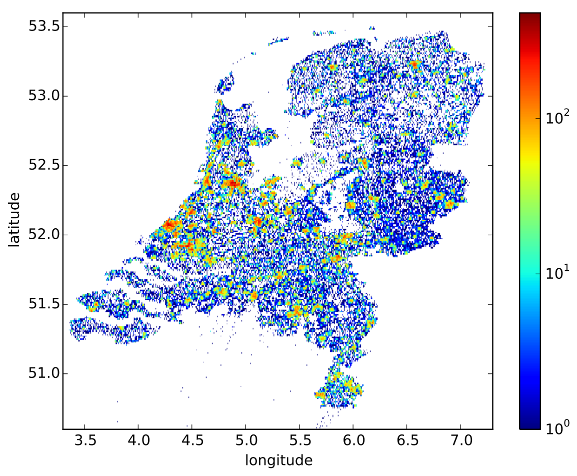


Figure 6.5: Density of zip codes in The Netherlands

6.2. Naive algorithm

Recall from chapter 2 that the OpenStreetMap Routing Machine (OSRM) [35] is used to compute the distance and travel time of a route. This subsection describes a runtime analysis of two versions of the naive algorithm implemented in C++11. The first version uses the HTTP API of OSRM and the other version uses the C++ API of OSRM. The C++ API is expected to be faster, because it does not have a network overhead which the HTTP API does have. The runtime analysis will also give insight into the maximum problem size that is still feasible to solve in one second of computation time. Although the algorithm has a runtime complexity of $\mathcal{O}(n^2)$ the runtime is expected to grow linearly with the number of taxi rides, because the calculation of routes will dominate the runtime, especially for small instances. All experiments are conducted on a `Intel(R) Core(TM) i7-2600K CPU` running at 3.40GHz.

All test data was randomly generated based on the model described in section 6.1.2. This dataset was used, because we are most interested on the performance of the algorithn with the Dutch use case. The fact the data is not real is less important, because only the run time is analysed. First for each problem size n taxi rides are generated and assigned to n taxi companies. Thus each of these taxi companies has one taxi ride scheduled and has their own base location generated with the model for the population density in The Netherlands. The problem instances constructed in this way have the largest number of feasible solutions with the smallest number of already schedules taxi rides and are therefore considered the worst-case instances. All taxi companies have the same cost per kilometre and cost per minute, which is for the runtime analysis unimportant. After n taxi rides are generated and assigned to taxis, a new ride is generated either from or to Schiphol with 50% probability, because many rides are to or from Schiphol. Then the runtime is measured for calculating the best solutions for this new taxi ride. This is done for a problem sizes ranging between 0 and 40 and for every problem size the experiment is repeated 40 times. The averages and the 80% confidence intervals are reported, in figure 6.6 as a curve and a filled area respectively. In this figure a dotted line is drawn at one second, which indicates the maximum runtime for a real-time algorithm. From this figure we can conclude that a schedule can only contain 20 taxi rides at any given time, if the C++ API of OSRM is used. Other the naive algorithm would take too long in a real-time setting. As expected algorithm that uses the HTTP API of OSRM performed worse and could only deal with 15 taxi rides.

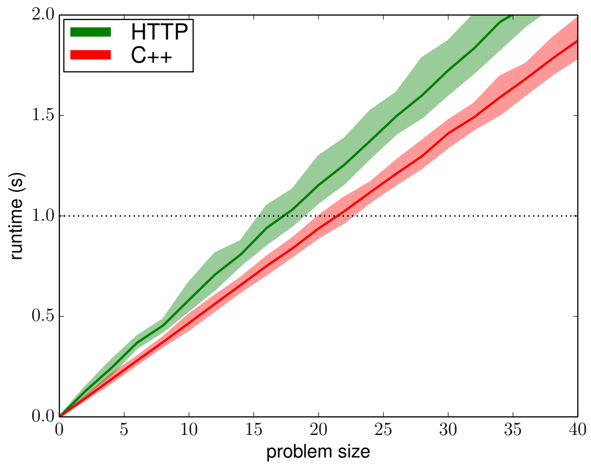


Figure 6.6: Runtime analysis of naive method

6.3. Parallel algorithm

This section describes the empirical evaluation of the parallel algorithm and the latter is described in section 4.3. The evaluation tries to find three answers, namely the runtime for various problem sizes, the speedup with various number of threads and the efficiency obtained with various number of threads. With the first we can determine the maximum problem size feasible to solve in one second. The metrics speedup and efficiency show the usefulness of the parallel algorithm. The speedup obtained with p processors or threads $S_p(n)$ for a problem size n is defined in equation 6.2, which is the runtime complexity of the best sequential algorithm divided by the runtime complexity of the algorithm that uses p processors. The efficiency $E_p(n)$ of a parallel algorithm is a performance measure that can give insight into the effective utilization of the processors, this metric is defined in equation 6.3.

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \tag{6.2}$$

$$E_p(n) = \frac{T_1(n)}{p \cdot T_p(n)} \tag{6.3}$$

Theoretically $S_p(n) = p$, but we do not expect that to be true in practice, because the creation of threads will cause overhead. However we expect the speedup to be near the number of processors and the efficiency above 90%. Additional overhead is caused by OSRM, because it locks certain resources. The overhead caused by OSRM can be resolved by giving each thread its own OSRM object, however this will significantly increase memory consumption. Namely by roughly 250 MB per thread for The Netherlands, which can be even more for larger countries. The evaluation described in this section is only done with the C++ API of OSRM. The parallel algorithm is implemented with the use of the `std::thread` provided by C++11. The experiments conducted, are again done on the `Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz`, which has 4 physical cores and 8 logical cores. Intel's Hyper-Threading Technology (HTT) is turned on, because this resulted in a better performance. Unfortunately this makes it a bit harder, but not impossible to interpret the speedup and efficiency metrics.

The analysis on the speedup and efficiency are conducted on problem instances with 500 taxi companies with each one taxi and one already scheduled taxi ride. The values reported in the plots are averages of 40 random problem instances, but the same are used for different number of threads allowing a fair comparison. The instances are generated in the same way as done for the evaluation performed in section 6.2. In figure 6.7

the speedup obtained with a different number of threads is plotted. For the memory efficient version of the parallel algorithm a speedup of 3.7 and 3.8 were achieved with 8 and 16 threads respectively. For the memory inefficient version a speedup of 3.8 and 3.9 were achieved with 8 and 16 threads respectively. However after doing a t-test with a p-value of 0.05, the observation of the memory inefficient algorithm being faster than the memory efficient one, turned out to be insignificant. Unfortunately we have to conclude that in practice $S_p(n) \neq p$, but it did come relatively close. Figure 6.7 also shows that the speedup scales linearly with the number of threads used until 8 threads.

Figure 6.8 shows the efficiency of the threads utilization, but note that this is a misleading due to HTT. Therefore we repeated the experiment with HTT turned off for 1 and 4 processors. The results and the computation of the effiency is stated in 6.4 for a problem size of $n = 500$.

$$E_4(500) = \frac{T_1(500)}{4 \cdot T_4(500)} \approx \frac{17843}{4 \cdot 5866} \approx 0.760 \tag{6.4}$$

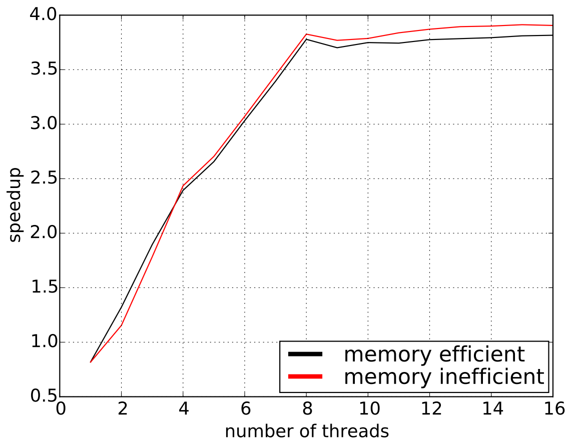


Figure 6.7: Speedup obtained with different number of threads and $n = 500$

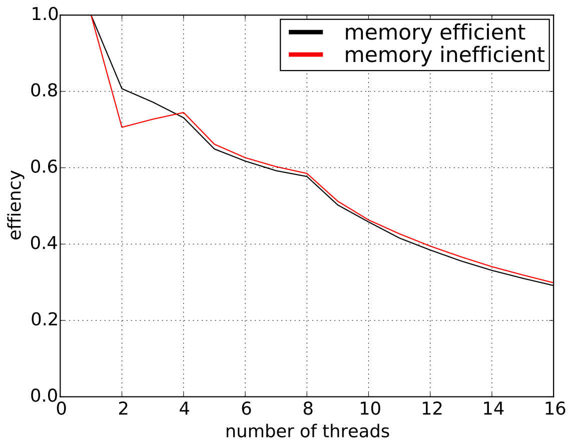


Figure 6.8: Efficiency obtained with different number of threads and $n = 500$

The analysis on the runtime is conducted by comparing the naive algorithm with the memory efficient parallel algorithm using either 1, 2, 4 or 8 threads. The problem sizes ranges from $n = 0$ to $n = 250$ in steps of 5. The results of this experiment are plotted in figure 6.9, where each value is an average obtained by performing the experiment of 40 different random problem instances. Note that for each algorithm the same problem instances are used. From this experiment we can conclude that the runtime of the parallel algorithm, just as for the naive algorithm, appears to scale linearly with the problem size. Which means that the calculation of the routes by OSRM still dominates the runtime of the algorithm. The figure also indicates the performance decrease for the parallel algorithm with 1 thread compared to the naive algorithm. Additionally the sequential algorithm also suffers a performance loss due to HTT.

6.4. Quality of route approximation

This section evaluates the quality of the approximation of the cost for travelling between two locations. This gives insight into how well the route approximation works and how much it can be improved. This information is also a good basis for the evaluation of the algorithm that makes use of these precomputed routes.

To measure the quality of the approximation of routes, a pair of random locations is generated and the distance and travel time computed by the OpenStreetMap Routing Machine is compared to the approximation done with the precomputed routes. This is done for the four selection methods that make use of grid stated in table 4.1. The other methods are either infeasible, because of the time required to collect all the necessary data, or turned out to be very bad in preliminary experiments. The cost is calculated by adding €0.50 for every minute and €0.10 for every kilometre.

In figure 6.10 the distribution of 500.000 measurements has been plotted for four different selection methods. On the x-axis of this figure the difference in euro between the approximated and the actual cost for each route.

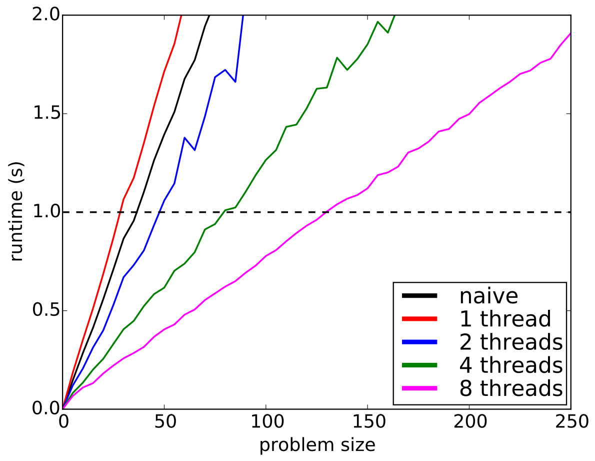


Figure 6.9: Runtime for different problem sizes

Less than a percentage of the data falls outside this figure. As expected the selection method "PC6⊞2" gave the most accurate approximation of the travel time and distance between two locations, but also the other selection methods could be a good basis for the algorithm that makes use of precomputed routes. The standard deviations of the distributions in figure 6.10 are plotted in figure 6.11. This figure shows a strong correlation between the standard deviation in euro and the cell diameter in kilometres of each selection method. The fitted line $y = 0.20 \cdot x + 0.95$ is computed by using the MSE as metric for the goodness of the regression. However we do expect a near zero standard deviation when the diameter is also near zero, but more experiments are required to see this relationship for small grid cells.

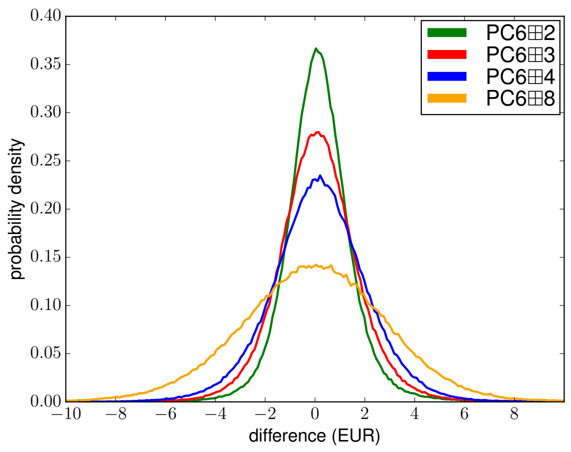


Figure 6.10: Quality of different selection methods

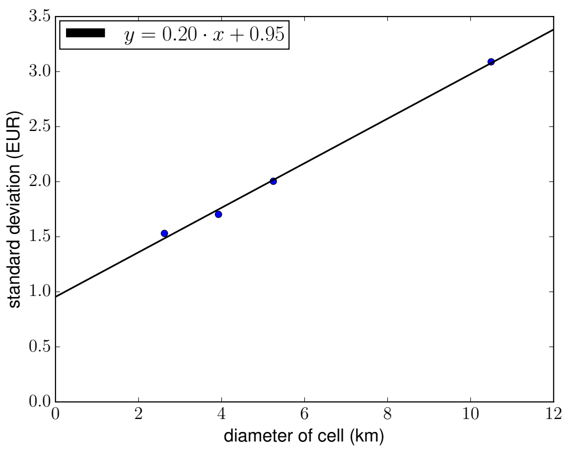


Figure 6.11: Correlation of standard deviation and cell diameter

6.5. Algorithm with precomputed routes

This section describes the empirical evaluation on the algorithm with precomputed routes, which is described in section 4.4. With this empirical evaluation we want to outline the trade-off that should be made while selecting the parameters δ_c and δ_w. Recall from section 4.4 that these parameters ensure that more solutions can be kept for reevaluation, which result in more computation time and better solution quality. In addition the runtime for various problem sizes is evaluated.

The first experiment considers a single objective variant of the problem, where the waiting time must be 0 and $f_c(x)$ is the only objective function. This first experiment is conducted with 500 taxi rides each assigned to a unique taxi company that has only one taxi. The experiment is repeated 200 times and the results are showed in figure 6.12. In this plot the average performance ratio is plotted for the various δ_c. The performance ratio

r is defined by equation 6.5, where $A(P)$ is the best solution found for problem P by algorithm A and $OPT(P)$ is the optimal solution found for problem P. Figure 6.12 shows that for all lookup tables the performance ratio converges to a certain value and on average the largest lookup table, which is denoted as “PC6⊞2” gave the best results. One should note that these results are a bit misleading, because one measurement can have a very big impact on the average performance ratio. This experiment shows that in very few cases a better solution can be found by using $\delta_c = 50$ instead of $\delta_c = 10$.

$$r = \frac{A(P)}{OPT(P)} \tag{6.5}$$

The runtime analysis on the algorithm with precomputed routes is performed in the same way as the evaluation described in section 6.2, but only the C++API of OSRM is used. So once more, a problem size of n means that there are n companies with each its own random base and one taxi. This time we also have to compare different settings of the algorithm. Since we now know what the range of useful values for δ_c is, namely from 0 trough 10, only $\delta_c \in \{0, 2, 5, 10\}$ are used. The results of this analysis on the multi-objective variant of the problem is stated in figure 6.13. This shows that the runtime of the algorithm tends to increase linear with the number of already scheduled taxi rides, but at a much slower rate than the naive algorithm. For small problems the runtime increases quicker than for large ones, this is caused by the way the number of reevaluated solutions increases. The computation of travel time and distance still dominates the total runtime of the algorithm.

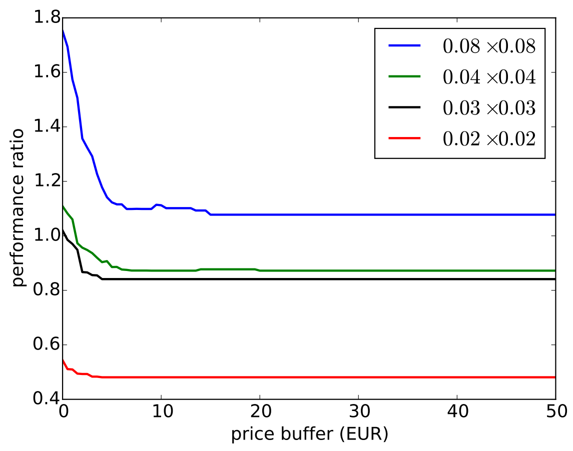


Figure 6.12: Performance ratio for various δ_c

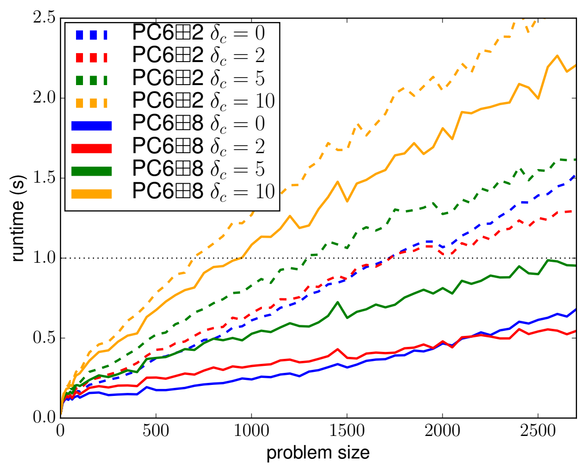


Figure 6.13: Runtime for various δ_c

The third experiment conducted on the algorithm is to measure the quality of the non-dominated set X produced by this algorithm with various δ_c. The quality is measured by two metrics, which both use the optimal non-dominated set $\bar{X}$ computed by the naive algorithm. The first metric expresses the number of solutions in X that are dominated by a solution in $\bar{X}$. This metric defined by $\mathcal{C}(X, \bar{X})$ is explained in more detail in subsection 3.3.2. The second metrics expresses the expected performance ratio of the algorithm for a customer with a random preference and this metric is defined by $\mathcal{Q}(X)/\mathcal{Q}(\bar{X})$. Note that also metric $\mathcal{Q}(X)$ is defined and explained in subsection 3.3.2. The expected performance ratio is computed by taking the average over 10000 random customer preferences, which consists of three weights one for the price and two for the offset to the requested departure time, because positive and negative offset is considered separately. The three weights are generated by drawing three numbers from a standard uniform distribution. For each problem instance new customer preferences are randomly generated.

The results for various δ_c, fixed $\delta_w = 0$ and 500 taxi rides each assigned to a unique taxi company that has only one taxi is stated in figures 6.14 and 6.15. These two figures show the averages over 200 randomly generated problems. From this experiment we can conclude that if we use the selection method PC6⊞2 and set $\delta_c = 0$ and $\delta_w = 0$ on average 0.3% of the solutions in the non-dominated set, produced by the algorithm with precomputed routes, is dominated by a solution from the optimal non-dominated set and for PC5⊞8 this is 3%.

The figures in this section show that more runtime cannot compensate for a smaller lookup tables with precomputed routes. This means that a larger lookup table is always worth it if accuracy is important and available memory allows this. The parameter δ_c should be set to relatively small values, for example to $\delta_c = 1$ should work well. However increasing δ_c comes with a price, because the computation time rapidly increases with higher values for δ_c. Note that the individual measurements of runtime varies a lot per instance. Therefore a runtime analysis on this algorithm gives more insight if the experiment is repeated more than just 40 times.

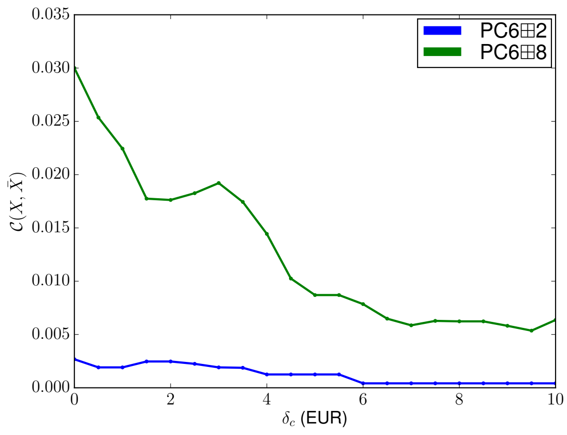


Figure 6.14: Metric $\mathcal{C}(X, \bar{X})$ for various δ_c and fixed $\delta_w = 0$

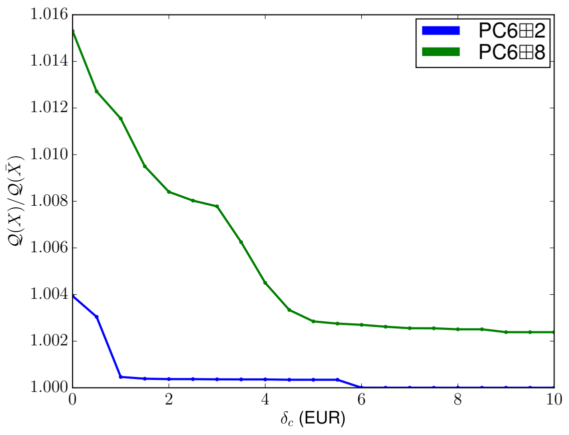


Figure 6.15: Approximated compared to optimal solution with metric $\mathcal{Q}$ for various δ_c and fixed $\delta_w = 0$

6.6. Expected cost

Recall from chapter 5 that a route consists of three segments, we are mostly interested in the expected cost of the first and last segment, and the customer is only during the second segment in the taxi. This section will describe the empirical evaluation done on the methods described in chapter 5. The experiments are only conducted on the first segment, because we can assume that the same results will apply for the last segment of a route.

First polynomial regression and just using the averages of the data are compared for the time feature for a fixed location, in subsection 6.6.1. Then four interpolation methods are compared for the spatial feature in subsection 6.6.2. All trained models have the same company base location, which is the Bay Plaza Shopping Center located at 40°52'01.2"N 73°49'49.8"W. Two sets of locations have been selected one containing 7629 locations, which are spread out as much as possible and are selected based on the locations in the green and yellow taxi data. The 1303 locations present in the second set are selected in the same way as the large set. Note that the intersection of both sets is empty. Gathering the data for all 7629 locations with the 1.4 million rides from May 2014 took roughly 60 hours using all four cores of the `Intel(R) Core(TM) i7-2600K CPU` running at 3.40GHz. The same amount of time is needed for the last segment and fitting the polynomials for all locations took less than 3 hours, which makes this setting applicable in the real world. For the small set of locations the taxi rides from May 2014 and the rides from June 2014 are used for validation.

6.6.1. Time feature

Recall from algorithm 6 stated in section 5.2 that first the averages of the lowest factors to pay of multiple days is computed before it is fed to the Levenberg-Marquardt algorithm (LMA). Preliminary research showed that this worked better than feeding all the data to LMA. The data itself is very scattered and because of that it is hard to find the best degree of polynomial to find the relationship between time of the day and factor to pay. Polynomial regression is compared with simply using the averages as model. Using only averages also speeds up computing the MSE during the execution of LMA. The degree of the polynomial is chosen separately per location. This is done by trying all degrees from 0 up to and including 14 and selecting the one with the lowest mean squared error. This was first done on a subset of the data and allowing degrees up to 20, but often a degree lower than 15 was selected. Often LMA was unable to find a fit at all with such a large number of parameters, which also significantly slowed down the whole experiment. Therefore the search space is limited to degrees below 15. The implementation of LMA from the SciPy library [29] is used. Note that overfitting, i.e. selecting a too high degree polynomial which will not generalize to similar data sets, is not likely to happen, because of the abundance of data points. Both models are evaluated in two experiments ways each with a different data set. First the leave-one-out cross-validation is used on the small set of locations with rides from May 2014. Second, a model is learned on the green taxi data from May 2014 with the small set of locations and evaluated with the green taxi data with rides from June 2014. The Boro taxis collectively conducted on average roughly 45900 and 44600 taxi rides per day in May and June, respectively.

Figures 6.16 and 6.17 show the average factor to pay over time for both the cost per minute and the cost per kilometre. The data points in the plot indicate averages of the factor to pay for the best match of the days May 1, 2014 until May 30, 2014 for the routes from the company's base to the intersection Wallstreet with Broadway (at 40°42'28.1"N 74°00'42.1"W) and to Terminal 1 of the John F. Kennedy International Airport (at 40°38'35.9"N 73°47'22.9"W). How a best match is found and defined is stated in section 5.2. This data is fitted with 8th degree polynomials, which is indicated with a solid line in the same figures.

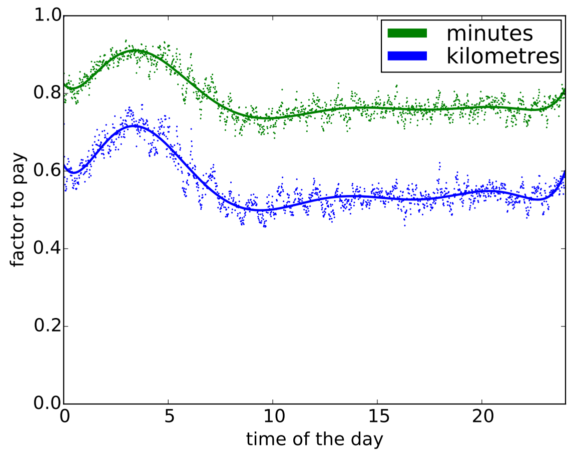


Figure 6.16: Factor to pay to the intersection Wallstreet with Broadway

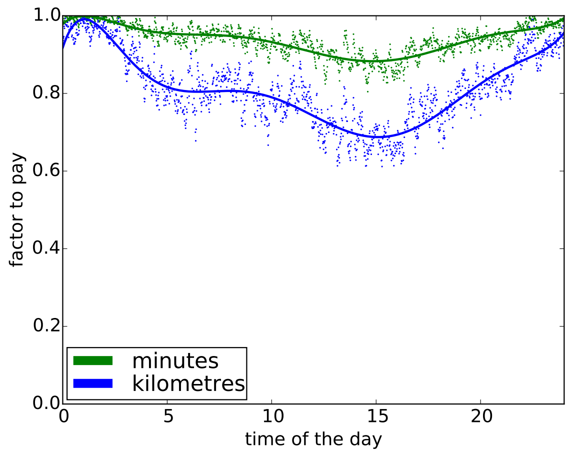


Figure 6.17: Factor to pay to Terminal 1 of the JFK Airport

The first experiment is to see if the model is able to generalize to other data in general. This experiment is done with data from May and the leave-on-out cross-validation method, which means that the average computed over all days except one and the excluded day is kept for validation. For the two models the MSE per day to the validation day is computed. The latter is repeated for each of the 1303 locations, which results in 1303×31 MSEs, because May consists of 31 days. The differences between the MSEs of both models, i.e. the MSE of the averages as model minus the MSE of the polynomial as model, are plotted in the histogram displayed by figure 6.18. Thus negative values are in favour of the polynomial as model and positive values are in favour of the averages as model. The second experiment is one that is closer to the real-world application and shows whether or not the model learned on data from one month can be used to predict the ratios to pay in the next month. This is done by using the data from May as training set and the days in June for validation. Again the MSE per day to the validation day is computed and repeated for all 1303 locations. Since there are 30 days in the month June this results in 1303×30 MSEs and again the differences between the MSEs of both models are plotted in a histogram. This histogram is showed in figure 6.19 and once more the negative values are in favour of the polynomial as model.

Please keep in mind that the domain of the two histograms is narrowed to make the figures more appealing, which caused very few measurements to be omitted from the histogram. Additionally all the values were multiplied with 100 before computing the MSE, which also made the histogram easier to comprehend. Both histograms show the worthiness of computing a high degree polynomial that fits the averages of the factor to pay. Note that an additional advantage of using polynomials as model is that it does not require much memory to store, in contrast to using the averages as model. The high degree polynomial did work better, due to noise reduction obtained by computing the polynomial.

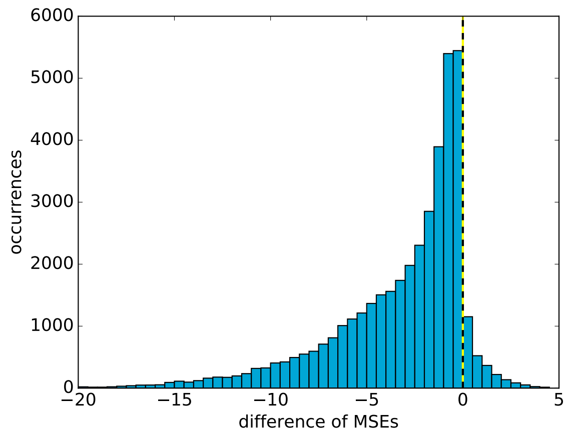


Figure 6.18: Differences of MSEs for May

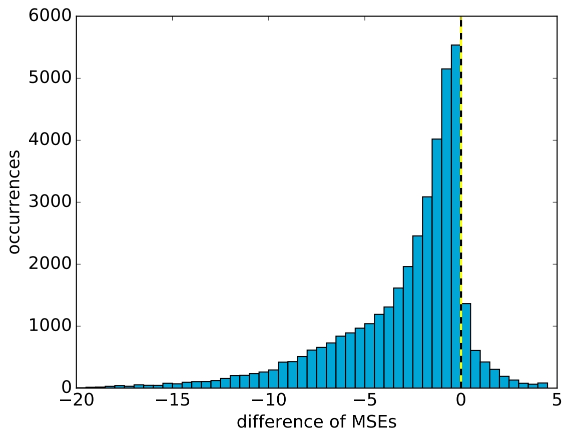


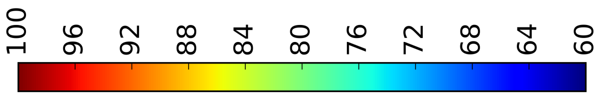
Figure 6.19: Differences of MSEs for June

6.6.2. Spatial feature

The previous section compared methods to predict the ratio to pay for a given time at a sampled location based. This section will compare methods to predict the ratio to pay for an unsampled location. The best method can be used for the implementation of algorithm 7 stated in section 5.2. We only consider the interpolation methods nearest-neighbour, linear, cubic and inverse distance weighting interpolation (IDW) for the experiments described in this section. Other interpolation methods such as Kriging, spline interpolation and adaptive inverse distance weighting, but also regression techniques are harder to implement and to apply, because they require tuning multiple hyperparameters. We did try to tune these hyperparameters for multiple regression, kernel regression and multilayer perceptron with the implementations from the scikit-learn library [48]. However the best models obtained not even came the interpolation methods listed in this section. Therefore regression is omitted from the comparison and left open for future work.

The factor to pay to each destination at 15:00 for different interpolation methods is plotted in figures 6.20, 6.21, 6.22 and 6.23. This time was chosen for the example, because then the most diversity between locations can be observed. The grey coloured area indicates unknown values. There are unknown values, because linear and cubic interpolation cannot extrapolate. The cyan coloured area indicates water, but note that not all rivers are indicated. The cubic interpolation method uses only first-order derivatives and the curvature of the interpolating surface is minimized. For both linear interpolation and cubic interpolation the implementation from the SciPy library [29] is used. For the implementation of IDW and nearest-neighbour the k-d tree implementation from the same library is used, because only the nearest eight measurements are evaluated while interpolating. For the example in figure 6.21 the IDW decay parameter $p = 3$ is chosen. The data for the plots are obtained by computing the average factor to pay for kilometres at 15:00 on all days in May 2014 for every location in the large set of locations. Then for each interpolation method a grid of 500 by 500 tiny rectangles is coloured based on the predicted value. These plots show a complex relation between location and expected cost, which indicates that applying regression is non-trivial. Additionally the non-smooth plot with nearest-neighbour Interpolation suggests that this method will not give accurate predictions. Since IDW has only one hyperparameter, namely the decaying parameter p, that allows tuning the model we expect that this method works the best.

For the performance evaluation of IDW multiple values for decaying parameter p are assessed, namely $p \in [0, 15]$. For each day of the month and every 15 minutes of the day the squared error between the predicted and actual value is computed for all 1303 locations. The MSE is for IDW with various values for p is reported in figure 6.24. This shows that for $p \to \infty$ the best results will be obtained, but already $p = 8$ is good enough to compare with the other three interpolation methods. The same experiment is done for these other three and the results are stated in the bar chart in figure 6.25. From this bar chart we have to conclude nearest-neighbour interpolation had the best predictive power. This is supported by the fact that higher values for p worked better and will be more similar to nearest-neighbour interpolation than linear and cubic interpolation.



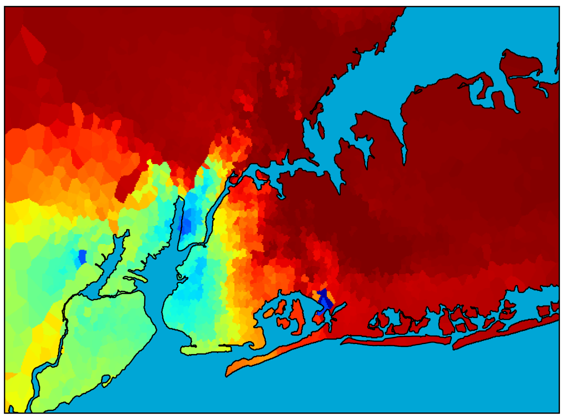


Figure 6.20: Nearest-neighbours Interpolation

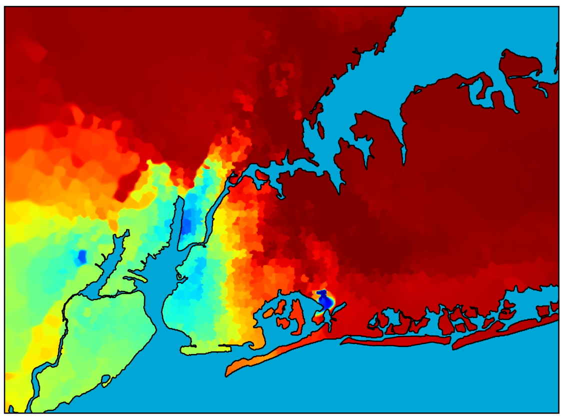


Figure 6.21: Inverse Distance Weighting ($p = 8$)

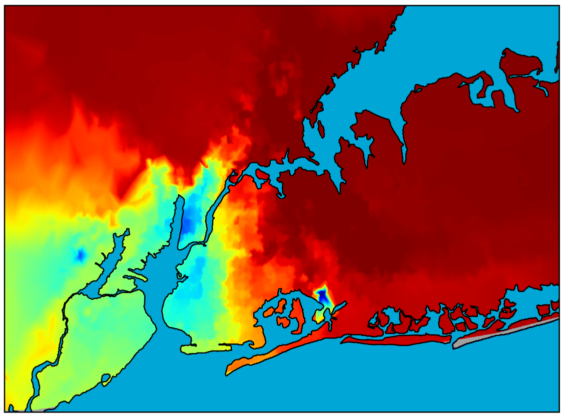


Figure 6.22: Linear Interpolation

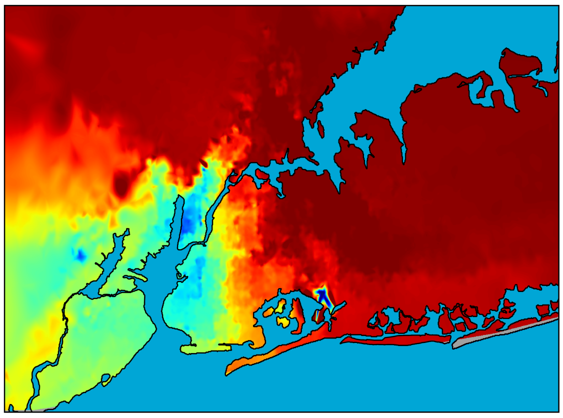


Figure 6.23: Cubic Interpolation

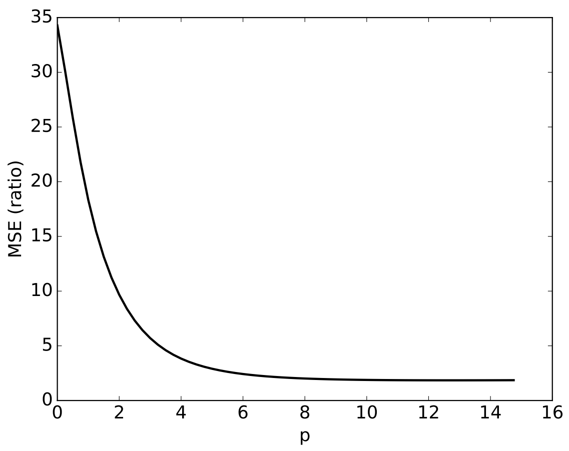


Figure 6.24: MSE for different parameters p for IDW

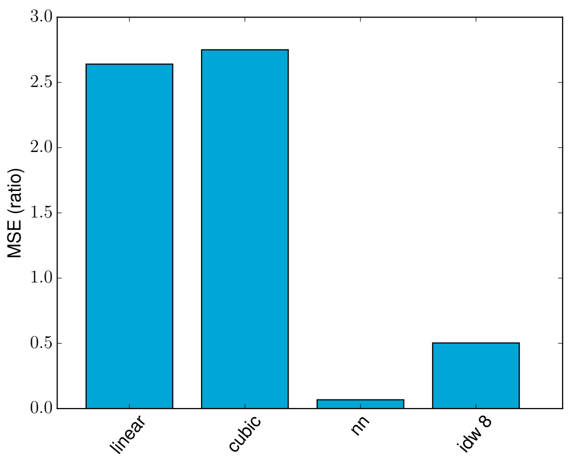


Figure 6.25: MSE of different interpolation methods

6.7. Risk aversion

This section will demonstrate the working of the adjustment proposed in section 5.3 by using two examples. The two examples are the two segments from company's base to JFK Airport and to the intersection Wallstreet with Broadway, which we will refer to as 'JFK segment' and 'Wallstreet segment', respectively. This section demonstrates a simulation of various risk attitudes with the four methods that are described in section 5.3.

The histograms in figures 6.26 and 6.27 show, for the JFK segment and the Wallstreet segment, the cost distribution when only roughly 50% of the 45900 rides is carried out by the taxi company. These are measurements with all 31 days of May 2014 repeated 20 times, each time with a different random subset of the rides, where each taxi ride had a 50% probability to be in the subset. The black and yellow dashed line indicates the unweighted average of the observations. These figures will help to better understand the other plots in this section.

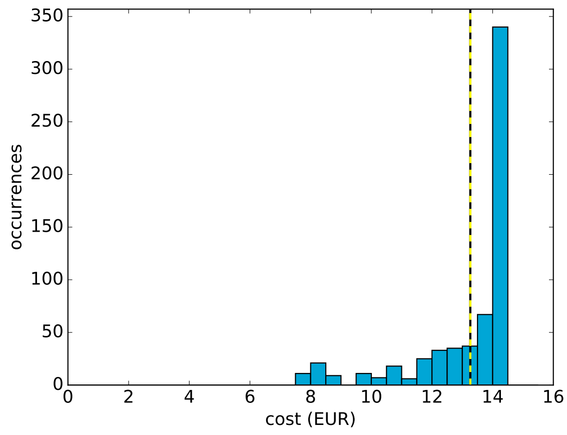


Figure 6.26: Cost distribution to JFK

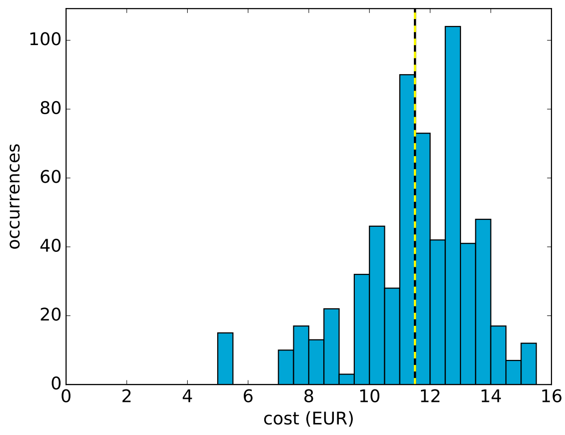


Figure 6.27: Cost distribution to Wallstreet

Figures 6.28 and 6.29 show for each of the four methods the relation between the risk attitude parameter α and the price for the example with 50% of the rides in May, 2014. Please recall that the second method limits the maximum allowable loss, which is set to α times the standard price in this example. The black dotted lines indicate both the expected cost and the worst case scenario. For the fourth method, which is the weighted average $\bar{x}_w$, the weight function $w_i = \alpha\frac{i}{n} + 1 - \alpha$ is used. Although method g_1 is very simple it has a nice property too, because its behaviour is very predictable and easy to understand. However it is vulnerable to outliers, because it depends a lot on scenario x_n. Also g_2 is vulnerable to outliers and also selecting the maximum allowable loss is a non-trivial task. This is showed by figures 6.28 and 6.29, because the price scales a bit weird.

Figures 6.30 and 6.31 show the prices for the four methods with $\alpha = \frac{1}{2}$ and for a different percentages of the total number of taxi rides. Again the black dotted lines indicate both the expected cost and the worst case scenario. Note that $\alpha = \frac{1}{2}$ indicates that the price should be somewhere in the middle between the expected cost and worst case scenario, but method g_2 always asks the expected cost as price in the two examples. Our opinion is that method g_3 is the best of the four, because it scales nicely with α and uses the distribution of the measurements.

Figures 6.32 and 6.33 linear and nonlinear weighting functions attitudes are compared for the weighted average method. The setting $w_i = 1$ is the same as the unweighted average and thus the unmodified version of the algorithm described in chapter 5. This shows that using nonlinear weight functions makes it possible to simulate more risk averse behaviour, than the easier to understand function $w_i = \alpha\frac{i}{n} + 1 - \alpha$. However these functions still do not scale towards a complete risk averse behaviour. Therefore we would not recommend using this method to averse risk.

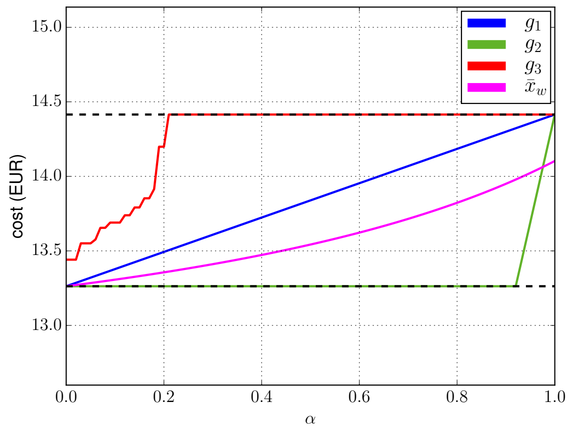


Figure 6.28: Risk attitude comparison for various α for the JFK segment

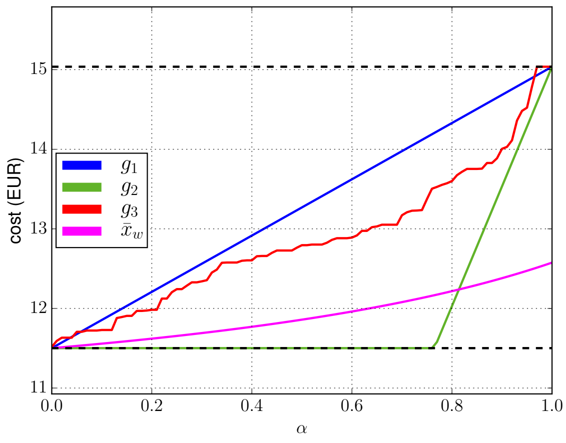


Figure 6.29: Risk attitude comparison for various α for the Wallstreet segment

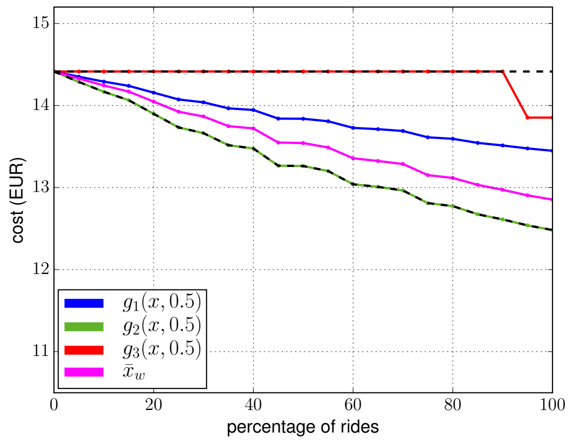


Figure 6.30: Risk attitude comparison for various number of rides for the JFK segment

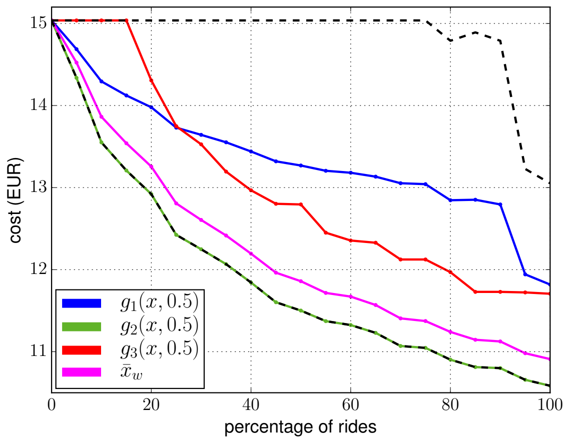


Figure 6.31: Risk attitude comparison for various number of rides for the Wallstreet segment

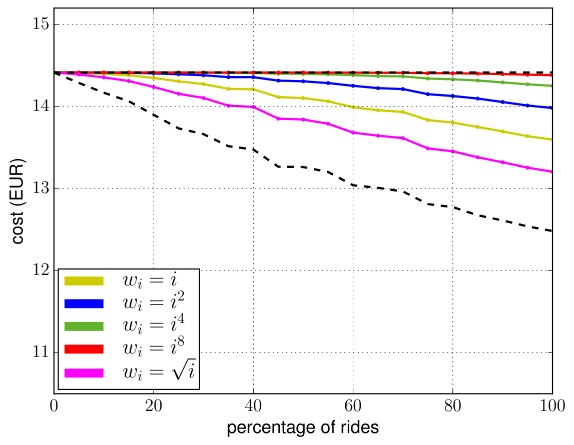


Figure 6.32: Nonlinear weighted average risk attitude comparison for the JFK segment

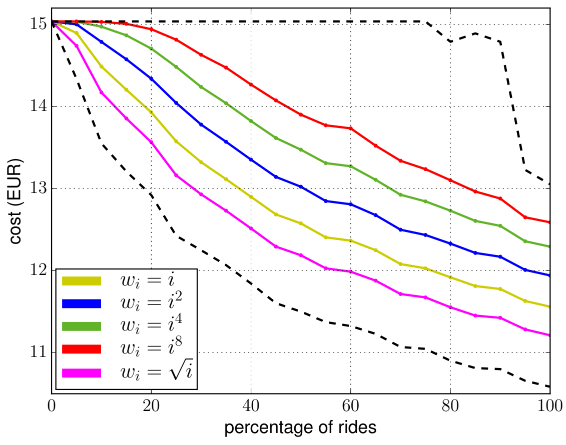


Figure 6.33: Nonlinear weighted average risk attitude comparison for the Wallstreet segment

7 Future work & conclusion

In this chapter future work is described, which gives insight into how this thesis could have been improved. In addition some of the assumptions and conclusions are discussed.

Chapters 1 and 2 described the need for an algorithm that can compose offers based upon previously booked taxi rides within one second. Chapter 4 described three algorithms, namely the naive algorithm, the parallel algorithm and the algorithm with precomputed routes. If the one second computation time constraint is respected, the naive algorithm can only deal with a maximum of 20 taxi rides in the worst-case scenario, which is in practice too little. The parallel algorithm has a speedup nearly the same as the number of processors. The computation of the routes has a large impact on the runtime of the three algorithms. Therefore the algorithm with precomputed routes was proposed and was able to deal with up to 3200 pre-existing taxi rides. In addition a speedup of nearly the number of processors can be expected for a parallel version of the algorithm with precomputed routes. The solutions were only slightly worse than the solutions computed by the naive algorithm, thus this algorithm is good enough for a real-world setting. However this algorithm also comes with a few disadvantages, namely a large lookup table that costs a lot of computation time to build and a significant amount of memory to store. Accounting the uncertainty of travel time, caused by traffic jams, during the cost calculation becomes harder when making use of a lookup table to approximate route distances. Another improvement of these algorithms can possibly be obtained by an unfinished feature of the OpenStreetMap Routing API, which makes it possible to group certain API requests. However this currently only computes the distance between two locations and not the travel time. By grouping API requests the overhead could be reduced, but it is uncertain how much this will effect the runtime.

The parallel algorithm is a great improvement upon the naive algorithm, but most likely an even greater speedup can be achieved when decentralizing the system, because these systems tend to have more computing power. Decentralizing the system means that every taxi company has its own computer trying to match incoming taxi ride requests with their existing pool of rides and respond with an offer to the customer via the Gogido platform. A decentralized system comes with a significant advantage, because when each company has its own computer the complete distributed system has a lot of computing power that can be used efficiently. Only little overhead caused by networking is expected. A decentralized system has more significant advantages, because the companies' critical information can be kept secret and every company can have his own way of bidding on a taxi ride. Thus in the perfect world each company has its own system that communicates with the Gogido platform and automatically bid on incoming taxi ride requests. In this way each company can choose a strategy that is used by their system. Also each company should have at any time a schedule, which can be improved by local search procedures in between bookings as described in literature [15].

In the current problem setting it is assumed that every taxi company has only one base location. This could be easily extended to multiple base locations, but this also introduces new research questions. When always the best base location is selected, the customer can be offered the largest discount. However, one base location could be become very popular and the company's taxis become poorly distributed over the area of operation.

At the end of chapter 4 the problem of many similar offers is discussed. This could be a problem, because a customer could find it hard to digest a large number of offers and make a choice between them. A new metric is proposed that should make it easier for an algorithm to find the most distributed subset of offers with a given number of elements. How a list of non-dominated solutions can be reduced to a specific size, such that similar solutions are discarded, is still an open problem. However we did build a foundation for a solution with a new metric in section 4.5. This metric can be used in a local search algorithm that starts with a random subset and swaps solutions to improve the quality of the subset. Another non-exact algorithm that could also work very well, repeatedly removes one solution until the desired size is reached. Note that an exact solution is infeasible due to exponential number of subsets. In addition any algorithm should be very quick, because not only the non-dominated set, but also the reduced version must be computed within one second. It might therefore be better to discard solutions already before reevaluation by the algorithm with precomputed routes.

Chapters 1 and 2 also described the need for a method to calculate the expected cost of a taxi ride. This can be used to acquire even lower prices, more customers and more revenue. The method proposed in chapter 5 simulates the past month per day to predict for each minute of the day the expected cost for a number of selected locations by computed the cost for the best match that day. Using the best match as prediction might be to optimistic especially when multiple similar rides come in, because of this low price. For each location a high degree polynomial is computed which maps the time of the day to the ratio of the standard price that needs to be paid. By using a high degree polynomial the amount of memory needed is reduced and the predictions become more accurate, because noise is removed. In section 6.6 four interpolation methods were compared which make it possible to predict the cost for unsampled locations. To our surprise the nearest-neighbour interpolation was the best method and inverse distance weighting interpolation was second best. This shows promise for the adaptive inverse distance weighting interpolation method. This method together with other regression methods such as multiple regression and multilayer perceptron is left open for future work. With the latter two it could be possible to build one model in theory, so it is not needed to compute the high degree polynomials for the time and price relation.

Since the expected cost could be lower than the actual cost, this introduces a risk for the companies. Section 5.3 described methods that make it possible for a company to increase or lower the risk it is willing to take with one simple parameter. The extra money earned by this price increase could be used to combine chapters 4 and 5, which means that both the concept of asking the customer the expected cost and offering the customer an extra discount when its taxi ride request can be matched by already accepted rides. However the price increase as suggested in section 5.3 does not take into account the behaviour of the customer, because he could choose for another taxi company if the price is too high. Less customers leads to a higher expected cost and thus less risk is avoided than initially indented. This suggests the need for a method that can increase the price to partly cover risk and at the same time takes both the behaviour of a competing taxi company and the customer into account.

All of the methods described in chapter 5 can be easily used for other countries and cities, thus also for The Netherlands. No simulation has been done with the constructed dataset for The Netherlands to see what this would mean for the Dutch taxi market. This is partly due to the reasons described earlier, but also because this generated dataset has some flaws as well. Firstly in the real world it is not likely that people want to travel from anywhere to anywhere, some nonpopulated areas such as Schiphol are very popular locations for taxis as well. The model could also be extended with information about income, car ownership and public transport accessibility, but it is unknown how much this will influence the use of taxis. If the method for computing the expected cost is also extended with the other features listed in section 5.1 where necessary and the problems discussed in this chapter are solved, it can play an important role in the Dutch taxi market.

The methods proposed in this thesis can be applied to other transport sectors such as freight transportation and possibly in the future shared self-driving vehicles. However we did not see other possible applications for the proposed methods outside transport businesses.

Bibliography

[1] Tabel van nederlandse gemeenten. URL `https://nl.wikipedia.org/wiki/Tabel_van_Nederlandse_gemeenten`.

[2] Postcode data. URL `http://www.postcodedata.nl`.

[3] IBM ILOG CPLEX Optimizer. urlhttp://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, Last 2010.

[4] MA Abido. Multiobjective particle swarm optimization for environmental/economic dispatch problem. *Electric Power Systems Research*, 79(7):1105–1113, 2009.

[5] Hiroshi Akima. A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Transactions on Mathematical Software (TOMS)*, 4(2):148–159, 1978.

[6] John W Baugh JR, Gopala Krishna Reddy Kakivaya, and John R Stone. Intractability of the dial-a-ride problem and a multiobjective solution using simulated annealing. *Engineering Optimization*, 30(2): 91–123, 1998.

[7] James C Bezdek, Robert Ehrlich, and William Full. Fcm: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2):191–203, 1984.

[8] Christian Blum and Max Manfrin. Metaheuristics network. `http://www.metaheuristics.net/index.php%3Fmain=1.html`. Accessed: 2010-09-12.

[9] Lawrence D Bodin and Samuel J Kursh. A computer-assisted system for the routing and scheduling of street sweepers. *Operations Research*, 26(4):525–537, 1978.

[10] Lorenzo Bracciale, Marco Bonola, Pierpaolo Loreti, Giuseppe Bianchi, Raul Amici, and Antonello Rabuffi. CRAWDAD dataset roma/taxi (v. 2014-07-17). Downloaded from http://crawdad.org/roma/taxi/20140717, July 2014.

[11] Han-wen Chang, Yu-chin Tai, and Jane Yung-jen Hsu. Context-aware taxi demand hotspots prediction. *International Journal of Business Intelligence and Data Mining*, 5(1):3–18, 2009.

[12] Carlos A Coello Coello, Gregorio Toscano Pulido, and M Salazar Lechuga. Handling multiple objectives with particle swarm optimization. *Evolutionary Computation, IEEE Transactions on*, 8(3):256–279, 2004.

[13] Jean-Francois Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573–586, 2006.

[14] 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.

[15] Luca Coslovich, Raffaele Pesenti, and Walter Ukovich. A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem. *European Journal of Operational Research*, 175(3): 1605–1615, 2006.

[16] Kalyanmoy Deb. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary computation*, 7(3):205–230, 1999.

[17] Google Developers. Google Maps Directions API. URL `https://developers.google.com/maps/documentation/directions`. Accessed: 2016-03-28.

[18] Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V Vinay. Clustering large graphs via the singular value decomposition. *Machine learning*, 56(1-3):9–33, 2004.

[19] Henrdk Esbensen and Ernest S Kuh. Design space exploration using the genetic algorithm. In *Circuits and Systems, 1996. ISCAS'96., Connecting the World., 1996 IEEE International Symposium on*, volume 4, pages 500–503. IEEE, 1996.

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[21] Carlos M Fonseca and Peter J Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary computation*, 3(1):1–16, 1995.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

[23] David E Goldberg et al. *Genetic algorithms in search optimization and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989.

[24] Bruce Golden, Lawrence Bodin, T Doyle, and W Stewart Jr. Approximate traveling salesman algorithms. *Operations research*, 28(3-part-ii):694–711, 1980.

[25] Michael Pilegaard Hansen. Tabu search for multiobjective optimization: Mots. In *Proceedings of the 13th International Conference on Multiple Criteria Decision Making*, pages 574–586. Citeseer, 1997.

[26] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[27] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[28] Jang-Jei Jaw, Amedeo R Odoni, Harilaos N Psaraftis, and Nigel HM 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.

[29] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`. [Online; accessed 2016-06-12].

[30] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[31] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992.

[32] Kenneth Levenberg. A method for the solution of certain non–linear problems in least squares. 1944.

[33] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[34] George Y Lu and David W Wong. An adaptive inverse-distance weighting spatial interpolation technique. *Computers & Geosciences*, 34(9):1044–1055, 2008.

[35] Dennis Luxen and Christian Vetter. Real-time routing with openstreetmap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 513–516, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1031-4. doi: 10.1145/2093973.2094062. URL `http://doi.acm.org/10.1145/2093973.2094062`.

[36] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[37] Oli BG 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.

[38] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. In *WALCOM: Algorithms and Computation*, pages 274–285. Springer, 2009.

[39] Kantilal Varichand Mardia, John T Kent, and John M Bibby. *Multivariate analysis*. Academic press, 1979.

[40] Bruce McCune. Nonparametric multiplicative regression for habitat modeling. *MjM Software, Gleneden Beach, Oregon, USA. Online at http://www. pcord. com/NPMRintro. pdf*, 2004.

[41] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[42] Lubos Mitas and Helena Mitasova. Spatial interpolation. *Geographical information systems: principles, techniques, management and applications*, 1:481–492, 1999.

[43] Luis Moreira-Matias, João Gama, Michel Ferreira, and Luis Damas. A predictive model for the passenger demand on a taxi network. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 1014–1019. IEEE, 2012.

[44] Fionn Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[45] Marialisa Nigro, Livia Mannini, and Marta Flamini. A clustering first–route second method for the solution of many-to-many dial a ride problem. *Proceedings of the 5th International Conference on Applied Economics, Business and Development (AEBD '13)*, pages 464–468, 2013.

[46] Ibrahim H Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[47] Sophie N Parragh, K Doerner, and Richard F Hartl. A survey on pickup and delivery models part ii: Transportation between pickup and delivery locations. Technical report, 2006.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[49] Santi Phithakkitnukoon, Marco Veloso, Carlos Bento, Assaf Biderman, and Carlo Ratti. Taxi-aware map: Identifying and predicting vacant taxis in the city. In *Ambient Intelligence*, pages 86–95. Springer, 2010.

[50] Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. CRAWDAD dataset epfl/mobility (v. 2009-02-24). Downloaded from http://crawdad.org/epfl/mobility/20090224, February 2009.

[51] Weda Jarst Poort Joost. Toekomst voor de taxi. Technical report, SEO Economisch Onderzoek, 2008.

[52] Harilaos N Psaraftis. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*, 14(2):130–154, 1980.

[53] Harilaos N Psaraftis. Analysis of an $O(N^2)$ heuristic for the single vehicle many-to-many euclidean dial-a-ride problem. *Transportation Research Part B: Methodological*, 17(2):133–145, 1983.

[54] Harilaos N Psaraftis. An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows. *Transportation Science*, 17(3):351–357, 1983.

[55] K. B. Bergvinsdottir R. M. Jorgensen, J. Larsen. Solving the dial-a-ride problem using genetic algorithms. *The Journal of the Operational Research Society*, 58(10):1321–1331, 2007.

[56] Rijksoverheid.nl. Wat zijn de tarieven voor een taxi? URL `https://www.rijksoverheid.nl/onderwerpen/taxi/vraag-en-antwoord/wat-zijn-de-kosten-voor-een-taxi`. Accessed: March 27, 2015.

[57] Stan Salvador and Philip Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 576–584. IEEE, 2004.

[58] George AF Seber and Alan J Lee. *Linear regression analysis*, volume 936. John Wiley & Sons, 2012.

[59] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524. ACM, 1968.

[60] Robin Sibson et al. A brief description of natural neighbour interpolation. *Interpreting multivariate data*, 21:21–36, 1981.

[61] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[62] Thomas G Stützle. *Local search algorithms for combinatorial problems: analysis, improvements, and new applications*, volume 220. Infix Sankt Augustin, Germany, 1999.

[63] Rienstra Sytze, Bakker Peter, and Visser Johan. International comparison of taxi regulations and uber. 2015. URL `http://www.kimnet.nl/sites/kimnet.nl/files/international-comparison-of-taxi-regulations-and-uber.pdf`.

[64] Taxi and Limousine Commission. NYC Boro Taxi Trips 2013/2014, . URL `http://chriswhong.com/nycborotaxidata`.

[65] Taxi and Limousine Commission. NYC Taxi Trips 2013, . URL `http://www.andresmh.com/nyctaxitrips`.

[66] Paolo Toth and Daniele Vigo. *Vehicle routing: problems, methods, and applications*, volume 18. Siam, 2014.

[67] Taxi Service Trajectory. Porto Taxi Data. urlhttp://www.geolink.pt/ecmlpkdd2015-challenge/dataset.html. Accessed: 2016-07-03.

[68] Stefan Voß, Ibrahim H Osman, and Catherine Roucairol. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, 1999.

[69] J Weda, J Poort, et al. De prijs van een taxirit: het effect van lokale factoren en marktfalen. *Tijdschrift Vervoerswetenschap*, 44(3):88–97, 2008.

[70] Rui Xu, Donald Wunsch, et al. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, 2005.

[71] David Yassky and Michael R Bloomberg. *2014 Taxicab Factbook*. New York City Taxi & Limousine Commision, 2014. URL `http://www.nyc.gov/html/tlc/downloads/pdf/2014_taxicab_fact_book.pdf`.

[72] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, 8(2):173–195, 2000.

"History may not repeat itself, but it does rhyme a lot."- Mark Twain (1970)