IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

BENG COMPUTING INDIVIDUAL PROJECT

Fleet Management in On-demand Transportation Networks : Using a Greedy Approach

Author:
Filip Stollar

Supervisor:
Prof. Julie McCann

Second Marker:
Dr. Robert Chatley

June, 2018

Abstract

On-demand transportation networks such as Uber or Lyft provide an effective and cheap mode of transportation which see an ever-growing popularity. A central component in these networks are fleet managers, systems responsible for solving the problem of matching drivers with passengers. The goal of this project is to investigate, implement and evaluate various scheduling methods used in fleet managers, covering areas such as exclusive taxi where only one passenger can be present at a time or a shared taxi with multiple passengers at once. In particular, the contributions include an implementation of the Insertion Heuristic algorithm with a proposed optimization which leverages the fleet manager's knowledge of a driver's queue of passengers. Further, we implement the Aggregated Greedy Dispatch algorithm while suggesting several possible enhancements which use the fact that the fleet manager has a perfect knowledge about the passengers requests and driver locations. Finally, we design and devise the last scheduling approach which proposes an unconventional way of matching drivers to passengers. In this approach, we periodically spawn 'gravitational points' in an area, which act as locations where drivers can move in order to pick-up more passengers and maximize the profits.

All of the approaches are evaluated with an aim of greedily improving a defined set of metrics which directly affect the performance of a fleet manager. The implementations achieve superior results on a variety of metrics, in comparison to currently widely used approaches. The reasons and analyses of why the evaluated results of our implementations are superior (or in certain cases inferior) are in detail provided in this report. In addition to the implemented methods, the project also delivers two developed applications which were essential during the process of evaluation.

Acknowledgements

I would like to thank my supervisor, Prof Julie McCann, for her help and support throughout this project. Her insights and suggestions have been of great importance, for which I am extremely grateful.

Further, I would like to thank my family, friends and my girlfriend Zuzana, all of whom are a part of the project thanks to their unconditional love and support, which has aided me not only during the writing of the thesis, but during the whole three year journey at Imperial.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

With nearly half a billion users worldwide [1], ride-hailing services are undoubtedly introducing a paradigm shift in the way we think about transportation. We are able to request a vehicle which arrives nearly instantly, we have a transparent view of the driver's location and information about the car and thanks to sharing economy, it is a self-regulating market and the passengers are able to save money on travelling [2]. Furthermore, besides the obvious negative environmental implications, owning a car or using old-fashioned taxi companies is more expensive [3] and increases the total number of cars within urban areas which is already enormous to the extent that they are being banned in certain cities [4]. Thanks to ride-hailing application, the number of drunk driving accidents has significantly decreased [5], simply because they are so accessible.

Within the space of ride-hailing systems, in the field of computing there are multiple problems being discussed, such as geolocation services, distributed systems or one of the key ones being fleet managers. Fleet managers are concerned with the problem of matching drivers to passengers. More specifically, they solve a special case of the Vehicle Routing Problem (VRP), an NP-hard class problem, which we are discussing in more detail in the next sections. The main reason for it's uniqueness is that the fleet management happens in terms of a dynamic ride-hailing environment, on a massive scale (hundreds of thousand passengers per day) where every single passenger wants to be matched as soon as possible. It is a relatively recent topic which is gaining popularity thanks to successful commercial uses in applications such as Uber [6] or Lyft [7].

A Fleet manager is a valuable intellectual property and many times we can only observe and approximate the algorithms and optimizations used by the larger companies to manage their large fleets of drivers. As an example, Lyft claims that they match their passengers with the closest available driver [8], which we will in later sections explore as the most basic heuristic for fleet management. The technique is called Nearest Vehicle Dispatch (NVD) and in our evaluation it turns out to be surprisingly effective. It can be considered to be the industry standard because of how easy it is to implement and scale up [9].

Further insight into why the specific approach is used is missing. As we will observe, there are optimizations which build on top of NVD and introduce significant improvements to our evaluated metrics. The improvements are caused by the massive scales of the fleet managers, where even small incremental changes in the way companies conduct their fleet management add up and have a far-reaching impact. One of the benefits of more advanced fleet managers is increased environmental sustainability. As an example of a modern urban city, in San Francisco alone, reducing the number of miles travelled within on-demand means of transport just by 3% would save 500 thousand miles travelled monthly [10]. The potential in such savings is that they are instant and it can be many times only a matter of a minor engineering effort. However, there is only a limited set of resources which would provide implementations and analyses of current on-demand approaches and their possible extensions, which we further discuss in Section 2.3.

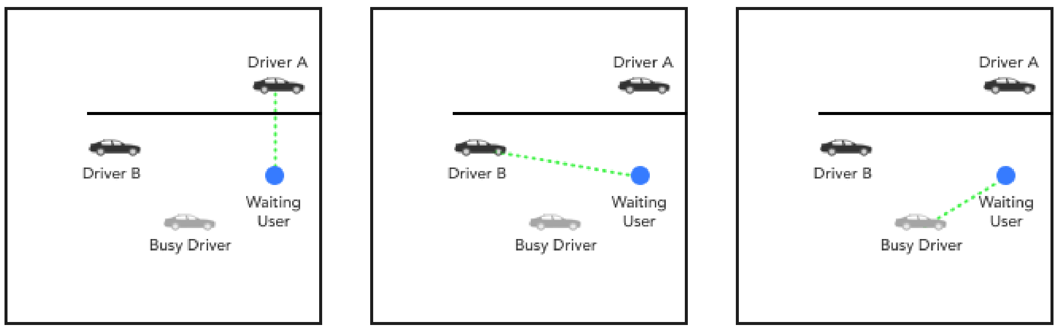


Figure 1.1: Example of different decisions a scheduler can make. Should it pick the closest driver based on the straight line distance, which saves us computational time as we don't need to run a routing algorithm? Or pick the closest available driver, but risking that the busy driver might be dropping off it's customer within the next few moments? Also, it might be the case that we are unable to consider all of our drivers, simply because there is too many of them.

Few sources discussing on-demand scheduling contribute to the reason why the global taxi market is resisting the change of having their drivers operated by a software, while the drivers continue working with decades-old methods involving communication by walkie-talkies and the fleet management being done by a team of dispatchers. This creates a friction between regular taxi companies and ride-hailing applications, since the taxi companies are not able to compete with the convenience and costs ride-hailing services offer. Yet, the trend is inevitable. Ride-hailing is overtaking the customers of regular taxi companies [11]. Uber and Lyft are the biggest players which together account for 65% of the market [12]. Based on the previous statistic of 500 million users, this still leaves us with roughly 175 millions of users scattered among the rest of the ride-hailing services. Usually, they operate at a much smaller scale compared to the big players and are not able to invest into their fleet management which remains basic. And what if an old-fashioned taxi company decides to undergo the dreaded change of becoming almost fully automated? In the majority of cases we would most possibly find a certain version of the basic NVD implemented.

As we will find out, just the NVD itself can be extended with several optimizations, each suitable for a different scenario which can happen during the scheduling process, ultimately saving resources for all participating parties in the fleet manager. Further, the performance of a fleet manager can be improved by techniques not directly related to matching drivers with passengers, one of which we devise and explore in Section 3.2.

1.2 Objectives

The goal of the project is to implement and analyze a set of methods, determine how they perform under various circumstances and establish which yield the best performance on a defined set of metrics in a large scale simulation. The metrics will remain consistent throughout the project. Namely, they are **Passengers Lost, Average Waiting Time, Total Distance Travelled** and each can be directly linked to the effectiveness of a fleet manager, where 'effectiveness' is left to be interpreted by a third party (usually party operating the fleet manager). It might be , for example, that some fleet managers benefit more from lower average waiting times at the expense of more distance travelled or vice-versa. A passenger can be lost if we exceed a time limit of 20 minutes, which is justified in Section 2.1.2.

If we would want to find the absolutely best approach which gives the best possible performance on the defined set of metrics, we would have to find the given problem's global optimum. With the current methods, such as Linear Programming, this might be possible, but it would take a large amount of time, a resource which is not available to a greedy on-demand system where every second counts. This narrows down our search scope, but there still remains a large set of techniques which allow us to find less optimal local optimums, at the benefit of an acceptable computing time.

This project focuses solely on greedy approaches and how they conduct fleet management. The first reason is simply that considering all of the possible techniques would be infeasible, given the time scope of the assignment. We will be looking into some of them, such as Ant Colony Systems (ACS) or the mentioned Linear Programming, but there are many more. Secondly, greedy approaches are used because they are native to an on-demand scheduling environment. It should be noted that greedy in this case does not mean picking the first match which satisfies our constraint, but rather picking a match which is optimal at this point of time, without considering the long-term consequences as doing so would render many times computationally infeasible. Every passengers is matched with the closest driver, without planning further into future, which can long term harm the effectiveness. This is exactly as the mentioned Nearest Vehicle Dispatch (NVD) behaves, it picks the closest driver to the user at this point of time. However, devising greedy extensions and algorithms is extremely challenging, especially when solving NP-hard tasks such as the Vehicle Routing Problem (VRP) on a massive scale of thousands of passengers and drivers.

1.3 Contributions

The project's objective is to explore and benchmark a range of greedy scheduling methods which include a subset of implementations based on resources provided by external sources or research papers. The source of original implementation is always stated, alongside of any modifications or extensions made for an on-demand ride hailing environment. The other subset includes methods which started with the basics such as the Insertion Heuristic (in Section 3.1) or Aggregated Greedy Dispatch (in Section 3.2) and were incrementally modified and optimized, usually based on observations in the Evaluation section. The main contributions consist of :

- **Scheduling of an Exclusive taxi** - In order to be able to evaluate the scheduling of taxi where a single passenger can be present at a time, I have implemented the Nearest Vehicle Dispatch, Insertion Heuristic and Optimized Insertion Heuristic algorithms. The details and challenges, such as optimizing a queue of passengers in the Insertion Heuristic are described in Section 3.1.

- **Scheduling of a Shared taxi** - For scheduling a taxi where multiple passengers can be present at a time I have implemented the AGD and DAGD algorithms alongside with their optimizations. This contribution was inspired by Lyft Engineering [13], which lacked a more detailed evaluation and reasoning about the behaviour of the algorithms. This introduced a challenge as many times I had to make important decisions regarding the design and logic of the AGD. The whole implementation process is described in 3.2.

- **Scheduling done by Gravitational Points** - Gravitational points are a way of distributing our fleet of drivers throughout the city in order to maximize our profits in the near future. I consider the gravitational points to be the biggest contribution of the project, because it takes a different look at fleet management and demonstrates positive results throughout the analysis. In Section 3.3, I have proposed 2 possible ways of spawning these points, one based on statistical analysis of past trends and the other based on predictions produced by LSTM networks.

- **Fleet management simulator** - Throughout the project, I needed a fast and reliable framework for evaluating all of the implemented approaches. For this, I have developed a system called The Simulator which enables me to 'plug in' any scheduling approach and with a certain degree of customization, execute and evaluate the approach on a large set of users and drivers. The Simulator is described in Section 4.1.

- **Visualizer** - The visualizer is a web based application which helps the user visualize different scenarios which happened during the execution of a fleet manager. It can be used, as was in my case, to make sense of the results produced by a simulator and understand why the simulator made some decisions. The Visualizer is described in Section 4.2.

Chapter 2

Background

Recent advances in technology such as GPS and fast cellular network have enabled precise vehicle tracking and almost instant communication with the drivers. The last decade has seen the development and successful commercial use of Intelligent Transport Systems (ITS), which are based on a combination of GPS technologies and increasingly efficient hardware and software which allows us to track the position of a large fleet of vehicles and customers and further execute our routing and scheduling algorithms while instructing drivers about their next actions. As a subset of these systems, the Fleet Management Systems (FMS) are specifically designed for managing a corporate vehicle fleet. The FMS are a crucial component in recently emerging ride-hailing services such as Uber or Lyft [14]. The main problem is the pick-up of goods or passengers and their further delivery to locations across a given area.

A key technological feature of FMS is the matching component. Traditionally, the process of matching the passenger and the driver relies on a team of human dispatchers who have a certain idea about how the fleet of vehicles is currently distributed throughout a city, therefore they are able to determine what is the most suitable driver and passenger match. Largely, the skill, experience and size of the team of dispatchers determine the outcome of the most critical operational process of a taxi service. Further, there is a linear relationship between the size of the team of dispatchers and the size of the fleet, which increases expenses as the fleet grows. What we will actually find out is that this team of dispatchers is solving a specific case of the Vehicle Routing Problem (VRP).

2.1 VRP

2.1.1 Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a generalization of the Travelling Salesman Problem (TSP) where a fleet of vehicles needs to be scheduled to deliver goods to a number of customers. Often, the added constraint is that prior to delivering goods, the driver needs to visit a central depot. The problem has been first introduced by Dantzig and Ramser in the context of dispatching trucks [15].

The original VRP sets the basic structure of the problem and it is almost 2 decades later when Wilson and Colvin [16] introduce the element of dynamism, ultimately defining the Dynamic Vehicle Routing Problem (DVRP) which describes a system able to schedule requests arriving dynamically. With additional constraints set, we can form a Dynamic Capacitated Vehicle Routing Problem with Time Windows (DCVRPTW). DCVRPTW considers the vehicle's maximum carrying capacity and imposes a notion of time windows, a range of time in the future during which the pick-up or delivery needs to happen. However, we should remember that we are still tied to the depot pick-up formed in the original VRP. Next widely known problem is the Dynamic Dial-a-Ride Problem with Time Windows (DDARPTW) which consists of designing vehicle routes and schedules for n users who specify pick-up and delivery requests between origins and destination for a certain time slot within a time period (e.g. day or week). Ideally, we want to express that the time window is due in a certain period from the time of the ride being requested and not just arbitrarily anywhere in the future. To define our system correctly and to further understand the possible approaches, we should specify what are the properties when the VRP is being expressed within a ride-hailing taxi service.

2.1.2 Ride-hailing Vehicle Routing Problem

Compared to conventional FMS that usually focus on passengers' requests in terms of their time-window constraints, the concerns in a ride-hailing environment will be different for both the FMS and the customer. In case of an exclusive taxi, where we do not allow any additional pick-ups during the ride, the customer's main interest is to be serviced as quickly as possible because most taxi trips are characterized as a short trip in urban area, and in a shared-taxi environment we are also taking into account the maximal detour time added by picking up other passengers. For the FMS it is similarly reducing the customers' waiting time, utilizing the fleet and decreasing the total distance travelled to save the expenses. This overall idea allows us to define the characteristics of the VRP this project is tackling.

- **Dynamic -** An original VRP might be solved using a two-step approach where we collect requests a fixed time frame W ahead, then plan and finally execute. The DVRP allows customer requests to appear dynamically during the planning or execution phase which might need us to reassign user agent pairings to maintain a set of routes closest to optimality.

- **Immediate -** Extending the dynamic property, the requests proposed in VRP need to be fulfilled at a certain time t (or a time range t_1 and t_2) which usually leaves us with a time window before the request to plan ahead. In our case, we will be receiving immediate requests, introduced by H.N. Psaraftis [17] where users are requested to be serviced as soon as possible so ultimately one of our main goals is to reduce the user waiting time. We are also assuming that at $t = 0$ we are not aware of any orders thus we are not able to plan ahead.

- **Multiple depot -** Every single customer request is treated as a depot which needs to be visited only once. Upon visiting it, the driver will is assigned an address denoting the drop-off location.

- **Capacitated -** We will be considering two instances of the problem, one where the vehicle is exclusive to the customer and therefore has only one drop-off location. Secondly, the increasingly popular ride-sharing model [18] where the vehicle can at one instance carry at most constant c people.

- **Time Window -** We will not impose a time window for the whole trip as this heavily depends on the customer's destination. However, we will have a time window on the pick-up of the passenger. It is difficult to determine an exact hard time limit on how long a passenger is willing to wait for a taxi as it is dependent on conditions or area where the scheduling is happening. We can approximate a time limit from available data, such as the average waiting time of passengers. For example, the reported estimated time of arrival average was 6 minutes in London in 2014 [19]. Based on the previously stated assumptions, we impose a 20 minute time window throughout the thesis, after which the customer will lose interest in requesting a taxi.

2.2 Current Approaches

For a general Dynamic Vehicle Routing Problem with Time Windows (DVRPTW), the most popular scheduling algorithms operate in 2 steps [20]. The first one being described as the insertion step, where according to selected heuristics we pick the driver customer pair and insert it to the global time schedule of planned deliveries, mostly at a random place within the designated time slot. This step is executed each time a new request is placed. The second step is the optimization step which reassigns all orders and the drivers paired with the orders to increase the overall effectiveness of the deliveries, where the effectiveness can be described as the total distance travelled or the total number of orders delivered.

In order to quickly derive a near-optimal solution to a request in the DVRPTW or DDARPTW, heuristics can be used which indicate a good performance without the need to consider all existing possibilities in the current state of the system [21]. We should also note the difference between heuristics and metaheuristics as both of them are used in these types of routing problems [21] [22]. Heuristics are usually problem dependent techniques and they are tied to the problem being solved by taking full advantage of the particularities within the problem. This comes with the disadvantage of trying to solve a certain local optimum which usually does not correspond to the global optimum. On the contrary, metaheuristics are problem invariant techniques and are not tied to the specifics of a problem. They can be customised to the current problem by tuning certain characteristics and parameters, but in most cases their overall solution space is larger than the one considered in regular heuristics. This is of course subject to how we set the parameters and pick the heuristics.

In this project we only consider heuristics, more specifically insertion heuristics because our problem consists of inserting new requests into the current routes at the best possible position known for each driver individually. One of the reasons for not considering metaheuristics is that they are not focused on the specific features of this kind of problem. The lack of a specific approach is usually linked to a shortage of data about the state or the subjects to which metaheuristics are applied. This is not the case as in FMS we assume to have very fine-grained data about out fleet and we can make optimizations on such tiny details as a specific vehicle type fuel consumption or area details of a customer request.

2.2.1 Exclusive Taxi

Exclusive taxi addresses the problem where our vehicles have their capacity restricted to the single customer, or a single group of people treated exactly the same as a single customer. Thus, the driver needs to stop at a single pick-up point and a single drop-off point. We are considering the heuristic based approaches where the global optimum might differ from the local one on a frequent basis. However, in the NVD and IH algorithms described below, we will be using the distance as the main basis for our heuristic, and due to the nature of our specific VRP problem, it is many times the global optimum to which we will arrive simply because the drivers which are the closest to user are usually the ones who can arrive fastest.

Nearest Vehicle Dispatch

Nearest Vehicle Dispatch (NVD) is the most used strategy in current applications, FMS and industrial solutions [9]. It is the most basic and therefore the easiest to implement, while still working well when compared to other approaches due to the distance heuristic described previously in 2.2.1.

The NVD is executed on two state changes, one is when a new request arrives and one is when a driver finishes his/her delivery as both of these actions contribute to the supply and demand change of a FMS. From the perspective of a newly arrived request, we start by looking at the nearest vehicle to the request's origin location, usually by expanding a radius around the origin. We further need to check whether the selected vehicle is able to arrive to the location within a specified time window after which the customer is assumed to lose patience by waiting on a taxi. If it is infeasible to arrive to the customer within the specified time, the algorithm can further consider additional drivers or leave the customer waiting with a hope that another driver will end their delivery nearby and will be able to pick up the customer. From a finished drivers perspective, upon successfully dropping off a customer, we do an NVD 'scan' in case there are passengers nearby who were previously not assigned a driver. The algorithm considers only a small subset of available vehicles and it does not plan further by looking at the passenger's trip destination or duration.

Insertion Heuristic

While NVD searches only for a nearest feasible vehicle to assign a new passenger, the Insertion Heuristic (IH) [23] compares all available vehicles to find the best available match for the request. Every new request is considered on a First-Come First-Served (FCFS) basis, individually and independently from other new requests. Typical IH without a customization for a ride-hailing service has four steps and the goal is to minimize passenger waiting times

1. Request is identified by it's origin and destination
2. Search through all vehicles, including the vehicles which are currently busy with delivering
3. Select a vehicle by considering the distance of the nearest vehicles but also by including the vehicles which might be nearby the pick-up location when they finish their current delivery
4. Select the most suitable vehicle and update the schedule

Since the strategy considers all the available vehicles, not only the idle ones, this broadens the choice of taxis and thus increases the chances of finding a better assignment compared to the first strategy. However, the heuristic is used to insert the route into the pick-up and delivery schedule of a driver and therefore certain modifications need to be made in order to adopt if for the ride-hailing scheduling. Since the heuristic considers a larger set of vehicles, in most cases, it is expected to outperform the first one.

Mixed Integer Programming

The problem can be also formed as an Mixed Integer Program (MIP) inspired by Cordeau [24]. The routing problem can be described as a directed weighted graph of pick-up and delivery locations as nodes. Between each node we will find a possible route with it's associated weight which can be the duration or distance travelled (or both).

We want to assign a driver to the user as soon as the user requests a ride. In such a dynamic environment, MIP is not the ideal candidate for a solution as with increasing number of drivers and requests, the execution time might not keep up with the incoming requests. MIP is more usually seen in a dynamic algorithm which needs to allocate the rides to a certain time window in the future. Nevertheless, it is a popular approach to similar problems and below is stated an example model of a MIP in a ride-hailing dynamic environment.

- Set of pickup locations P with a cardinality of n and each location being denoted as $P = \{a_1, a_2 ... a_n\}$
- Set of delivery locations D with a cardinality of n and each location being denoted as $D = \{a_{n+1}, a_{n+2} ... a_{2n}\}$
- The set L which are all the pickup and delivery locations ($L \in P \cup D$). For simplicity, drivers are assumed to start at a certain pickup location and will end at one of the destinations. This means a that they will not have to travel to the very first customer, but then will act as is expected in all next rides.
- The rides will be formed as a combination of locations from L. More precisely, we will denote a ride as a tuple $R = (i, j)$ where $i, j \in L$ and $i \neq j$
- The cost of a ride between locations i, j is denoted as c_{ij}
- Set of drivers K where each driver $k \in K$ where each driver has a fixed capacity of C_k (in our case just 1 since we have an exclusive taxi)
- Number of people the driver is carrying denoted as w_i^k upon leaving location a_i. For simplicity, assume that the driver can always pick-up only one person.
- Not all locations combinations will be executed, so we need a binary variable x_{ij}^k which is equal to 1 when driver $k \in K$ has allocated a ride starting at i and ending at j.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{k\in K}\sum_{i\in L}\sum_{j\in L} c_{ij}x_{ij}^k && && (2.1)\\
\text{subject to} \quad & \sum_{k\in K}\sum_{j\in L} x_{ij}^k = 1 && i \in P && (2.2)\\
& \sum_{j\in L} x_{ji}^k - \sum_{j\in L} x_{ij}^k = 0 && i \in P, k \in K && (2.3)\\
& 0 \le w_i^k \le C_k && i \in P, k \in K && (2.4)\\
& w_j^k \ge w_i^k x_{ij}^k && i,j \in L, k \in K && (2.5)\\
& x_{ij}^k \in \{0,1\} && i,j \in L, k \in K && (2.6)\\
& c_{ij} \ge 0, C_k = 1 && i,j \in L, k \in K && (2.7)
\end{aligned}
$$

The objective function (2.1) is a minimization of the costs of taken rides. Further, (2.2 and 2.3) ensure that eventually every customer is picked up and that the drivers go directly from the pickup to the delivery point. We did not pose any hard time window upon which the customer might lose patience. Figure (2.4) limits the carrying capacity of each vehicle to between 0 or it's maximum allowed capacity, (2.5) says that the load of a vehicle at the end of a location needs to be higher than at start thus we do not allow 'empty' rides. The last figures (2.5 and 2.6) state that x_{ij}^k is a binary variable, the maximum capacity of a vehicle is 1 and that the costs must be non negative.

Such a defined function would be plugged into a MIP solver (example being GLPK [25]) and the variables alongside with the result of the objective function will be determined. The variables are a direct mapping of the passengers and drivers in the fleet manager, therefore as soon as we find the optimal configuration proposed by the MIP solver, we are able to map the output variables to our fleet manager. We wouldn't be able to run the solver every time a new request is added, as solving a MIP is a computationally expensive task, therefore an approach such as running it every x minutes or after n new passengers would have to be implemented.

2.2.2 Shared Taxi

In the problem with a shared taxi, we are considering a vehicle with a capacity restricted to c customers, where c is the specified maximum number of customers it can carry in one instance. Shared taxis are becoming increasingly popular [26] as the customers are paying a smaller amount of money for the service and the drivers usually travels less miles and decrease their total expenses. Contributions are also significant from the environmental viewpoint thanks to the reduced carbon footprint and lesser amount of cars on the road.

Every passenger needs to be picked up and dropped off at a certain location, therefore we need to search through a combination of possible routes we can execute while still respecting the time window constraint regarding the maximum pick-up time. Also, we will introduce a new constraint d which is the maximum detour for each customer. If the customer requested a ride which would usually take time x depending on the route length and traffic conditions, but it is also assumed that there will be a certain extra time incurred by servicing the other customers in the shared taxi. We can't set d to be static (e.g. 5 minutes), since the maximum detour time will be different for a ride which takes 10 minutes and 1 hour. In the evaluation phase, other research sources use maximum % from the trip's total time and the percentage varies from 0.1 to 0.5 [27]. As with the maximum waiting time, it is therefore difficult to impose a hard limit on the maximum detour, but given the ranges of 0.1 to 0.5 we will use the middle ground of 0.25%.

When compared to the exclusive taxi, the problem's solutions space is several magnitudes larger and therefore we will be less likely to achieve a global optimum. Below, we will be considering a heuristic based approach similar to NVD called Aggregated Greedy Dispatch and a variation of the Mixed Integer Linear Programming from before, while introducing Ant Colony Systems, a metaheuristic based technique. Note that there are more possible approaches that focus on the same family of problems. This project is considering a selection of the most suitable techniques related to a dynamic shared taxi. An interested reader should further refer to [20].

Aggregated Greedy Dispatch

The initial and easiest to implement approach is based on finding a match between a set of customers with a cardinality smaller than capacity c, and the routes for their pick-up and drop-off while keeping in mind the maximum detour d for each customer. This approach is also most widely employed in the applications which perform shared taxi deliveries.

Aggregated Greedy Dispatch (AGD) approach considers all vehicles available, creates combinations of feasible pick-up and drop-off locations for the customer set being considered and greedily picks the first customer vehicle match which satisfies the mentioned constraints. By combination of pick-up and drop-off locations we mean that if we have customer labelled pick-up locations 1, 2 and 3 with drop-off locations as 1', 2' and 3' we can consider arrangements such as 1,2,1',3,2',3' and many more. We can discard a large subset since e.g. a drop-off 1' can't be followed be the pick-up 1 of the same customer. There are 2 possible approaches for AGD, each one with it's specific advantages and disadvantages.

- **Static -** In the static version, we will impose a time window W, starting when the customer issues a request. During W we will try to match the customers with other customers in the same time window, while still preferring customers which are closer to each other (similar to executing NVD on customers). We try to match these customer combinations with available vehicles. If we find a match, the users sharing a ride are immediately notified and the driver is dispatched according to the most efficient permutation of pick-up drop-off locations. If not, the users which have not been served are simply moved to the next time window.

- **Dynamic -** The core idea behind the dynamic approach is that customers can also be served from cars which have began their journey and still have space available. The initial process is the same as in the static problem where we are also starting with a time window W during which we try to create a match. If we match number of passengers below the maximum vehicle capacity, the drivers can make a detour to pick-up a new passenger and update the routes accordingly, while still respecting the detour d imposed by other passengers.

Mixed Integer Linear Programming

The approach has been described in 2.2.1, now we simply need to alter the constraint C_k to the number of passengers we wish to carry. Similarly as before, the algorithm has it's most effective application in a setting where we have specified time windows in the future and we are replanning our overall dispatching schedule based on the arriving orders. In an immediate setting, such an approach is feasible yet it can't be guaranteed that the time window W where we are creating our matches between customers and vehicles is sufficient to consider the whole set of customers and vehicles. As has been discussed, for dynamic problems a more suitable technique is to narrow down the solution space by using heuristics.

Ant Colony Systems

Ant Colony Systems (ACS) used to solve the DVRP were introduced by R. Montemanni et al. [28] and are based on scheduling and optimizations of a set of vehicles and customers split into time slices of the day of equal duration. Any requests which arrive during the time slice will be processed at the end of the slice so the optimization is run statically and independently of others with main advantage being similar processing time during these slices. The optimization step is based on the natural way of how ant colonies work. Ants use pheromone trails to communicate the shortest path to food and they create a path of this substance throughout the environment. When an ant wishes to find it's way to the food they will become attracted by the pheromone substance which they will ultimately and with a high probability follow, while also reinforcing it by their own pheromone. Translating the approach to DVRP, we will be essentially constructing a set of preferred paths which the agent chooses according to a probability distribution. The probability distribution is further specified by our custom picked heuristic, such as distance and the previous pheromone trail left on the path.

As in our problem the solution space stays static throughout the time slices (in the next time slice, we are still performing the routing and scheduling in the same city) we can pass the information about the 'good solutions' to the next time slice and leverage this information in constructing the new set of paths. Similarly as in AGD, we can immediately see that the smaller the time slices, the less optimal will the solution be since the algorithm has a smaller set of requests and vehicles to consider. Further, the optimization step is similar to MILP where we potentially need to execute many iterations within a positive feedback loop to determine a set of suitable paths with the pheromone trails. Terminating this process early, e.g. by specifying a hard time limit on the optimization's execution time might very likely leave us with a set of sub-optimal trails or trails with a uniform spread of pheromones, thus resulting in the inability to deterministically choose any of them.

2.3 On-demand Transportation Networks

In the previous sections we have defined the VRP we are tackling within the FMS of a ride-hailing service and the possible approaches for solving it. Now we want to put the FMS into the context of the whole application in order to explore existing libraries and real world applications using the mentioned approaches. Also, we will find out that there are more techniques which can be used to improve the customer's waiting time and the total distance travelled by our fleet which are contained outside of the scope of what scheduling and dispatching cover.

2.3.1 Environment

To narrow down the set of existing applications and libraries, we need to specify the On-demand Transportations Network (ODTN) environment where the system will operate. This was partially defined in terms of the VRP previously, but not in the context of the whole application.

- **On-demand -** The system needs to respond to customer requests as soon as they arrive. Further, the aim is to minimize the waiting time for a user so it is essential to pick a match between the user and agent as soon as possible.
- **Transportation -** We are concerned with transportation of passengers.
- **Network -** We have a network of agents (drivers) which we also interchangeably can call a fleet. While every agent is an individual, we will try to leverage the information about the position of all of the agents to make sensible dispatching decisions.

2.3.2 Industrial solutions

As has been previously mentioned, even small optimizations within the FMS are capable of saving a large amount of money annually by decreasing the total amount of miles the vehicles have travelled or increasing the total amount of customers served. Similarly to this, we have companies who are providing the FMS software in form of a Software as a Service (SaaS) or an API because it might be more comfortable for a company to pay per each request while not bearing the costs of developing their FMS in-house. Also another factor might be the lack of such solutions on the open-source scene which we are discussing further in 2.3.3.

Stand-alone FMS

As an FMS is usually used as one component of a larger system (usually being a ride-hailing application), it is very common to offer the fleet manager in form of a stand-alone library or an API. This is then further implemented into the rest of the application, perhaps by connecting it to the front-end component used for displaying the fleet and allowing customers to view where the vehicle currently is or with the geolocation services in the phones of the drivers to be able to effectively track their location throughout a city.

Compared to creating a full on-demand transportation service which is also assumed to employ own fleet, a library or an APIs is much less demanding to develop and maintain. Therefore the market for paid SaaS applications which handle fleet management is rich and potential solutions range from simple VRP solutions up to dynamic routing problems spanned across large geographical areas. Usually, the service assumes that the customer has the capability of his/her own fleet and they need a software to effectively match and schedule the incoming requests. Example are services as InDemand [29] or Juggernaut [30]. Further, there are also solution as UberRUSH [31] which offers the existing network of Uber drivers. A problem with these services is that the offered solutions are not usually tailored to the customer, they are offered without any context. With additional knowledge, which could be obtained from a comprehensive overview of on-demand scheduling approaches, the customers would be able to make more data driven choices.

On-demand Transportation Services

This section considers a complete application with an interface provided for both users and drivers with examples being companies as Uber or Lyft. In the recent years, thanks to a successful large scale execution across many cities worldwide, the research in the area of DVRPs has gained significant popularity. Such a service is composed of many components and layers, yet the FMS plays a significant role, maybe the one most directly tied to the revenue stream of the application.

As has been already mentioned, when it comes to the fleet management systems of these services, very little is known as this can be considered a valuable intellectual property. However, Lyft has claimed to be using the NVD [8] algorithm in an exclusive taxi and a variation of the dynamic AGD [13] algorithm for their shared-taxi rides. Further, there is also evidence of Uber using the NVD in their UberX (exclusive taxi) service [32].

2.3.3 Open-source solutions

One is able to find VRPs in many aspects of our lives. We are very dependent upon distribution and moving of things and people, with very basic examples being a daily commute to work or supermarket supply scheduling. This may offer an explanation for a rich amount of papers and algorithms tackling this topic on many different layers. However, there exist very few implementations in the open-source scene, where the majority are libraries focused on the general VRP. Even fewer of the solutions can deal with an instance of a VRP with many constraints and the scale of vehicles and requests which are present in the real world. Going further, there is a minimum of applications capable of a full fleet management with the added benefit of an interface for the customer and driver.

Libraries, Toolkits and APIs

This sections provides an overview of existing open-source implementations tackling different variations of the VRP. Many of these libraries are built for tackling the general VRP or that they solve a larget set of scheduling problems using metaheuristics, however one can extend them to fit a specific VRP with added constraints, such as our ride-hailing version of the VRP.

Library	Overview and Approach
jsprit [33]	Currently the most popular library for solving the travelling salesman problem (TSP) and the vehicle routing problem (VRP). The general problem can be customised with a large set of available constraints.
	Approach is based on Simulated Annealing - metaheuristic which can be used to solve a large family of VRPs, usually with less dynamic elements.
Open-VRP [34]	Framework to model and solve VRP based problems. The set of available constraints which can be set on the fleet is however smaller when compared to jsprit, currently being only the capacity or time windows.
	Implementation assumes that the developer will supply an algorithm used for the specific problem he/she have modelled and wishes to solve. The is also a default option for a metaheuristic based tabu search.
Hipster4j [35]	Heuristic search based library in Java with particular focus on customisation of the search algorithm. Therefore it is possible to form and solve almost any VRP.
	Available is a wide variety of common search algorithms such as A*, Bellman-Ford or Hill-Climbing. The basic use case of the library is to extend or adapt these basic search approaches to a custom problem.
OptaPlanner [36]	Constraint solver and a planning engine for a wide variety of scheduling job such as VRPs, task scheduling or timetabling. Again, the specific VRP needs to be formed in terms of OptaPlanner's scheduler.
	The approach used is a combination of several heuristic and metaheuristic approaches, most namely simulated annealing and tabu search.

Table 2.1: Open-source libraries and toolkits

On-demand Transportation Services

As has been defined 2.3.2, we are considering applications with a full interface which would ultimately work in a 'plug-in-your-fleet-and-play' fashion. Such an initiative requires a great amount of work, maintenance and optimization in areas such as the system's fleet manager to provide efficient solutions for the users. As a result of this, there is only a handful of such projects the most widely known project is LibreTaxi [37] which offers an interface for both the customer and agent while incorporating a basic fleet manager. Besides this, most open-source projects provide just the interface, such as Opencabs [38], while leaving the routing logic to the developers. Since LibreTaxi is an open-source project, we can observe that it uses an instance of NVD for the fleet management where it creates a radius and notifies all nearby drivers [39].

2.4 Other

In addition to the core dispatching algorithm which is concerned with finding a suitable solution for the given VRP there exists a variety of areas where we can think of optimization which increases the overall effectiveness of the fleet. They are usually not concerned whether we are solving the exclusive or shared taxi scheduling. This section outlays number of possible areas which significantly affect the effectiveness of the system.

2.4.1 Distance measurement

There are several ways of measuring distances between vehicles and customers that can be applied for finding the nearest vehicle:

- Straight line (SL) - The SL approach uses the haversine formula which computes the distance between two points on a sphere given their longitudes and latitudes. This is simply the equivalent of 'as the crow flies' distance, it offers the lowest precision. The main advantage is that it can be quickly calculated since it does not require running the shortest path search.
- Travel distance (TD) - The shortest distance path from the vehicle to the customer. Usually computed using the A* algorithm which acts according to a chosen heuristic (such as straight line distance) and expands the next available node which has the lowest cost. If we are currently at node $n0$ and considering a node $n1$, the cost is defined as a combination of the cost associated with the edge from $n0$ to $n1$ and a value returned by computing the chosen heuristic from $n1$ to the end node. In such fashion, we can iteratively find a path from start node to the end node.
- Travel time (TT) - From a set of possible routes we pick the one with the shortest travel time. Travel time can be fixed at a certain constant speed of the vehicle or can be subject to traffic and weather conditions.

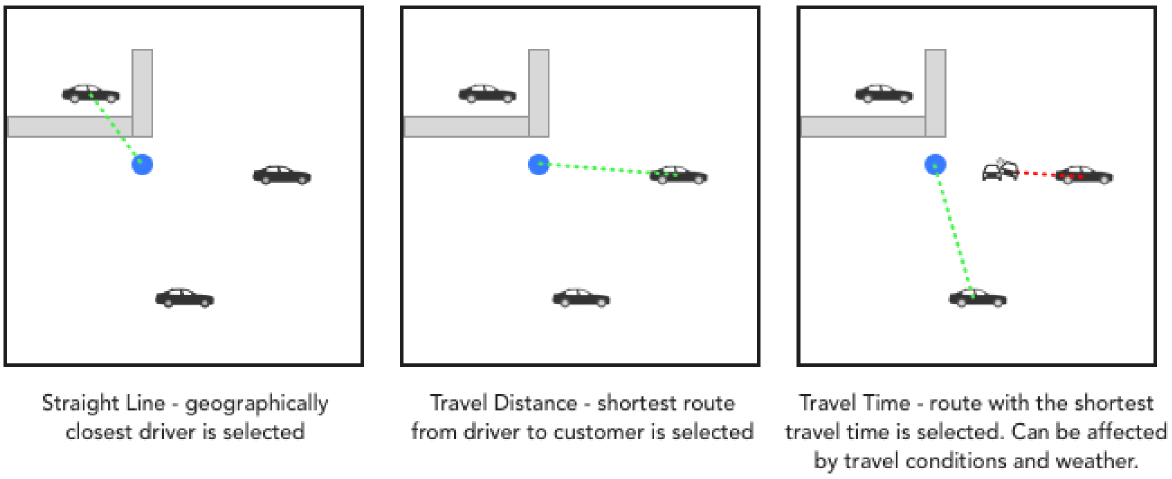

Straight Line - geographically closest driver is selected
Travel Distance - shortest route from driver to customer is selected
Travel Time - route with the shortest travel time is selected. Can be affected by travel conditions and weather.

Figure 2.1: Distance measurement options

2.4.2 Free Drivers

In areas with a smaller user to driver ratio it happens that agents do not have a request assigned and are therefore waiting for the next request. When such a situation happens, the agent might be interested in optimizing his/her chances of finding the next request as soon as possible. Below are described three ways of possible agent behaviours, based on observations from Uber [40].

- Stationary - Agents do not move unless they have a passenger. Driver therefore does not incur any more fuel expenses, however he/she might be exposed to a longer waiting time for the net request.
- Random - The agent might want to wander randomly around the area in hope to find a nearby match.
- Gravitational - Agent will navigate back towards an area with the highest demand density since the drivers usually know where the popular pick-up spots are.

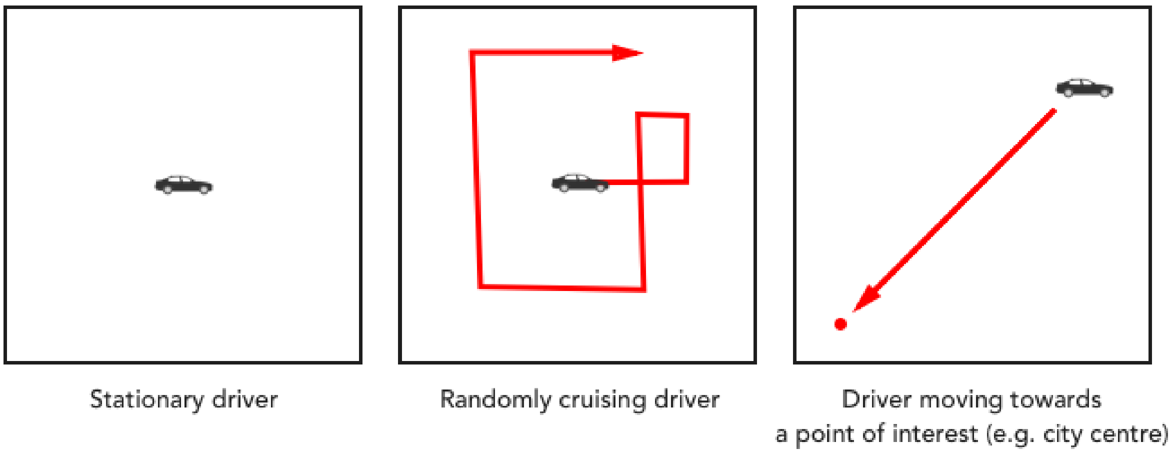


Figure 2.2: Free driver's cruising options

2.4.3 Area clustering

The fleet manager does not operate on a global scale as this would cause unnecessary memory and processing demands. Instead, the scheduling area can be split by selected demographics and a unique set of demographic attributes would make us consider creating a separate transportation network with a disjoint fleet manager. This project does not directly address the topic of area clustering, but it is a proposed extension which can build on top of the existing implementations.

Chapter 3

Implementation

In the Chapter 3 of the project, we will be looking into the implementation details of the proposed scheduling approaches. In case of a fleet manager, we usually think of scheduler as the system used for pairing the passengers with the most suitable drivers. This is from a one way of looking at the scheduling problem, and we are describing proposed approaches in sections 3.1 and 3.2. These sections address the most immediate problem of receiving passenger requests at certain locations and matching them with the available drivers. Usually, these matches are based on certain heuristics and are optimized even after they are initially allocated.

For pairing passengers and drivers, we are addressing two different problems. The first one is Section 3.1, where the domain is an exclusive taxi, meaning that only one passenger can be in the taxi at a time. Secondly, it is the increasingly popular ride sharing [18] option where a driver can have multiple passengers in the cab with multiple drop off locations. This problem is addressed in Section 3.2.

Lastly, in Section 3.3, we will also explore a nontraditional area of the fleet manager, which is scheduling drivers (without pairing them with users) to drive to places where they have an increased chance of finding a passenger. Firstly, an approach based on looking at the most immediate history of the state of the fleet manager is used to predict the future demand is used. After, that, the next step is incorporating machine learning to infer more complex relationships and give more accurate predictions.

3.1 The Insertion Heuristic

The Insertion Heuristic (IH) approach proposed in 2.2.1 can be modified to suit the setting of a ride-hailing application. The algorithm will need to take into account both free agents and agents currently transporting a customer. In a general DVRP, the IH would insert the planned trip into a schedule planned for a certain period of time (e.g. a day). This contrasts with the NVD approach where we don't have a notion of a schedule at all, only the ongoing customer journeys.

One option would be to implement the IH with the notion of a global schedule, such as the mentioned DVRP, where we would insert every single new ride. Global schedules are used in order to optimize the driver and passenger matches within a time-frame which spans several hours or days ahead. However, doing so in this case does not work well, simply because we would be always appending to the global schedule as we want to serve the customers as soon as possible. Another major challenge is maintaining a global queue which would be concurrently updated by all the incoming requests and scheduled rides.

As a result, IH is in the middle ground between these two approaches, since it will be using a schedule, or more accurately a passenger queue, per every agent. It is an important part of IH as it is a source of optimizations within an already heavily constrained space, which we would not be able to perform using a global queue. We will also consider under what conditions having a queue might not be a good idea, mainly because of the challenges tied to uncertainty in travelling times. The detailed workings are explained in detail in the next subsection.

3.1.1 Algorithm

For a customer who has issued a pick-up request the algorithm starts with the NVD approach by first considering the free agents set and trying to pick the agent according to a heuristic, which is usually the closest range, or shortest travel time as we will see in later experiments. In it's next step, the algorithm will now consider all busy agents where it needs to approximate the ending time of the busy agent's ongoing delivery and calculate if there is a time gain by waiting for the busy agent to finish the order and then assign the passenger to him/her. If a busy agent is determined to be the more suitable match than a free agent, the passenger is appended into the busy agent's passenger queue. Of course, all of the heuristics above are subject to the passenger's maximum waiting time which we can't exceed because then they will lose patience. This step is also called the **insertion step**.

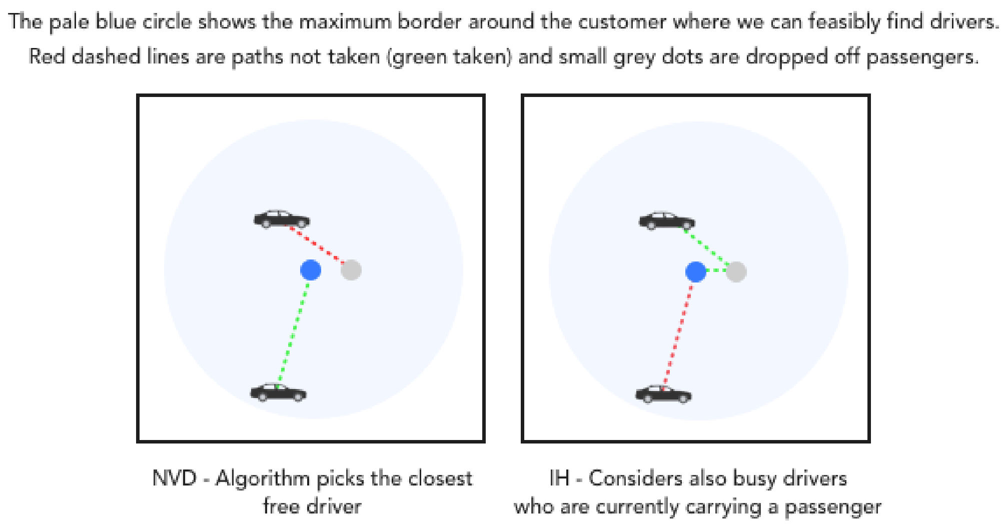
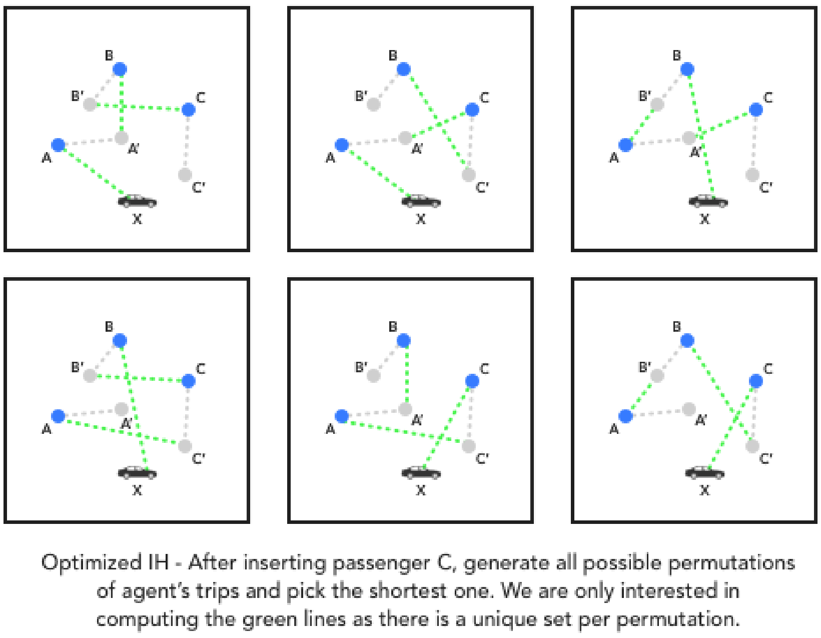

The pale blue circle shows the maximum border around the customer where we can feasibly find drivers.
Red dashed lines are paths not taken (green taken) and small grey dots are dropped off passengers.
NVD - Algorithm picks the closest free driver
IH - Considers also busy drivers who are currently carrying a passenger

Figure 3.1: NVD and IH comparison where the IH finds a more suitable match

As has been described in 2.2, algorithms for solving a certain version of the DVRP usually involve also an **optimization step**, which we have determined is not feasible in our case because we don't have a global schedule to optimize on and that they are usually expensive in terms of the computation power. But now, we actually do have a schedule, the passenger queue. Size of the queue never grows to infeasibly large scale because of the constraints we have posed on the customer's maximum waiting time which allows us to find solutions efficiently.

The optimization step is executed every time a customer requests a ride which is matched to a busy agent who has at least 1 or more passengers in his/her future passenger queue. Future queue simply means that we don't consider currently ongoing rides (as they are still in the queue) but only ones which have $t' > t$ where t' is the pick-up time of the passenger and t is the current time. We append the new passenger to the queue and we start by essentially solving the Travelling Salesman Problem (TSP) on a small scale. The goal is to find out what is the shortest path which serves all passengers in the passenger queue, thus we need to determine the most suitable order of pick-ups and deliveries of the passengers. We will do this by generating all possible permutations of the queue and determining which one is the most effective.

As an example, let's take an agent with a starting position X, a passenger queue (where all passenger trips start in the future) of size 2, with passenger pick-up coordinates A and B and delivery coordinates A' and B'. A new passenger with pick-up C and delivery C' is inserted. The possible permutations are (X AA' BB' CC'), (X AA' CC' BB'), (X BB' AA' CC'), (X BB' CC' AA'), (X CC' AA' BB'), (X CC' BB' AA') also shown in 3.2. The X is added in front of every permutation because the agent does not necessarily start at passenger pick-up points so a combination of driver's starting point and the first customer pick-up needs to be considered too. Note that this is a problem addressing an exclusive taxi described in 2.2.1, every pick-up needs to be followed by the drop off of the same customer (otherwise we would have multiple passengers in the cab) which means that we will always need to generate $(n+1)!$ permutations for a queue of length n and with a one new passenger being added. Since the pick-up followed by a drop off trips (AA', BB', CC') will be always of the same length we can save computation time by calculating the travelling distance only for the connecting trips of a permutation (A' to B, X to C and others) as there is a unique set of these in each permutation.



Optimized IH - After inserting passenger C, generate all possible permutations of agent's trips and pick the shortest one. We are only interested in computing the green lines as there is a unique set per permutation.

Figure 3.2: Overview of the permutations generated in the example

The algorithm terminates when all of the possible permutations have been tested and the resulting permutation is the one which involves the least travelling time between the connecting trips. Also note that the IH is expected to outperform NVD just by incorporating the insertion step and the optimization step which considers permutations is optional.

3.1.2 Driver Selection

One of the challenges in the insertion step of an IH is the matching of a passenger with the closest driver, which is very often based on the Haversine formula and the actual route length is only found out when the agent is going to pick-up the passenger. This is because if the set of all the agents to consider is large, computing a pathfinding algorithms such as A* for every agent can be computationally expensive. There are optimizations in form of Geohashing to bucket latitude and longitude coordinates so we would need to consider only a subset of drivers, but that is out of the scope of this project.

However, many times there are instances when the Haversine distance can be misleading, e.g. if the driver is behind a hill or there are other obstacles on the route. A possible solution which is also used in this project is to collect N drivers who are closest to a passenger. Then, for every driver in the list we will compute the actual route using a pathfinding algorithm and pick the driver who has the shortest route, or who has the shortest estimated travel time. The size N of the list should be set to a reasonable amount in order to decrease the amount of drivers we need to consider in the end.

3.1.3 Passenger Queue per Driver

In theory, we should be able to insert as many passengers into the queue as possible, as longs as the maximum waiting times of passengers are respected. This allows for extreme situations such as when we would have series of very short trips e.g. each taking approximately 2 minutes and each of these trips would be assigned to one same agent. If we assume that the maximum passenger waiting time is 20 minutes, the agent has ultimately 10 passengers in the queue (we are neglecting here the time to travel from a drop-off of one passenger to the pick-up of another one). In practice, as the times are just estimations, we can never know how long will the trip take and it might be suboptimal to greedily schedule all of the passengers to the single driver. The estimations are getting less precise the more further ahead we are looking and ultimately very minor events such as an unexpected red light might lead to losing the passengers at the end of the queue.

While this project operates in a deterministic environment where we don't consider delaying events such as car failures or traffic conditions, we can observe what are the implications of restricting the driver's passengers queue to grow uncontrollably. A possible way of doing this is to limit the queue to schedule at maximum N passengers. This means that even if a busy agent would be chosen as a suitable match for the passenger, but the agent has already N passengers queued for delivery, they will be passed on to be considered by other agents. Another possible way is to limit the queue based on total distance the driver has to travel in order to pick up and drop off all the passengers contained in his/her passenger queue as the distance travelled is directly linked to the error we might have in our estimations.

3.2 The Aggregated Greedy Dispatch

The Aggregated Greedy Dispatch (AGD) approach was proposed in 2.2.2 and it is based on the algorithms used by Lyft, described in [13]. At it's hearth, it is a greedy approach used to schedule trips where multiple passengers can travel at once in the same taxi. The literature only provides a theoretical overview of the AGD, therefore the exact implementation details are left to interpret for the developer. We start off by implementing a static version of the AGD, described in 3.2.1, which explains how we aggregate the passengers who will be dispatched together. We use the word static because we don't allow a journey which has begun to be altered (e.g. to make a detour and pick up a passenger, even though he or she might be along the way). However, since the static AGD only provides a way for aggregating the passengers together, dispatching them is a different problem described in subsection 3.2.2.

Combining the static AGD and a solution for dispatching the aggregated passengers results in an approach called basic AGD which we will implement and which we will be extending. Therefore, starting from subsection 3.2.3, each subsection will correspond to an extensions of the basic AGD. We discuss the implementation decisions made when developing the proposed extensions and we will evaluate the respective computational efficiency. The computational efficiency is becoming increasingly important, as we could potentially end up comparing all passengers with all drivers, which would many times render as impossible, because we need to determine a set of passenger driver matches within a few seconds.

3.2.1 The Static AGD

There are two ways of approaching the design of the Static AGD. When users issue a request for a ride, we can set a fixed period of time during which they can be matched with other passengers if their time windows are overlapping. As soon as we greedily find a match which is good enough to satisfy all passenger constraints, we dispatch the closest driver to full-fill this match. Other approach, one taken in this project, is to have a single matching pool where all users requests arrive and which is processed every *t* seconds or minutes. For example, if we set to process the matching pool every 5 minutes, then a request which arrived at time 14:02 will be considered together with a request which arrived at 14:04. However, the user who's request arrived at 14:06 needs to wait 4 more minutes to be potentially matched. Users who are not matched are transferred to the next pool, until they are matched or total of 20 minutes passes since their request and they loose interest.

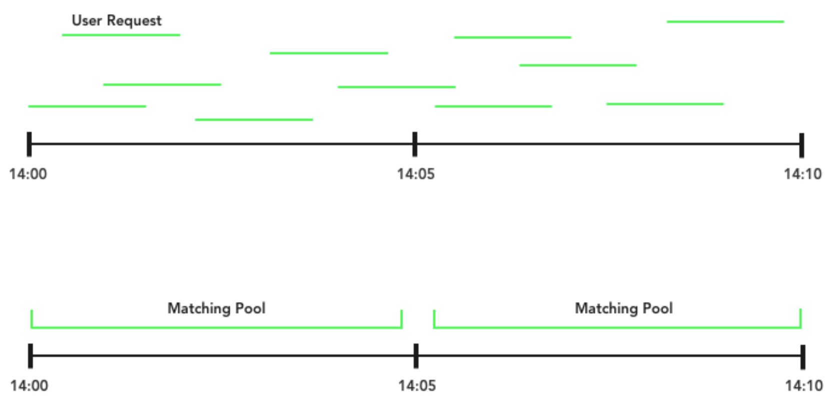


Figure 3.3: Fixed period of time (on top) compared to Matching Pools (on bottom)

There is not a large difference in picking between these two approaches, however the pools give us an advantage of having a single array which we need to sort in the 3.2.4 extension. Otherwise, we would need to perform the sorting for every single passenger separately.

After the pool has been established, at the end of every time window we need to determine a set of shared trips among all the users in the pool and dispatch free drivers to fulfill these trips. For this, we will use a standard greedy algorithm.

3.2.2 A Greedy Matching Algorithm

Firstly, an important parameter, determining the maximum amount of passengers simultaneously allowed in the taxi, needs to be set (usually it is 4). The more passengers at once we allow, the more complex it is to find a suitable combination of users out of all the possible combinations. The simplest and computationally most expensive way is to loop through all possible permutations of 4 passengers and for each of these combinations generate the possible orderings for every passenger's pick-up and drop-off locations. During this process, many combinations can be excluded by checking the distance between any two passengers. If the estimated time for a driver to travel this distance is larger than the maximum waiting time, it is not possible to pick the combination so it can be excluded. However, if all constraints are met, the best performing combination will be chosen and the process repeats. If we consider all 4 passenger combinations, but none would be found to be suitable, we move to 3 passenger trips and then to 2 passenger trips. The assumption is that trips which accumulate more people are more effective because they exploit the common path the passengers share.

The challenge which I faced with the above mentioned approach is the complexity. For a pool which accumulates hundred or more passengers during the time window, we will end up doing billions of comparisons. One way of mitigating the complexity is to decrease the time window size, which will result in less passengers accumulated, but potentially schedules rides further from the optimal solution. The approach could be though to be ”decremental”. It looks at the 4 passenger rides, once it considers all of them it moves to 3 passengers and finally 2. Inversely, we can look at an incremental approach. We start by picking two passengers who have the shortest combination of pick-up and drop-off locations. If we denote the pick-up locations as numbers and the respective drop-off as a number with an apostrophe, we can see that for 2 passengers, there are only 4 possible orderings to consider. That is [1 2 1’ 2’], [1 2 2’ 1’], [2 1 1’ 2’], [2 1 2’ 1’]. From there, we will consider a third passenger to be added, however with the restriction that the order of the existing match can’t change. Therefore if the ordering [2 1 2’ 1’] was chosen, and we are inserting passenger number 3, we can’t have [1 2 2’ 3 1’ 3’] because that would alter the previously chosen match. In a similar incremental fashion, we can get to 4 passengers or to any N. If we are searching for e.g. the 4th passenger and we fail to find a match which would satisfy all the detour and passenger patience constraints, we stop and use the best 3 passenger match.

Since we don’t need to generate all permutations, the computational complexity of this approach is significantly reduced, however at a cost of a potential decrease in effectiveness.

3.2.3 The Dynamic Matching Extension

Currently, every new passenger request is instantly added to the matching pool. However, what would make more sense is to first consider the busy drivers who have free space and are able to divert from their existing scheduled route without violating the timing constraints posed by other passengers on the existing journey. In this manner, the Dynamic Matching extends the Static approach. This subsection describes the implementation details and limitations of the approach.

The algorithm is executed every time a new request arrives. We start by considering all busy drivers who are within a 20 minute driving range. A busy driver is picked and the new passenger is inserted into the schedule at every possible starting/ending slot and a set of combinations is generated (again, without reordering the previous schedule). Firstly, on each of these combinations the detour which we have introduced by picking up a new passenger needs to be calculated. If the detour violates any of the existing passenger’s maximum detour time, the combination can’t be considered further. If found, the best driver’s best combination is used to set a new schedule for the driver and the passenger is arranged to be picked up. However, many times we may find the drivers to have an already tight schedule and not a single match will be found. Then, the request is added into the matching pool where it is considered by the free drivers at the end of the processing window. An example of this behaviour is shown in Figure 3.4 where we have a driver starting the same journey. In case of the static matching, the journey is fixed and no changes are made. The dynamic approach however allows us to pick up additional passengers and thus be more effective.

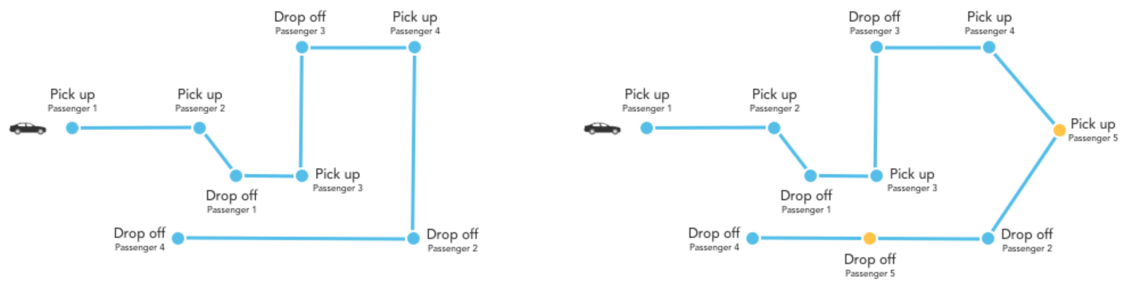


Figure 3.4: The route determined by static matching (left) vs. a dynamic route (right)

3.2.4 Distance Sorting

Throughout the previous algorithms, we are extensively comparing passengers and drivers based on their location. Also, the very first thing we check is whether the two locations which are being compared are within the range of maximum waiting time. Such behaviour can be leveraged by having the matching pool or any other list which involves the locations of drivers or passengers sorted by the latitude or longitude (depending on the geographical characteristic of the area where we are performing the scheduling). If the passengers or drivers are sorted in an increasing order, we can break from the loop as soon as we find the first item which is not within a suitable distance as it is guaranteed that the next driver or passenger will be even further.

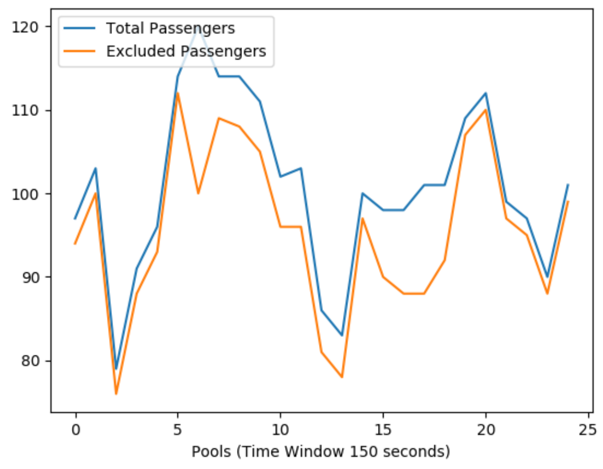


Figure 3.5: Overview of the number of excluded passengers after applying distance sorting

The Figure 3.5 demonstrates an example where we picked a random passenger in the queue every time a pool was processed and plotted how many other passenger did we not have to consider thanks to the distance sorting.

A more elaborate and optimal solution can be designed. We would need to convert the incoming requests latitude and longitude to a 2D plane x,y coordinates. This could be done using the Mercator projection, however at a cost of losing precision (mainly at the poles where we don't expect to perform any scheduling). Such x,y coordinates can be stored in Quadtrees which are able to compare two dimensional data points in O(log n) time. However, we don't provide an implementation or evaluation of this solution as it would cause a major deviation from the approach taken in the source [13] which we are following in this section. Overall, these approaches do not bring us closer to the optimal solution, but they should decrease the computational complexity.

3.2.5 Route Exchange

The Route Exchange can be though as an extension applicable to both Static AGD and to it's dynamic matching extensions. In this project we are implementing the Route Exchange optimization on top of the dynamic matching.

From a large portion, the time window size after which we process the matching pool largely determines the optimality of the scheduled rides. Simply, the larger it is the more passengers we consider and the more options we have. However, the size must be sensibly set because one of the most important metrics which this project aims to optimize is the passenger waiting time. Inevitably, we will not always be able to accumulate the optimal set of passengers and they will be scheduled in different pool batches. However, even after a certain time after a passenger has been scheduled, they might be still waiting for the driver since they are part of the fixed journey. This leaves us space to exchange a large part of the driver's scheduled journey, the part which has not been fulfilled yet, between the drivers. This is because the locations of drivers constantly change and every time we finish processing a matching pool, we have a new batch of journeys added. Therefore, it is easily possible that drivers scheduled by two different matching pool batches exchange their routes because one of the drivers was not available in the earlier batch.

Firstly, a time window tw needs to be set after which the route exchange algorithm will be executed. We need a time window with a large enough size in order to accumulate enough drivers with scheduled journeys. We start by iterating through the set of the busy drivers and building a list of unfulfilled journeys. Every driver contributes with the largest part of his or her journey which has not been fulfilled yet. We are not aiming to exchange a random subset of the journey, it must be a part which spans till the end of the scheduled journey. If we refer back to the notation used in 3.2.2 then in an example schedule [1 2 2' 1' 3 4 4' 3'] where the driver has already picked up the passenger number 1, we must wait until the 1 has been dropped off (1'), therefore the largest part which can be 'sliced' is [3 4 4' 3'] as at that time the driver does not have any other passenger in the car. A different example would be [1 3 3' 2 2' 1'], and as soon as the driver picks up passenger 1, he/she can't participate in the route exchange as they are obliged to drop the passenger off at the end, which would not be possible if we suddenly scheduled a different journey for the driver.

After the list of unfulfilled journeys has been built, we will start comparing them to each other. We pick a journey and find the driver associated with it. Then, we need to look at the location where the driver will be immediately before the unfulfilled journey. We will call this the starting point of the driver. In the Figure 3.6, the green dot symbolizes the starting point and the green line is the distance between the starting point and the beginning of the journey which can be exchanged. We wish to minimize the length of the green line, because the other parts of the journey remain static, disregarding the driver they are assigned to.

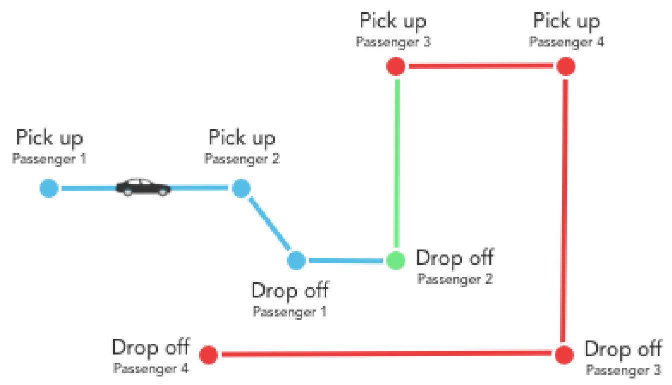


Figure 3.6: Example of a journey. The red path is the unfulfilled part and can exchanged for a different driver's journey.

Note that the exchange is performed for two drivers, therefore the green line changes on both sides and we need to calculate the gain or time loss on both sides. Of course, we only accept an exchange of two unfulfilled journeys if the total distance of these two exchanges is lower than the original one. However, this would also include the case when, for example, the first driver will get assigned a new part of the journey which is 90 seconds closer, but the second driver will get a journey which is 20 second further. We have still gained 70 seconds, however the maximum detour times for the second driver need to be checked, as it might happen that the extra introduced time will cause some of the passengers to lose patience.

It should be also noted that this optimization does not necessarily need to be executed at the end of set time windows. We could also execute the route exchange step every time a driver is in the middle of a journey but at that time there are no passengers present in the taxi. Referring back to Figure 3.6, this would be the part of the graph in green colour. We would simply find other drivers in a similar situations, looking for a route exchange and perform any viable exchanges.

Once the best match is found, we update each driver's journey plan and remove the respective journey from the unfulfilled journey list.

3.3 Gravitational Points for Agents

We have already covered the main scheduling approaches which focus on matching a passenger with an agent. This solves the most immediate problem of a ride hailing application, yet there is also a large space left unexplored when the agents are in between rides, waiting for a passenger to get assigned to them. The drivers might be able to drive towards locations experiencing heavy demand which ultimately increases the effectiveness of the fleet manager. Possible approaches were discussed in 2.4.2 where it can be seen that the most effective strategies for a driver are to either remain stationary, or more interestingly, to drive towards a known hot-spot where they are more likely to find a passenger [40].

Perhaps a more suitable description is that the drivers ”gravitate” towards the local hot-spots of the city. They are driving closer and closer to the points of interest (e.g. train stations, historical city centres, airports) with the assumption that in these areas, they are more likely to get matched with a passenger. These points of interest are usually general knowledge amongst drivers or can be easily determined within a city. However if the majority of drivers would follow this approach, there are several concerns which arise.

1. **Supply & Demand** - If most of the drivers would be pulled towards the same set of locations, it can easily lead to oversaturation of the target areas by drivers. The drivers would be more successful if they chose a different location, yet from a driver’s point of view this is a challenging task as they don’t have an overview about the distribution of other drivers.

2. **Starvation** - If most of the drivers drift towards these more attractive locations, the more distant areas will remain uncovered or the passengers will have to wait longer for the nearest driver. This works closely with the previous point, the situation can be that if too many drivers move from a location, it can be that some of them would find themselves returning back to the point from where they left a while ago simply because so many drivers left the location that they have a higher chance of getting a passenger there.

3. **Reliability** - The set of locations which represent the main points of interest are static and they don’t necessarily reflect the true state of the arriving requests. The true state most likely depends on the day of the week and the given hour, but the drivers do not posses the same information as the fleet management software does, therefore they are likely to drive towards sub-optimal locations.

The main problem for drivers is the lack of information about the global state of the requests arriving in the city and what is the location of other drivers, therefore the decision made by the driver to which point they should drive now are made autonomously without any input. A large portion of valuable information a fleet manager has, such as which areas of the city are currently experiencing the heaviest demand or how are other agents distributed throughout the city, remains unused.

In order to leverage the knowledge a fleet manager has about the global state of the passengers and drivers, we can take a more dynamic and data-backed approach when it comes to the points towards which drivers gravitate. The main idea is to have these gravitational points spawning (and disappearing) dynamically across the city at various locations. The locations are determined based on the area's demand and would be only created only if there is a lack of drivers to meet the demand. Further, these gravitational points would be only shown to a certain subset of free drivers, subset large enough to satisfy the demand in the area. The would cover the point **1** of our concerns, as we would only dispatch a certain amount of the drivers to the gravitational point. Point **3** would be satisfied simply because the fleet manager has an overview of the whole fleet and all the arriving requests and only suggests a gravitational point if there are circumstances to do so. We will be discussing the problem raised by point **2** in the implementations themselves.

I have suggested two approaches for tackling this problem, each with it's own advantages and disadvantages. Approach one analyses the recent trends and assumes that these trends will continue in the nearest future. Approach two builds on top of the first one but also includes historical data from the given area and uses machine learning in order to predict the demand.

3.3.1 Approach 1

The first approach responds to the most recent state of the fleet manager. We fix a configurable time window (e.g. 1 hour) during which different areas of a city collect passenger metrics, such as total number of passengers lost in the area or the average waiting time. This is also called the **Collection** phase of the algorithm. At the end of the collection phase, the **Proposal** phase begins where every area proposes a value which is a result of a cost function computed based on the collected metrics. Lastly, we have the **Dispatch** phase where based on the proposed values, a subset of areas is selected and the gravitational points are spawned. We then select another subset of free drivers and dispatch them to these points.

Initialization

In order to be able to suggest exact locations for the gravitational points, we need to segment the city into areas. These areas can't be too small, because the differences between them would be minimal and we would ultimately start spawning a large amount of gravitational points. They also can't be large because it can easily happen that a single segment would be covering multiple more optimal smaller segments. To tackle this problem, we will be splitting a city into tiles according to the Military Grid Reference System (MGRS). MGRS offers different granularity of the tiles, and the developer should fit the size of the tiles accordingly to the size of the city where the fleet manager operates. Sensible values can range from 100x100 metres to 1x1 kilometer.

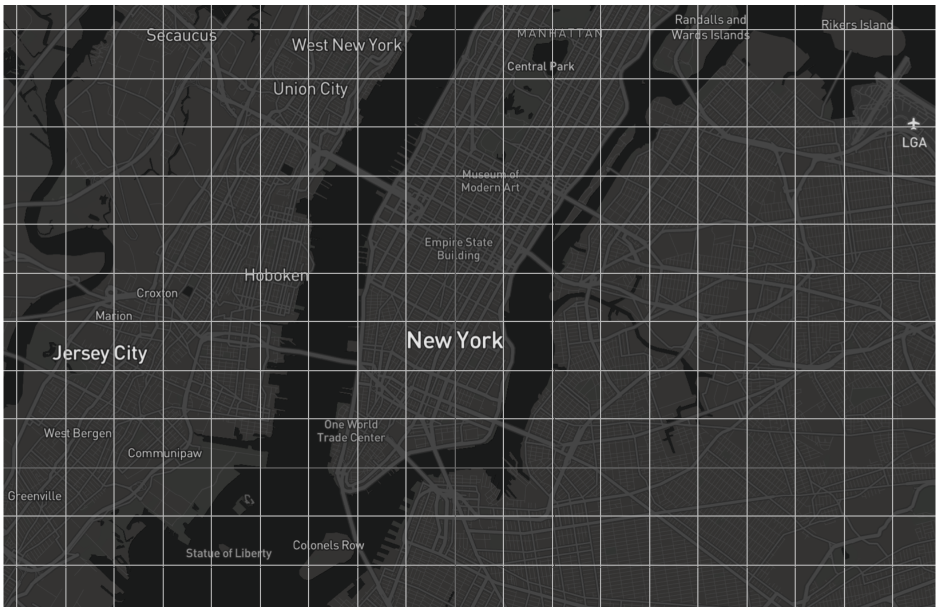


Figure 3.7: Segmentation overview for New York

Collection Phase

Global time window tw is set, during which every tile will be collecting passenger metrics, the most basic being

- Total passengers lost
- Average waiting time
- Number of pick-ups in the area

The size of the time window is a configurable metric, but a value which would allow us to respond quickly enough to changing trends is needed, such as 45 minutes or 1 hour. One can think of the tiles being stored as objects in a hash-map, as they have unique MGRS coordinates. In order to log the events, every time a passenger loses patience (is lost) or other event such as pick-up happens, we map the passengers coordinates to the correct tile in MGRS and find the tile in the hash-map. Every tile has access to a global clock, therefore synchronously every tw time units a **Proposal** phase of the algorithm can begin.

Proposal Phase

In the second phase, all tiles will need to propose a value based on the collected metrics. The proposed value is a result of a cost function which takes the metrics as inputs. The cost function can be as simple as directly returning the number of lost passengers, so the gravitational points would be created in areas where most passengers are lost or it can assign weights to different metrics and the resulting cost can be a combination of all metrics. Only requirement is that at the end of a time window, all tiles need to act according to the same cost function in order to ensure fairness. However, we might wish to change the cost function at different points in time. As an example, in the morning it could be more sensible to spawn gravitational points according to the average waiting time, but in the evening we would want to focus on having as least passengers lost as possible, therefore we would pick the passengers lost metric.

Further, we have a global set which holds the proposals from all tiles. Since the values are all computed by the same cost function, we can sort the set and start spawning points at tiles associated with the most critical values. This process happens in the **Dispatch** phase. An example execution can be seen in 3.8 below. The most red tiles represent areas with the highest proposed value and they have the largest chance of spawning a gravitational point.

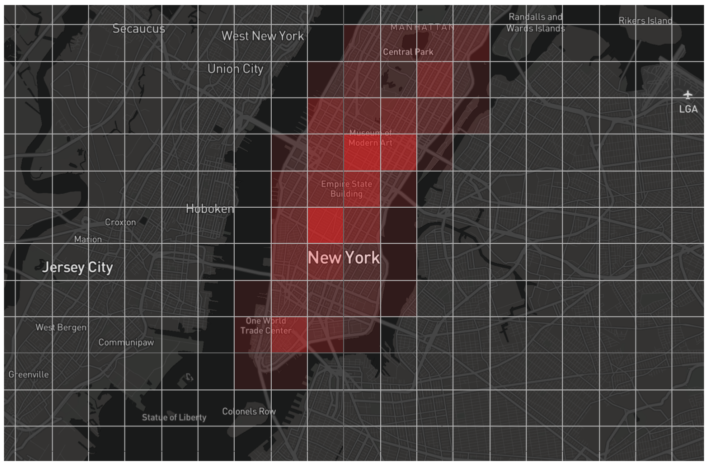


Figure 3.8: Heatmap of proposed values over Manhattan.

Dispatch Phase

Taking the global sorted set, we are now able to determine how many gravitational points we need to spawn and how many drivers to dispatch to each of these individual points. There are several ways to tackle this problem and we will be exploring the following two approaches.

- **Fixed number of points** - We can simply fix the upper number of points to be spawned in every **Dispatch** phase (e.g. 3 or 5). This needs to be an upper limit because the situation can be that we have more free drivers than there are needed in these tiles, therefore we may wish to serve only a subset of the selected points. Then, we take the sum of all proposed values of the top N tiles and for each of these values we calculate what is it's ”share” amongst the sum. This will return a float in the range 0-1 which we can multiply by the total number of free drivers and dispatch that many to the created gravitational point.

- **Dynamic spawning** - This type of spawning should be used with the lost passengers as cost function. It works it's way from the top of the sorted set to the bottom (starting from the largest value). For every proposed value, it looks at the number of passengers lost at the tile associated with the value, spawns a gravitational point and dispatches a number of drivers linearly proportional to the number of passengers lost at that tile. Linearly proportional because If a tile has lost N passengers, we don't need to dispatch exactly N drivers, we might want to dispatch less or more depending on what more suits the given situation. This approach can theoretically observe every single tile, but it is more likely that we run out of free drivers at some point during the execution, which is also when we need to stop. Therefore, the number of gravitational points is dependent on the number of free drivers we have available and the distribution of lost passengers.

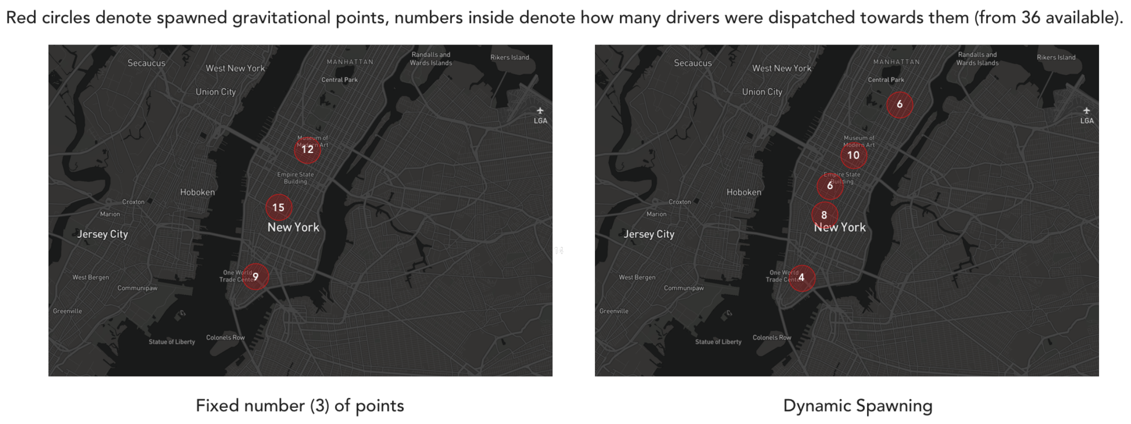

Red circles denote spawned gravitational points, numbers inside denote how many drivers were dispatched towards them (from 36 available).
12
15
9
New York
Jersey City
Hoboken
6
10
6
8
4
Fixed number (3) of points
Dynamic Spawning

Figure 3.9: Spawning of gravitational points under different approaches.

Starvation

One of the previously mentioned concerns which was not addressed in this approach is starvation. It is possible that many of the drivers will be directed towards gravitational points, leaving the outlying areas unable to be serviced, which can in turn have a harmful effect on the evaluated metrics such as the total number of passengers picked up. A solution to this is to always leave a subset of drivers stationary during the **Dispatch** phase. This means that in the Fixed number of points and Dynamic spawning approaches, we will initially not consider the set of all free drivers but a set with a reduced size, such as 80% of the whole set (it is a configurable parameter). As a result, a random subset of the free drivers will remain stationary in the outlying areas, available to serve the incoming requests there. It is important to sensibly choose the percentage of how many drivers remain stationary in order to satisfy the demand in the outlying areas while at the same time maximizing the gain from the spawned gravitational points.

Next Steps

Approach 1 works by analysing the trends of certain time window tw in the past. However, there is still a problem when it comes to the ”freshness” of the data, because the proposed value is based on the whole time window and it might be that the situation is not currently what it was a certain time before. Secondly, we need to account for the time it takes the drivers to arrive to the gravitational points which again can result in sub-optimal predictions. Ultimately, more robust system is needed which can observe not only the local trends but also the trends emerging from the historical data (to which we have access) and actually estimate what will be the demand at different areas in the future. We will cover this in the second approach which uses machine learning in order to predict the demand.

3.3.2 Approach 2

The second approach builds on top of the first one and collects the historical data about the demand in different tiles in a similar fashion. What differs is that a subset of tiles will now contain a trained neural network which will be used to predict the passenger demand in the near future. The predictions serve as an additional input to the cost function, computed during the **Proposal Phase**, in order to propose a more accurate value which would reflect the state of the fleet manager in the next time window closer to the reality. This section outlines the underlying machine learning problem of predicting the demand per every tile and the incorporation of the predictions into the existing system.

Using Machine Learning to Predict Tile Demand

The approximation of the demand in specific tiles from the 3.3.1 can be also posed as a machine learning problem. Given a sequence of length l, where every entry represents the demand in an area (tile in our case) for a certain time window tw, what will the demand be in the next time window. This problem deals with time series data and unlike a typical regression problem, time series adds the complexity of a sequence dependence in the input variables. To be more specific, we have an instance of a Univariate Analysis, because our sequential data will always contain only one variable, the number of requests in the tile. In order to capture the trends of the demand evolution over time, we will be using a Long Short Term Memory Network (LSTM), an instance of a Recurrent Neural Network (RNN). An LSTM is capable of learning and forgetting time series features specific to the trained problem. We will be discussing the data characteristics and the LSTM implementation details in the next sections.

Data

The dataset used to train the network is the Uber TLC FOIL Response, which contains 4.5 million Uber pickup coordinates from April to September 2014 [41]. The provided data has the form depicted in 3.1.

Header	Description
Date, Time	Date (MM/DD/YYYY) and Time (HH:MM) of the pickup
Lat	Latitude of the pickup
Lon	Longitude of the pickup
Base	TLC company code affiliated with the pickup

Table 3.1: Raw data structure from the TLC FOIL Response

Now, we map the request locations to the MGRS tiles, as described in 3.3.1 and in each tile we count the number of requests per every time window tw. For simplicity, throughout the rest of the examples, we will fix the tw to be 1 hour, so we will have a total of 24 entries per day. For each tile, the result will be an array of length 4392, because we have 6 months of data and in each month there are 24 data points for every day of the month. This is more closely depicted in the figure 3.10.

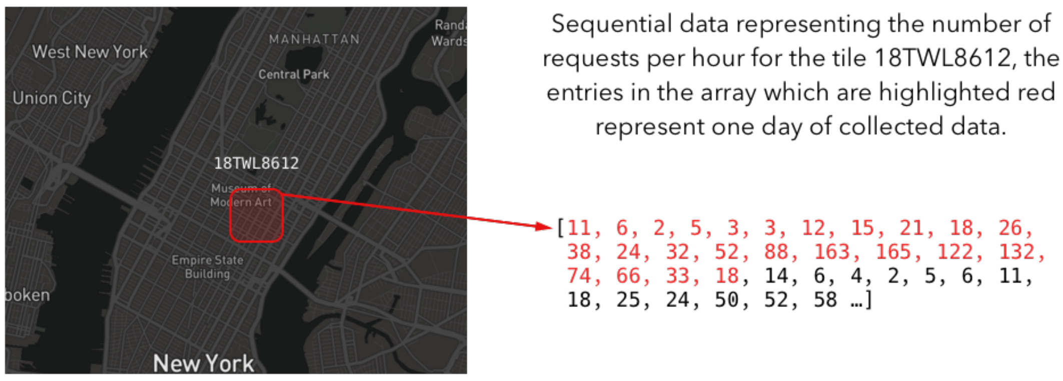


Figure 3.10: Transforming raw data into array of time series data points.

As this is a supervised learning task, the next step is to transform the sequential data into the form (X, Y) where X forms the input to the network and Y is the desired output. However, this depends on one of the hyper-parameters of the network, $look\_back$, which controls the size of the input to the network, or in other words, how far in our time series we look back. Therefore, for now we can treat it as a non-zero positive integer lb. One can think of the (X, Y) dataset transformation as having a sliding window of size lb, where X are the elements in the sliding window and Y is the next element outside of the window.

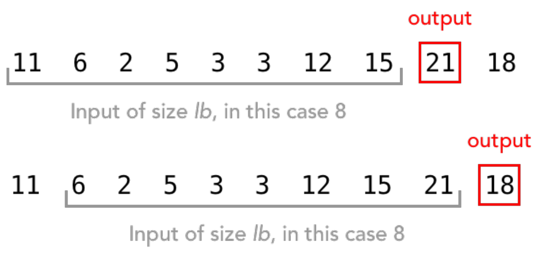

output
11 6 2 5 3 3 12 15 21 18
Input of size lb, in this case 8
output
11 6 2 5 3 3 12 15 21 18
Input of size lb, in this case 8

Figure 3.11: Sliding window data transformation

The example bellow illustrates an X,Y splitting of a small dataset where lb has been again set to 8. The result is an array of pairs.

```
Dataset
[11, 6, 2, 5, 3, 3, 12, 15, 21, 18, 26, 38]

(Input, Output)
( [11, 6, 2, 5, 3, 3, 12, 15], 21 )
( [6, 2, 5, 3, 3, 12, 15, 21], 18 )
( [2, 5, 3, 3, 12, 15, 21, 18], 26)
( [5, 3, 3, 12, 15, 21, 18, 26], 38 )
```

When every tile has the (X, Y) labelled data ready, the final step is to rescale the values to a [0,1] range in order reduce the sensitivity of the model to the magnitude of data. The MinMax scaling shown below was used, and it should be noted that the rescaling needs to be done on the whole dataset, not on disjoint subsets separately.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Data in this form can be split to training, validation and test batches (in our case, the ratio is 8:1:1).

Model

The model used throughout the project is a Long Short-term Memory (LSTM) network consisting of 3 layers. The justification for the choice of an LSTM was explained in 3.3.2. A standard LSTM would have a single LSTM layer followed by an output layer. However, we prefer to infer more abstractions about the behaviour of the demand for a longer period of time (6 months in our case, because of the available data) and therefore we will be using the Stacked LSTM architecture. This means that we have multiple LSTM layers stacked on top of other, in our case it is a LSTM layer with 32 blocks and a hyperbolic tangent as the activation function followed by a LSTM layer with 16 blocks with the same activation function. The final layer is a Dense layer which transforms the vector of length 16 from the previous LSTM layer to a single value. The input to the network is determined by the size of lb hyper-parameter, which is configurable, however the best performance has been shown under an lb of 32, which we will be using throughout the project. The model is shown in 3.12

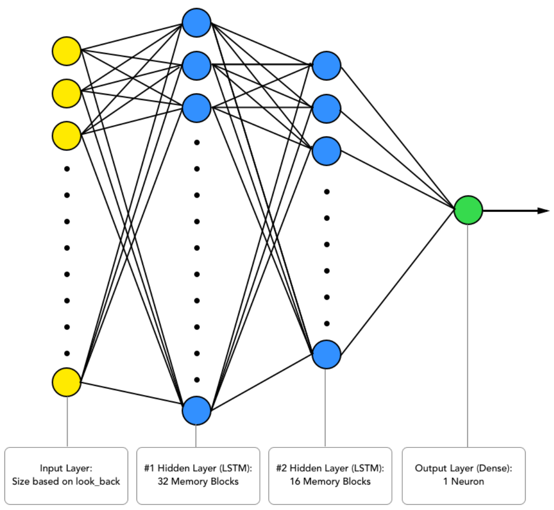


Figure 3.12: Model Architecture

Granularity of Trained Models

One major problem which I have immediately noticed was the low accuracy of predictions when a single general model was used for predicting the demand. This was simply caused by a large amount of variation in the tiles. As an example, some tiles experience heavy demand on Friday and Saturday nights, but otherwise they experience low or close to none demand. Therefore a single trained model would not be able to generalize effectively. An analogy would be using a single trained model to predict the value of any stock on the stock market. This is not feasible, and if we would like to predict the stock's value, we need a separate model trained on the historical data of the particular stock. Example of the behaviour is given in the below graphs showing the demand from a 1 week time slice on two different tiles.

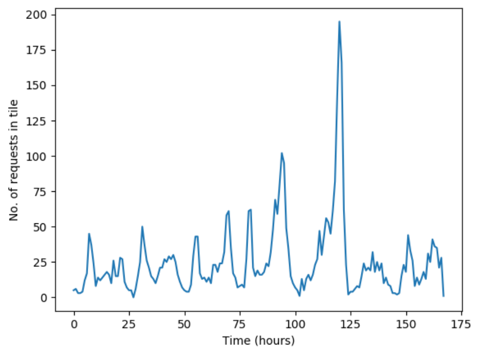


Figure 3.13: Tile 18TWL8508

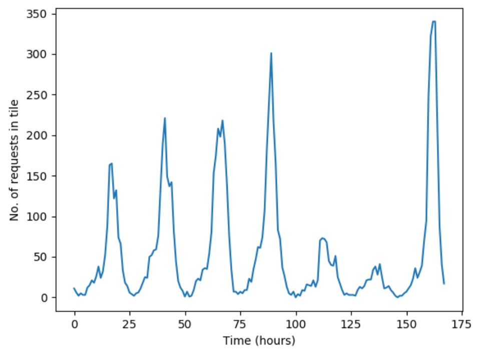


Figure 3.14: Tile 18TWL8509

A subset of tiles are chosen which will contain a machine learning model, trained on the tile's collected data. The model is then used to predict the demand for the next time window. The reason why only a subset of tiles is chosen is because a large portion of these tiles contain very sparse and inconsistent data, mostly because they are used only on certain days or weeks. Another large part are the unpopular tiles. These count tens of customers daily, while the numbers in the most popular tiles range in thousands, therefore their contribution to the overall ecosystem of the predictions is fractional.

In this project, I aimed to train the models which account for 70-80% of all the incoming requests. Tiles which do not contain a trained model fall back to the approach taken in 3.3.1. More about how these approaches cooperate is explained at the end of this Section.

Pipeline

The pipeline for supporting the training of tens or hundreds of models and their deployment to a fleet management simulator was an engineering challenge and understanding it will help the reader to see how the system predicts the demand in different areas. This section describes the overall training and deployment process.

1. **Training of a single model -** A single tile is chosen and a model is trained on the provided data. We are using the Root Mean Squared Error (RMSE) to monitor the loss.

   $$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_t - \tilde{y_t})^2}$$

   During this phase, we are trying to pick a set of hyper-parameters which will minimize the loss on the validation set. The model is trained using the Keras library and Hyperas library was used to speed up the hyper-parameter optimization process. The best performing set of hyper-parameters is determined, which later forms the basis for all the models.

2. **Tile selection -** Select a subset from the available tiles. Usually, tiles with incomplete data or low demand are excluded.

3. **Training of selected tiles models -** Now we need to load the available data, structure it and normalize it as has been described in 3.3.2 section. Per each of the selected tiles from the previous step, the corresponding dataset is built. Then, a model is trained with the hyper-parameters picked in the first step. Also, if we have a fixed Tile selection process and we always train the same subset of tiles, we can also compute the Aggregated Root Mean Squared Error (AMSE)

   $$AMSE = \frac{1}{m}\sum_{j=1}^{m}\sqrt{\frac{1}{n_j}\sum_{t=1}^{n_j}(y_{jt} - \widetilde{y_{jt}})^2}$$

   Which is a mean of all the tile's MSE from the subset. This helps us to know if we are decreasing the loss across all tiles as we are changing the set of hyper-parameters from the first step.

4. **Model deployment -** The models are saved locally as files (e.g. an h5) or uploaded and exposed in form of API endpoints.

Integration into the Fleet Manager

Now have two disjoint subsets of tiles, some with a trained model and some without. The tiles without a model should be able to contribute to the overall decision process of spawning a gravitational point at the same magnitude as the tiles with a model. In order to capture this behaviour, we will need to modify the cost function mentioned in the **Proposal** phase. In the Approach 1, the cost function is calculated using the available collected metrics. Now, for the tiles with a trained model, the cost function will have an additional metric available, named *next_demand*, which is a result of the prediction for the next time window. The cost function this project implements is

$$cost = next\_demand * (\frac{lost\_passengers_{t-1}}{total\_passengers_{t-1}})$$

The $t-1$ variables stand for the metrics in the previous time window. Or in other words, we assume that the ratio of lost passengers will be similar in the next time window, and we multiply it with the predicted amount of passengers, which should give us a more informed overview of the fleet manager's state. The tiles without a trained model contribute in **Proposal** phase exactly as before, by assuming that the demand will be similar to what it was in the previous time window.

3.4 Next Steps

Chapter 3 has covered the main concerns of a fleet manager, namely the problem of matching passengers with the drivers and distributing the drivers throughout the area as efficiently as possible.

For the matching problem, we have devised and optimized an algorithm in Section 3.1 which is responsible for matching a single passenger (or a group of passengers with a single drop off point) with a driver. The main contributions involved aggregating driver's several future passengers and optimizing the journey of the driver by reordering the passengers in the queue.

A more interesting and complex problem is when there are several passengers in the car with several drop off locations. In this case, we have discussed a static and a dynamic approach in Section 3.2. The optimization here was similar to the one in an exclusive taxi, yet we have instead considered exchanging a subset of the passengers between the drivers.

The scheduling was concluded by the Section 3.3 where we spawn gravitational points throughout the area where we are managing the fleet in order to match more passengers more quickly with the drivers. Firstly, we have discussed a basic approach considering only a certain time window in the past and later we have added more complex LSTM networks which are able to infer the trends from a larger set of historical data.

The next step is to create an environment, or rather applications where we can ”plug in” the algorithms we have implemented in order to later test them.

Chapter 4

Applications

After describing the theoretical approaches and their implementation details, we will now be looking into the details of the applications that were used to test and evaluate the behaviour of the mentioned approaches. Usually, when we want to evaluate the performance of a fleet manager and we don't have an access to a real fleet, we need to simulate the environment, the arriving passengers and the conditions present in the environment. Therefore, the first presented application is the Simulator. Using it, we are able to evaluate tens of thousands of passenger requests spanning across several days. The execution of the simulation time mainly depends on the complexity of the used algorithm. However, a major disadvantage of the Simulator is that we are not able to look at particular decisions made by the fleet manager inside the Simulator. This would be an extremely useful feature, as many times in our 3 section, we are expanding a single idea with many layers and extensions and we might not be able to determine why a particular decision was made.

We could for instance implement a logging system and and log every action the fleet manager inside the simulator takes. This way, we could track down the exact event and by studying the logs find out why such an approach was taken. However, this would still require a great deal of imagination, because the logs would contain a large set of coordinates and approximating the behaviour based on that might become an uneasy task. Instead, a better approach would be to visualize the behaviour. For this purpose, the Visualizer was created, which shows us a real time state of the drivers and passenger requests.

Both Simulator and Visualizer are described from their technological side in this section. We will discuss the challenges and decisions made throughout various choices, along with the limitations of the mentioned applications.

4.1 The Simulator

4.1.1 Overview

Thinking of the Simulator as a single application would not be correct in case of this project. More accurately, it is a framework to which different scheduling approaches can be plugged in and which can be customized to fit the needs of the scheduling problem at hand. However, the purpose for which the Simulator was created, to provide a basis for deterministic testing of the implemented algorithms on a large scale and as efficiently as possible, remains the same across all of the use cases in this project.

In total, there are 4 applications built on top of the Simulator framework which are underlined by the same event system, the same data set on which they execute and the same metric logging system. The separate applications are divided by the topic which they are addressing in Chapter 3, so there is the Exclusive Vehicle Simulator, Non-exclusive Vehicle Simulator, Gravitational Points Simulator and Machine Learning Gravitational Points Simulator. As the scheduling task they are performing is unique in every case, separating them into isolated applications made sense from the perspective of both functionality and maintainability.

This Simulator is composed of two main parts, the data which it parses and feeds to the fleet manager and the fleet manager itself.

4.1.2 Data

The data set has been previously described in detail in Section 3.3.2. The data set which simulator uses is the same, with the exception of having a separate data set for spawning the drivers. Therefore, each of the passengers is identified by a request time and a request origin, all of them arriving in the correct time order. One significant change which was made is that the time is not more expressed in a format of HH:MM:SS, but rather all times are converted solely to seconds. This is because we would need the seconds anyway during our calculations, so instead we normalize all the values to seconds while parsing the data. As an example, a passenger does not send his or her request at the time 01:00:10, but at time 3610.

4.1.3 Distance Calculation

The calculation of distances, usually between two points specified by their latitude and longitude, is left out to be implemented by the plugged scheduling approaches. The approach used in this project is to initially calculate the distance using the haversine formula (explained in Section 2.4.1), for example when comparing large lists of passengers and drivers. When the best match based on the haversine distance is picked, we issue a request to an open-source router (OSRM directions API [42]) in order to get the exact data about the trip's length and time.

4.1.4 Fleet Manager

The main feature of the Fleet Manager are the state changing events. We could implement the fleet manager to loop through every second of a day and execute for example a whole month of scheduling, but we would waste a large amount of resources, simply because there are gaps of time when nothing happens with the state. Instead, we wish to only undertake an action when an event occurs and set the current time of the simulation according to the time when the corresponding event has happened.

A state changing event can be in every of the mentioned use cases of the Simulator something different, for example an event would be a spawning of a new gravitational point, when we would try to find drivers which can move towards the point to have more success of finding a passenger. We would emit an event for the selected drivers to move from their location to the location of the gravitational point. However, throughout the whole simulator framework, we have always 2 guaranteed state changers, a new passenger request and a finished drop-off of passenger(s). In case of a new passenger, we would immediately try to find a driver which can be scheduled to pick the passenger up, which in turn produces another events, a journey to the passenger pick up and drop off. In case of a driver who just finished his or her journey, we may want to look around the finished location to see if there are not any potential passenger matches.

As was mentioned, the Simulator framework ensures a correct working of the event based scheduling. More exactly, it is ensured by having a global MinHeap where all events are inserted and which, if requested, always pops the next most immediate event. We plug in a unique scheduler algorithm or technique into it which differs in the way how and when it emits the event for passengers pick-ups and drop-offs.

The core functions of the fleet manager are the following:

- `start()` - Starts the whole simulation loop which terminates only if there are no other events in the global MinHeap. The `start` is preceded parsing the data set and building the passengers and drivers list.
- `new_request()` - Simulates incoming of a new request by taking the most immediate passenger from the list of future requests (built by parsing the data set).
- `handle_event(event)` - After we pop an event from the MinHeap, we pass it to `handle_event` and determine what actions should the fleet manager take. Here we can implement the state changing events logic or we could add a new event type to handle.

4.2 The Visualizer

4.2.1 Overview

The Visualizer is a web based application for displaying the real-time state of the fleet. It displays an interactive map with all the customers, agents, ongoing trips alongside with real-time updates to metrics such as total distance travelled or how long have the passengers waited. The initial intention behind building it was the visual help when I was trying to understand some of the decisions made by the scheduler, but it later evolved into a piece of software easily usable by a person responsible for overlooking the fleet. As can be seen on the Figure 4.1, the passengers are denoted with a triangle and the drivers can have an orange path assigned to them which is either the path to pick up a passenger or a path to drop off the passenger.

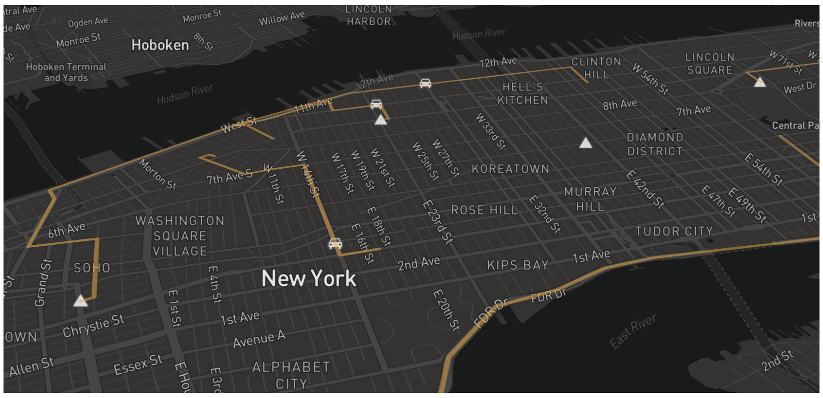


Figure 4.1: Visualizer overlooking a fleet in New York

The Visualizer is composed of three main parts, the front-end, the fleet manager and the request emulator. All of these can be though of as different services listening on different ports and communicating together through a common API, or in this case WebSockets [43], which we will be discussing in later sections. As usual, the front-end displays the decisions made by the Fleet Manager on an interactive map and reports the accumulated metrics. Usually, the passenger requests would be based on a location determined by a GPS module in a passenger's mobile phone. However, as was mentioned, our data does not arrive from real users, therefore we need a separate service which will simulate the requests incoming in real time. For this specific use case, the next part of the Visualizer is the Request Emulator. All of the components with the technologies used to implement them are depicted in 4.2.

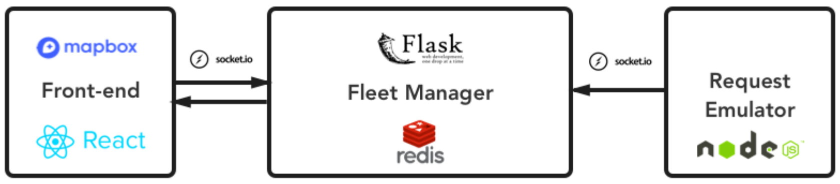


Figure 4.2: Overview of the Visualizer components

4.2.2 WebSockets

When wiring the communication between different services, one challenge was how to handle a large volume of data sent in constant streams. The majority would be flowing between the front-end and the fleet manager. WebSockets provide a reliable bi-directional communication between the front-end and the server, and the fleet manager can be in this case though of as a server. Every time we have a state change or a new journey was scheduled, we immediately send it to the front-end to render the changes. Thanks to this, we don't have to waste resources on techniques such as polling.

4.2.3 Front-end

The front end is a React [44] application built of two main components. The most dominant component is the Map, which as the name suggests, display the current state of the fleet manager on a map. In this project, the Mapbox [45] solution was used which offers a React component displaying the map and a simple API used for adding, animating and removing symbols on a map. The symbols are used to display the drivers and the users. Another part of the API are layers, these are used to display the journey paths. Then we have the sidebar component, which has a WebSockets connection established with the Fleet Manager and listens to reported metrics and updates them accordingly. React was chosen because of the Mapbox support and because we are updating the Document Object Model (DOM) with tens or hundreds of passengers or drivers at every second, therefore we need an efficient library for handling this kind of volume.

One important thing to note is that the Fleet Manager works based on events, it makes a match and the journey is set a starting position and an ending position. Therefore, the drivers position is only updated at a drop-off, the position is not updated continuously in-between, as this would pose a significant computational burden. This also affects the front-end, mainly the Map component. The component has only access to the start and to the end, it does not know anything regarding the path.

The first step to solving this is a request to the OSRM directions API [42]. The fleet manager uses the API in its calculations of the most optimal path, however it does not pass the information along to the front-end. Then, it will pass forward a part of the response, more particularly it is the waypoints. Waypoints are composed of the latitude, longitude pairs of all turns in a path from point A to point B. This will effectively allow us to build the path on the client side, and can be used to animate the drivers along the path. Unless we would want to make several hundreds of requests per every animated path, we are not able to simulate real traffic situations and real distances between two waypoints. Instead, we will use the Haversine distance of two waypoints and determine to what portion of the total journey distance it corresponds. Then, we take the total time of the journey in seconds and multiply it by the calculated portion floating point value we have, and we get the time for how long we need to animate a car on a straight line. This way, we can build the whole animation.

4.2.4 Request Emulator

The emulator first parses a data file of passenger requests where each of the entries must have a time when the request arrived and the position of the request. The emulator then emits an event using WebSockets to the Fleet Manager. The emulator in this project was written in Node.js [46], however it is a fairly small component and a wider range of technologies could be potentially used to build the emulator.

4.2.5 Fleet Manager

The Fleet Manager is the hearth of the Visualizer. It tracks the current state of the drivers and passengers, based on which it then schedules journeys. The state of the fleet manager is usually changed by 2 main actions, either a new passenger arrives or a driver finishes his or her journey. More specifically, it is a Flask [47] server listening for WebSocket events (such as a new user arrives) or the server itself is emitting events to the front-end, describing the scheduling decisions. The state needs to be always saved in a database, in this case we are using Redis [48]. Any of the mentioned implementations can be customized to fit into the scheduling function of the Flask Fleet Manager, however they were primarily written to work with the Simulator. That also justifies the choice of Flask, to keep the same programming language (Python) across the Simulator and the Visualizer for the fleet manager.

4.3 Next Steps

We have now covered the details of the algorithms in Chapter 3, further we will wish to observe their behaviour and determine the performance. This can be done using the tools that were developed in Chapter 4. Therefore, the ultimate step is to connect both chapters and evaluate the proposed approaches. We will be looking into how and under what conditions the algorithms affect the metrics which we have chosen to evaluate their performance.

Chapter 5

Evaluation

In this chapter, I aim to evaluate the performance of the algorithms implemented in Chapter 3. Similarly to the third Chapter, here we will also have 3 main sections each tackling a different area, more specifically being the Exclusive Taxi, Shared Taxi and Gravitational Points.

We will be always looking at 3 metrics, the number of Lost Passengers, Average Waiting Time and the Total Distance Travelled. We will simply be trying to greedily minimize these metrics as they are all directly linked to the effectiveness of the fleet manager. Of course, this would require a definition of what an 'effective' fleet manager should do, but in our case it would be to maximize a theoretical revenue by serving as many passengers, as quickly as is possible. For conducting the evaluation, the Simulator application and it's data set, described in Section 4.1 is used. It should be noted that different parts of the data set (or different random seeds) might have been used in different sections of the evaluation (e.g. the NVD used in Chapter 5.1 is executed on a different part of the same data set as the NVD in Section 5.3). Of course, they remain consistent within the sections.

The behaviour of driver agents (drivers) throughout this project has been simplified. We assume that we have a fixed set of drivers who are always available and don't have an option to reject an assigned passenger. They are all available, without any passengers, at the start of the scheduling process, at positions which were extracted from the used data set. This makes the evaluation more accurate and reduces the involved non-determinism.

Most importantly, we will be evaluating all of the algorithms in terms of driver to passenger ratio. Because we are able to force the fleet manager into extreme, or, in contrary, relaxed situations, by simply varying the ratio, it's an effective way of determining under what conditions might it be beneficial to use (or not use) some of the approaches. Similar evaluation technique is used when determining the effectiveness of ride-sharing systems [49] and it also helps in a more practical manner, by providing the operators of fleet managers valuable information about what might be the best amount of drivers to aim for under the circumstances of their fleet managers. More about the exact technical details of what ratios we are going to use is described below.

There are two main features which we will be varying when determining the behaviour of the implemented approaches.

- **Request Density** - We are simulating a 24 hour time window during which requests will arrive and the fleet manager will operate. The request density can be controlled in terms of how many ride requests will arrive within these 24 hours.

  According to the used Uber data set, the daily number of passengers ranges from roughly 8000 to 20000, depending on which period of time we look at. We will use the upper limit of 20000 passengers as the basis for a 'normal' request density. Then we will also want a more extreme density, to capture a system which is stressed. The fleet manager is expected to behave differently in each of these settings. This will be set to 40000 requests per 24 hours, two times more than a normal usage, or in other words a request arriving approximately every 2.2 seconds.

- **Driver to Passenger Ratio** - The next configurable parameter is the driver to passenger ratio, which we will limit between 0 and 0.5, with implications that the smaller the ratio, the worse the overall performance in terms of the evaluated metrics will be. It would be possible to have a ratio larger than 0.5, however such a case would be highly unlikely to occur in a real situation, as the passengers outnumber the drivers in large numbers. Also, after the threshold of 0.5, there is not much difference in the decisions the fleet manager takes because it almost always has the majority of fleet available. On the other side of the spectrum are extremely low ratios which force the system to take more calculated and planned decisions. Therefore we will test the implementations with smallest ratios starting from 0.01, 0.05 and 0.1. Then, we will incrementally increase the ratios by 0.1 until 0.5, ultimately resulting in 7 tests [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5].

5.1 Exclusive Taxi

The problem of scheduling a taxi where only one passenger can be present at a time is directly linked to Section 3.1. As was discussed in Chapter 2, it is hard to determine what is the state-of-the-art scheduling approach in on-demand fleet managers, mostly because such information is considered confidential or it is such a complex system that it can't be expressed in terms of a single algorithm and it's optimizations. However, we can observe that the open source implementations use the Nearest Vehicle Dispatch (NVD) to perform scheduling [9] and since we are interested in greedy approaches, the NVD portrays an ideal candidate because it does not try to plan ahead to the future, it just greedily chooses the current best passenger driver match.

Referring back to Section 3.1, we will be comparing the NVD with IH, which is the Insertion Heuristic without the optimization step and to IH (O) where we turn on the queue optimization. Firstly, we will look at the results in a more global overview, with larger intervals between ratios and then we will inspect the behaviour in more granular intervals to find out what might be the best situation to implement the mentioned algorithms.

5.1.1 Normal Density

In the normal density, we simulate 20 000 incoming requests and the ratio of drivers towards this number is expressed in the *D/P Ratio* column of the Figure 5.1.

D/P Ratio	**Passengers Lost**			**Avg. Waiting Time**			**Distance Travelled**		
	NVD	*IH*	*IH (O)*	*NVD*	*IH*	*IH (O)*	*NVD*	*IH*	*IH (O)*
0.01	11633	10150	10068	538	939	941	86.19	87.02	86.90
0.05	128	127	126	159	181	182	154.98	151.84	151.77
0.1	89	75	75	114	137	137	149.53	149.01	149.01
0.2	28	41	41	86	92	92	147.36	146.71	146.71
0.3	20	27	27	56	70	70	145.94	145.40	145.40
0.4	17	18	18	45	65	65	145.26	144.92	144.92
0.5	16	13	13	44	58	58	144.83	144.82	144.82

Table 5.1: Exclusive Taxi : Evaluation of measured metrics under normal request density

The results confirm our assumption that the IH variations outperform the NVD approach, mainly because they have access to a larger set of drivers. However, in case of the total passengers lost, one might expect the IH to outperform the basic NVD approach in every single case, but it can be seen that in ratios 0.2 or 0.3, that is not the truth. All of the values differ fractionally, except for the first case which is also where a large gap between ratios 0.01 and 0.05 was created. The rest of the results behave in a more linear and controlled manner, proportional to the ratio. To understand why such a gap was created, we examine the Table 5.1 depicting the Passenger Lost metric which showed the most fluctuation. The ranges start from 200 drivers, corresponding to 0.01 ratio, up until 1000 drivers, which is 0.05 ratio.

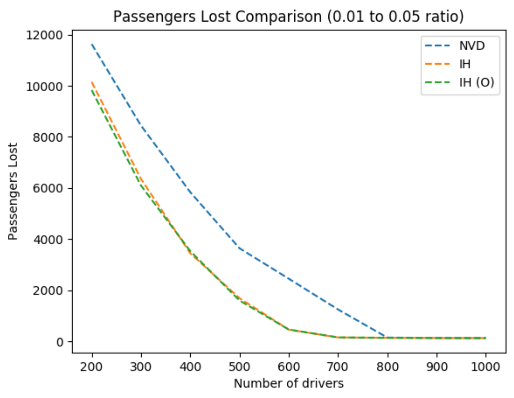


Figure 5.1: Graph of lost passengers in lower ratios - normal density

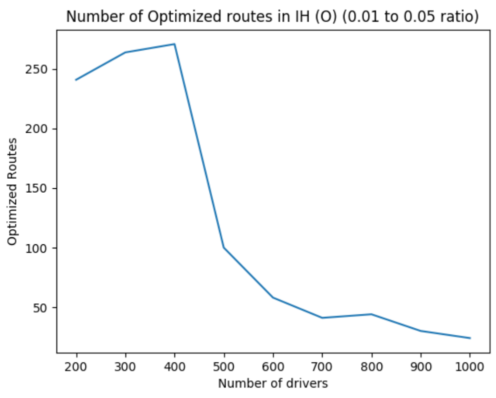


Figure 5.2: Number of optimizations done by IH (O)

In case of the passengers lost, we can see from Figure 5.1 that at around 800 drivers, all approaches converge to the same level, with some small fluctuations which can be further followed in Table 5.1. The reason why the implementations converge is that there are enough drivers to accommodate the demand, even if we don't plan ahead at all and stick with a basic approach such as NVD. Difference lies in the amount of drivers when a given approach starts to get close to the local optimum. The Figure 5.1 shows that IH and it's optimized version start converging sooner, roughly at a point of 600 drivers. That is simply caused by the ability to leverage extra resources in terms of the busy drivers. However, such an optimization does not come for free. The average waiting time for every single instance of IH approaches has been higher than the one of NVD. The IH is more likely to make more complex trips, where we have a queue of pick up and drop offs, ultimately resulting with the passengers at the end of the queue to have larger waiting times, which in the end increases the total average of the waiting times.

As for the travelled distance, in the first case we see larger distance travelled for IH and IH (O) which actually should optimize the distance better than NVD does. They actually do optimize it, the larger numbers are simply caused by the IH approaches picking up more passengers, therefore travelling more. After that, the IH approaches yield always smaller results than the NVD. The common intuition is that if the drivers drive shorter distance, then they deliver more customers. Referring back to Table 5.1, we can say that this assumption is false. In some cases, the IH shows worse performance in terms of passengers lost even if it has less distance travelled than NVD. However, these differences are fractional and occur only occasionally, therefore we can't make any definite conclusions, but we should still keep in mind that local decisions at time t, which IH can pick 'better' than NVD, do not always mean better decisions in the long term.

We can also observe that the difference between IH and IH (O) is only present at the minimal ratio levels, then they both behave the same. Remember that the IH (O) only works if the drivers have a queue long enough to be able to optimize on, and it generally applies that the longer the queue the more options we have for generating the most suitable permutation. What happens as we add more drivers is that we simply have a larger pool of free drivers and we are not forced to pick a driver who is currently delivering a passenger, which in turn reduces the average size of queue per every driver. This behaviour can also be seen on Figure 5.2, where initially we perform over 250 optimizations because we are heavily constrained by the lack of drivers and we have many passenger queues which have a space large enough to be optimized. The number decreases towards 0 as we keep adding drivers. It should be also noted that the distance which the optimized algorithm saved is fractional, in this case ranging from tens to hundreds of kilometers.

The general decreasing trend of all approaches is also expected. As we keep adding drivers, the numbers in all tables decrease because of the extra influx of options introduced by new drivers. In conclusion, it can be said that the IH approaches outperform NVD in terms of the passengers lost and distance travelled, at the cost of the previously mentioned waiting time and also the larger computational complexity. The best use case for either IH approach is a heavily stressed system which has a low ratio, with the optimized version expected to outperform the normal one.

5.1.2 High Density

In the high density, we simulate 40 000 incoming requests and the ratio of drivers towards this number is expressed in the *D/P Ratio* column of the Table 5.2.

D/P Ratio	**Passengers Lost**			**Avg. Waiting Time**			**Distance Travelled**		
	NVD	*IH*	*IH (O)*	*NVD*	*IH*	*IH (O)*	*NVD*	*IH*	*IH (O)*
0.01	23254	20003	19932	543	950	955	174.82	178.93	178.25
0.05	194	169	167	166	210	211	308.79	302.38	302.36
0.1	91	85	85	131	153	153	298.83	297.01	297.01
0.2	37	42	42	95	112	112	292.53	291.63	291.63
0.3	31	31	31	59	64	64	289.88	288.57	288.57
0.4	26	25	25	48	62	62	288.89	288.19	288.19
0.5	18	18	18	46	55	55	288.74	288.11	288.11

Table 5.2: My caption

When simulating two times as many requests, we can observe similar results to the ones seen during the normal execution. The higher density affected the metrics only in a way which was expected. We observe higher numbers in terms of lost passengers, higher waiting times and more distance was travelled. The gap after the first result is also present, in this case with even a larger difference.

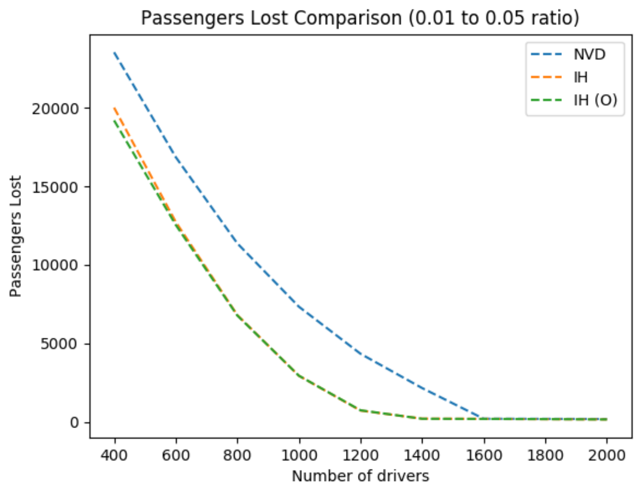


Figure 5.3: Graph of lost passengers in lower ratios - high density

Similarly as in the normal case, the graph in Figure 5.3 follows a declining trend which eventually converges to the same level in all of the cases. The threshold when the convergence happens is also approximately located at a ratio of 0.04, therefore we can't conclude that the highly dense case would show any different trends than the previous findings have shown.

5.2 Shared Taxi

Scheduling a shared taxi, where *N* multiple users up to the maximum vehicle capacity can be travelling to *N* different locations, is described in the Section 3.2. The evaluted implementations are based on the approaches used in an on-demand transportation company Lyft, described in [13]. Therefore, we will wish to understand and evaluate under what conditions might it be beneficial to use the implemented algorithms, along with determining how introduced parameters, such as a pool time window, affect the behaviour of the system.

The basis of this section is the Static AGD, denoted in the tables and graphs simply as AGD. The first extensions is the Dynamic AGD, which allows detours for the busy drivers to pick up a nearby passenger, denoted as DAGD. Lastly, we presented a Route Exchange optimization, which is implemented on top of the DAGD in this project (but does not necessarily need to be so, it will also work on top of the Static AGD), denoted as REAGD.

Along with the described optimizations and extensions, we have also introduced a parameter which significantly affects the results. It's the pool time window, or the time after which the pool of gathered users should be processed. Setting it too low would result in having sub-optimal matches as we would not be able to accumulate enough passenger, but we would have an advantage of low waiting times of the users as they would be dispatched more frequently. On the other side, larger pool time windows allow us to accumulate more users and make more suitable matches, at the cost of larger waiting time and, potentially, more passenger losses. There is an extra layer of complexity in how does the pool window affect the proposed extensions which we will be discussing in sections below. In the following examples, we will use a pool time window of 5 minutes, however we will be conducting experiments to see how does this number affect the performance.

Referring back to Section 3.2, one of the proposed optimizations was also the longitudinal sorting. As this one does not affect the performance, just the computational time, we will look at a separate evaluation section devoted to it.

5.2.1 Normal Density

In the normal density, we simulate 20 000 incoming requests and the ratio of drivers towards this number is expressed in the *D/P Ratio* column of the Figure 5.3.

D/P Ratio	**Passengers Lost**			**Avg. Waiting Time**			**Distance Travelled**		
	AGD	*DAGD*	*REAGD*	*AGD*	*DAGD*	*REAGD*	*AGD*	*DAGD*	*REAGD*
0.01	7624	11538	10933	715	1020	1012	90.53	78.02	76.94
0.05	955	1202	1176	799	1047	1041	144.39	137.24	135.91
0.1	668	580	571	808	950	938	190.74	194.54	194.03
0.2	654	576	572	756	883	872	195.91	193.97	192.44
0.3	610	584	584	757	886	884	196.38	192.92	191.31
0.4	620	517	515	772	888	882	195.42	193.31	192.54
0.5	603	501	500	748	864	864	197.05	192.96	191.12

Table 5.3: Shared Taxi : Evaluation of measured metrics under normal request density

The results are again very similar to the ones presented in the previous section on Exclusive taxis, with the difference that we can see a clear dominance of AGD in the first two cases. AGD has outperformed the DAGD and REAGD in terms of the passengers lost and average waiting time metrics, however not in distance travelled. On the other hand, the distance metric is understandable, as distance travelled is directly tied to how many passengers we pick up, therefore it should be higher. However, the question still remains, how does a naive approach significantly outperform the 'advanced' versions at first and then falls slightly behind but keeps a linear pace? The answer to this question is closely linked to using the passenger pool and processing it every x minutes. The DAGD and REAGD keep the drivers busy, since they keep adding new and new detours to their task schedules, thus leaving us with fewer free drivers who would be able to pick up the passengers from the pool every time it is needed. This is an undesired behaviour at lower ratios, where we actually prefer to keep the drivers ready to take care of the user from the pool. However, once we hit a certain threshold when we will always have enough drivers to take care of the pools, we are ready to assign extra load to the busy drivers in order to maximize our profits. In the example of above Table 5.3, this is somewhere between the ratios 0.05 and 0.1, where DAGD and REAGD start outperforming the AGD in terms of passengers lost. As it would be expected, after the threshold the performance has stabilized and we observe a linear decrease at every ratio (since we have more drivers available).

Thee Average Waiting Time has the lowest values in the AGD and experiences slightly higher levels in the other two approaches. Drivers in both DAGD and REAGD make extra detours which they are able to tightly fit into their already busy schedules, therefore increasing the average waiting time for the passengers.

The DAGD and REAGD achieve smaller travelled distance than the AGD. This is understandable initially, as they lose more passengers, but later they maintain the lower levels even though their performance in terms of picked up passengers increases. This is caused by the dynamic approaches taking the closest detours which already share a large portion of the driver's trip's path.

If we isolate the dynamic approaches and compare their performance, it becomes clear the the Route Exchange optimization has shown improvements in all three measured greedy metrics. The improvements are many times fractional (or none) since when we are exchanging the routes between drivers, we exchange them based only on one path (which is seen as the green line on Figure 3.6), which often gives us a small amount of extra value. The previously mentioned evaluations are linked to the currently used pool time window and we will be further determining how does the time window affect the execution in Section 5.2.3.

5.2.2 High Density

In the high density, we simulate 40 000 incoming requests and the ratio of drivers towards this number is expressed in the *D/P Ratio* column of the Figure 5.4.

D/P Ratio	**Passengers Lost**			**Avg. Waiting Time**			**Distance Travelled**		
	AGD	*DAGD*	*REAGD*	*AGD*	*DAGD*	*REAGD*	*AGD*	*DAGD*	*REAGD*
0.01	15248	18538	18122	718	1048	1022	182.34	172.58	170.95
0.05	1908	2202	2076	810	997	988	291.27	280.93	273.48
0.1	736	740	729	770	984	975	374.52	366.38	365.81
0.2	708	631	642	746	1015	994	380.11	375.36	368.22
0.3	655	584	567	761	953	942	378.45	373.52	371.05
0.4	630	535	522	749	940	941	378.95	376.11	373.49
0.5	631	539	525	748	931	928	379.04	378.01	374.15

Table 5.4: Shared Taxi : Evaluation of measured metrics under high request density

The most significant observation is the general performance trend of all AGD approaches under higher density. In the first case of normal density, we have seen a roughly 40-55% passenger loss, here we are in the area of 35-45%. This effect also applies to the next ratios. Besides that, we can't conclude much new from the Table 5.4. The trends remain more or less the same, AGD outperforming both advanced approaches in first two cases and then falling slightly behind (however, never in terms of average waiting time). Under higher density, there are more opportunities to optimize on, therefore the gaps between DAGD and REAGD have increased in favour of REAGD.

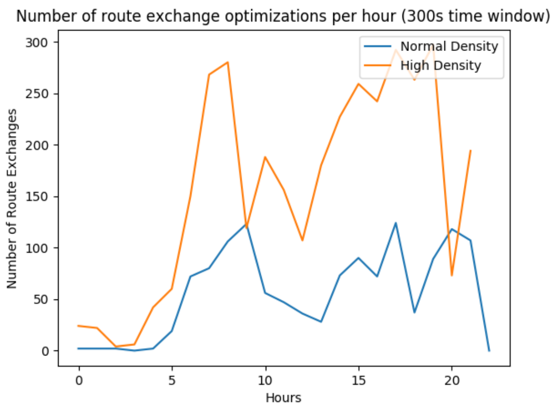


Figure 5.4: Number of Route Exchange Optimizations per hour : Comparison of Normal and High demand (300s time windows, 0.1 ratio)

Such behaviour can also be observed in Figure 5.4 where we can also clearly see that the number of route exchanges follow a usual pattern, i.e. they are highest in the morning, when usually people request a taxi to work, and similarly so in the evening.

It is possible that someone could think of REAGD in a similar way as of the IH (O) from the previous Exclusive Taxi evaluation in Section 5.1. In a way, that's true, because both approaches optimize on small parts of scheduled journeys, just in different environments. However, there is one major difference, which we have seen in the evaluation of IH (O). As we increase the driver to passenger ratio, it becomes that IH and IH (O) converge to same performance levels as there are enough drivers to accommodate all passengers and IH (O) does not have a queue large enough to optimize on. We don't see the converging trend here because REAGD always has journeys which it can optimize on, thanks to the default implementation of AGD which creates 3 or 4 person long trips, thus having routes large enough to be potentially exchanged.

5.2.3 Pool Time Window

In order to determine how the pool time window affects the overall scheduling process, we will look at 4 different time windows (75s, 150s, 225s, 300s). The simulations are performed under a normal density of 20 000 users with different driver to passenger ratios. The REAGD has not been included in the tests as it is only an extension of DAGD and can be expected to behave similarly.

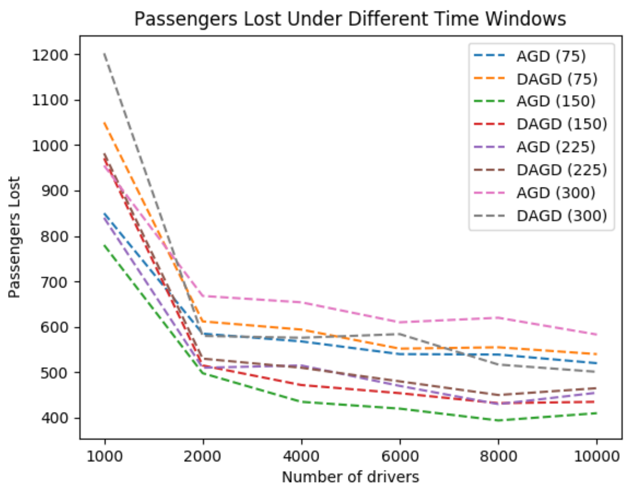


Figure 5.5: Graph of lost passengers throughout different time windows

A certain time window represents a threshold when the AGD starts to outperform the DAGD (in this case it is between 300 and 225), which was not the observed phenomenon in Section 5.2.1. The performance of DAGD is moderately affected by the time windows changes, in a positive trend. This makes sense as most of the drivers are usually busy because of the dynamic detours and more frequent pool processings give us more chances to assign drivers to pools. On the other hand, the more significant impact was on the side of AGD. It seems that in our case, the more frequent pool windows benefit the AGD's performance because it is able to send out drivers more frequently, at the cost of less optimal routes, which in this case is not a problem as the best performing choice is an AGD with time window of 150 seconds. As would be expected, there is another extreme in the realms of 75 second time windows, where both of the algorithms performed worse than almost all of the other implementations, except the AGD with 300 seconds time window. This clearly shows us that there is a point between 75-150 seconds when the pools are too small to make good passenger driver matches.

In practice, the best approach would be to look at the time windows with even a larger granularity, approximately having them 10 or 20 seconds apart. We can't say that a pool time window of length n is always to most optimal one, as this might be very specific to the fleet or to the area where the scheduling is happening.

5.2.4 Distance Sorting

The Distance sorting, or in this case more specifically longitudinal sorting, is an optimization which aims to improve the computational efficiency. The tests were done on a normal simulation with 20 000 users, 1000 drivers and with a pool time window size of 5 minutes. The results were measured using the `time` python package, on a machine with 3.1GHz Intel Core i7 processor with 16 GB 1867 MHz DDR3 RAM. We report the average time needed to process a pool and a maximum and minimum time across all processed pools. The instance where longitudinal sorting is turned on includes the time it needs to sort the pool.

Longitudinal Sorting	Average Time	Max. Time	Min. Time
No	8.21s	26.77s	0.16s
Yes	3.29s	24.91s	0.47s

Table 5.5: Comparison of pool computational efficiency with and without the longitudinal sorting optimization

As can be seen in the Table 5.5, the version with longitudinal sorting outperforms the naive one. A slice from the algorithm's execution is on the Figure 3.5, where we can see instances of pools considering very few passengers, thus being able to break from the loop after looking at just 1-2% of nearest passengers. Another notable difference can be seen in the higher minimum time when longitudinal sorting is enabled, which is most likely caused by the sorting of the pool, since it takes actually more time to sort it than to process it. Overall, we have not improved the general complexity, which is still n^2, however we have introduced significant computational efficiency savings.

5.3 Gravitational Points

The Gravitational Points are concerned with a different scheduling problem than the ones we have already observed in the previous sections of Evaluation. We are trying to move the drivers to locations which will bring more value, later in time. Gravitational Points are completely disjoint from approaches such as IH or AGD as they don't create passenger driver pairings, they schedule solely drivers and they complement IH, AGD or any of the previously used scheduling techniques. Therefore, we need to pick an algorithm (or several algorithms) which we will use as a basis. The evaluation will then be performed by comparing the algorithm with and without spawning gravitational points. As this optimization does not interfere with the core scheduling process, it should not make a significant difference when picking what algorithm should be used as a basis for the gravitational points testing, it is applicable to all approaches. As a result, we can use the simplest NVD and observe how does Approach 1 and Approach 2 from Section 3.3 affect it's decisions. At first, we evaluate both approaches separately, as one is statistically based and the other is machine learning based, and we would like to explore different features of each. Then, we will evaluate an example where we compare Approach 1 against Approach 2 and observe what are the strengths and weaknesses of each.

In the previous sections, we have evaluated the algorithms in 24 hour long time windows. As some of the days from the data set had less than 20 000 or 40 000 passengers arriving in a day, we had to normalize several days into a single instance of single 24 hours. Now, we will not need to worry about this, we simply let the number of passengers determine how many days of scheduling we need. The change is required because of how we process the data set which we feed into the machine learning part of the scheduler. As was mentioned in Section 3.3.2, we create a total of 4392 training/validation/testing examples, one for each hour of the data set. The relation is violated if we squeeze several days into one. This would result in a large data inconsistency which affects the model used to predict the future demand, as it would suddenly see data with completely different patterns as it was trained on. Therefore, we will also not be able to simulate a high density of requests, just the density which the data set offers.

In section 3.3.2 we described one of the hyper-parameters of the LSTM network, the look back, which represents the amount of time the network looks back in order to predict the next time window demand. We have set it to 32, which means that we will need at least 32 hours of previous scheduling data in order to be able to feed it to the network. Intuitively, this means that we need to be simulating at least 33 hours, preferably more, in order to evaluate the Approach 2. We will actually set the number of passengers to 50 000, an equivalent of approx. 4 days of simulation, in order to observe the behaviour of both approaches.

5.3.1 Approach 1

D/P Ratio	Passengers Lost		Avg. Waiting Time		Distance Travelled	
	NVD	*NVD (GP)*	*NVD*	*NVD (GP)*	*NVD*	*NVD (GP)*
0.01	18900	10763	528	479	344.60	470.30
0.05	2422	1665	183	176	420.47	499.16
0.1	2237	1559	160	157	417.13	492.38
0.2	1923	1368	123	119	411.58	479.07
0.3	1606	1121	90	89	407.19	466.55
0.4	1355	903	65	63	404.36	455.37
0.5	913	596	47	46	407.97	445.82

Table 5.6: Gravitational Points : Evaluation of measured metrics under Approach 1

The results have shown a performance increase for every measured ratio. This was an expected outcome, however it would be very hard to predict how big percentage gains does spawning of gravitational points produce. In terms of the passengers lost, it can be observed that the NVD (GP) performs roughly 28-43% better than a classic NVD, depending on the situation. On a smaller scale, we can also see improvements of the average waiting time, approximately by 1-9%. The reasoning is simple, since we are able to a certain degree tell where the next passengers will show, we can move subsets of our fleet there which in turn improves these two metrics. However, moving the fleet comes at a cost, the travelled distance, which performed worse on the part of NVD (GP). Part of the increased distance travelled can be accounted to picking up more passengers, but it is certain that a large portion was contributed by drivers moving to the gravitational points. This behaviour can be shown by calculating the distance per picked up passenger, which can be obtained directly from the provided data. In ratio 0.01, we can see that NVD travels 11.08km per passenger and NVD (GP) travels 11.99km, precisely 0.91km more, which is 8.2% more.

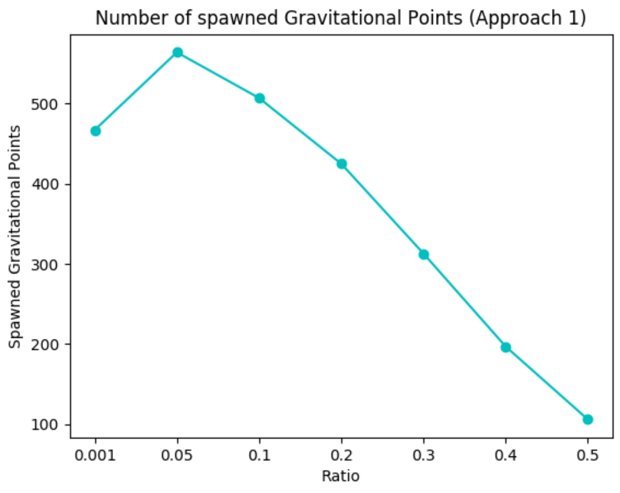


Figure 5.6: Number of spawned Gravitational Points for different ratios in Approach 1

If we go into more detail, the largest impact of NVD (GP) can be seen in the first test, which might at first sound counter-intuitive. We would expect the NVD (GP) to perform worse on lower ratios since the drivers would be driving to the gravitational points instead of picking up customers. We can afford such behaviour in the higher ratios where we have enough drivers to cover the demand and drive to gravitational points. However, we must not forget that we dispatch only free drivers to the gravitational points. In the lowest ratio, most of the drivers are busy because of the driver's shortage and therefore only a small amount of drivers move towards gravitational points every time they are spawned. In order to confirm this assumption, we can again observe the distance travelled per picked up passenger. As we have already calculated, the NVD (GP) at 0.01 ratio sees an 8.2% increase, when compared to NVD. Comparing again NVD (GP) to NVD at the 0.05 ratio, we can see a 16.85% increase in the distance travelled per picked up passenger, which means that the drivers at the 0.01 ratio travel more than two times less to gravitational points while still being highly effective. The ability to position the already limited number of drivers smartly introduces a significant decrease in both passengers lost and average waiting time metrics.

A larger number of drivers in the fleet manager results in a higher chance that an area already has a sufficient number of drivers present. As a result, we observe lower percentage differences between NVD and NVD (GP) as we increase the ratio. This trend is also confirmed by the decreasing number of spawned Gravitational Points on Figure 5.6. While in general we can expect that more spawned points mean a higher performance (because they are always spawned for a reason), we can see one outlier at the ratio 0.05. The reason why the percentage gain in terms of the measured metrics is lower is simply the fact that a spawned gravitational point is clearly more valuable to a constrained system than to a relaxed one.

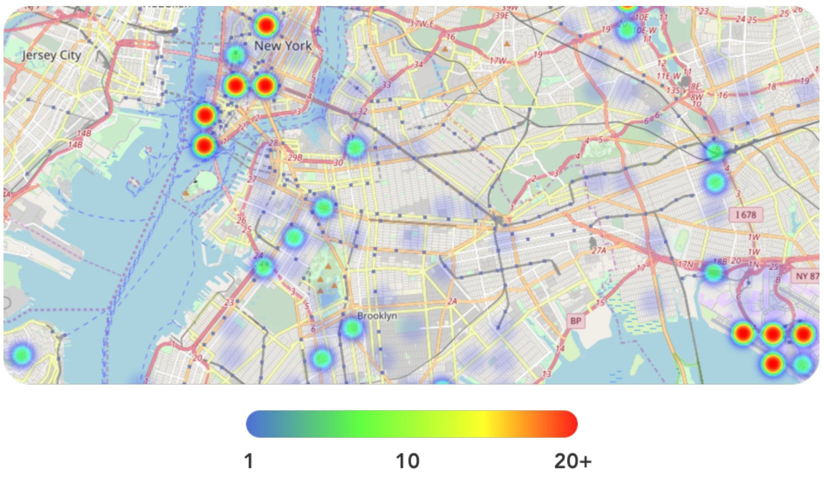


Figure 5.7: Heatmap of a subset of spawned Gravitational Points in Approach 1. The legend displays the number of times a point was spawned at a tile. This is a run with 50 000 passengers and 5000 drivers.

Figure 5.7 was generated at the end of one of the evaluation runs and we can use it to observe what are some typical gravitational points in a part of our scheduling area. There are 2 major areas. First is the lower side of Manhattan, which is a popular touristic and a business district thus experiencing heavy demand almost non-stop. Secondly, with the location in the bottom right corner, there is the John F. Kennedy International Airport, also operating non-stop and being the starting point of a large subset of the passengers list. Besides that, we see more subtle, but still significant spawned points throughout the residential areas such as Brooklyn, with a larger concentration closer to Manhattan. The points are spawned according to a user defined cost function, as described in Section 3.3, therefore this heatmap represents just a single instance out of a wide range of possibilities (to be exact, this heatmap was constructed by the cost function being the number of lost passengers), but it can be still seen that the spawned points follow locations with high passenger density.

5.3.2 Approach 2

First, we will discuss the evaluation of the underlying machine learning problem. Since we are using tens of models, or more exactly 47, we need to measure the performance in terms of Aggregated Root Mean Squared Error (ARMSE), as was described in Section 3.3.2 of the Implementation. On this data set, the 47 tiles have an average of 17.90 passengers incoming per hour. We will be using this metric for comparison with the network's predicted values.

Every tile's data was split to a training, validation and testing instance (in ratio 8:1:1) and we will be reporting the RMSE or ARMSE based on their performance on the testing subset. The table below displays the results, with ARMSE being an average taken from all tiles and RMSE representing just one tile (in this case one with the smallest and one with the largest error). The units of RMSE (or ARMSE) are the same as those in the model's input, therefore they are free to interpret as 'the prediction missed on average by *x* passengers'.

Metric	Value
Avg. Number of Passengers Per Hour (across all tiles)	17.90
ARMSE	2.91
Min. RMSE	1.83
Max. RMSE	4.29

Table 5.7: Evaluation of measured metrics on the trained models per every tile

The ARMSE shows that every prediction is on average off by **2.91 passengers** from the reality. Taking into account the average number of passengers per hour, we can expect a 16% deviation from a predicted value. If we take the average prediction error of Approach 1 for the same subset of tiles, we get an average miss of **6.22 passengers**. This expressed in percentage is a 35% deviation per every predicted value, therefore a 16% error is certainly an improvement. Further, the min and max RMSE show us that there was a tile where we missed only by 1.83 passengers, or inversely missed by more than 4 passengers. These higher and lower numbers are simply caused by the different characteristics of the tiles. Tiles experience different passengers loads and the predictions can have values ranging from 0 to more than 300. Therefore the higher RMSEs belong to tiles which predict on average larger values since they are prone to generate larger errors. The same applies to the lower RMSEs.

Now we can now evaluate the effects of applying machine learning powered prediction to the spawning of Gravitational Points.

D/P Ratio	**Passengers Lost**		**Avg. Waiting Time**		**Distance Travelled**	
	NVD	*NVD (GP)*	*NVD*	*NVD (GP)*	*NVD*	*NVD (GP)*
0.01	18900	10742	528	479	344.60	468.03
0.05	2422	1648	183	176	420.47	497.75
0.1	2237	1533	160	157	417.13	490.79
0.2	1923	1334	123	118	411.58	478.77
0.3	1606	1066	90	88	407.19	465.66
0.4	1355	845	65	63	404.36	456.07
0.5	913	559	47	45	407.96	448.28

Table 5.8: Gravitational Points : Evaluation of measured metrics under Approach 2

The results shown in Table 5.8 are very similar to the ones observed in previous Section 5.3.1. This is understandable, as Approach 2 builds on top of the first one. We also need to remember that Approach 2 can only be executed after the first 32 hours have been schedules and data was collected (because we have set the $look\_back$ parameter to 32) and during that time, Approach 2 falls back to being Approach 1. The general effects, such as less passengers lost or more distance travelled on the side of NVD (GP) has already been reasoned about in the previous section and it applies also here, because of Approach 2 working in a similar manner as the Approach 2. Therefore, we will continue by describing the differences between these two approach in the next section.

5.3.3 Comparison

The results from Approach 1 are under the columns marked NVD (GP). Results from Approach 2 are under NVD (GP-ML).

D/P Ratio	**Passengers Lost**		**Avg. Waiting Time**		**Distance Travelled**	
	NVD (GP)	*NVD (GP-ML)*	*NVD (GP)*	*NVD (GP-ML)*	*NVD (GP)*	*NVD (GP-ML)*
0.01	10763	10742	479	479	470.30	468.03
0.05	1665	1648	176	176	499.16	497.75
0.1	1559	1533	157	157	492.38	490.79
0.2	1368	1334	119	118	479.07	478.77
0.3	1121	1066	89	88	466.55	465.66
0.4	903	845	63	63	455.37	456.07
0.5	596	559	46	45	445.82	448.28

Table 5.9: Gravitational Points : Comparison of Approach 1 and Approach 2

In all of the measured metrics, the NVD (GP-ML) performs better than the NVD (GP). The most interesting feature is that the gravitational points in the evaluation of Approach 1 and Approach 2 improved the passengers lost metric at the cost of distance travelled, because the drivers had to drive extra to these points. Now, we actually don't see this behaviour. We observe fewer lost passengers and at the same time, less distance travelled. The reason for this is intuitive, NVD (GP-ML) has only a 16% prediction error, while the NVD (GP) has a 35% error (discussed in previous section). As a result, the NVD (GP-ML) makes more accurate and calculated predictions, resulting in less gravitational points, which are more effective. The assumption is also confirmed if we look at the Figure 5.8. The NVD (GP-ML) achieves better results with fewer gravitational points spawned. It should be also noted that both NVD (GP) and NVD (GP-ML) algorithms don't only decide where a point is spawned, but they also propose the number of drivers which should move to the location, which ultimately gives them more control over the fleet manager.

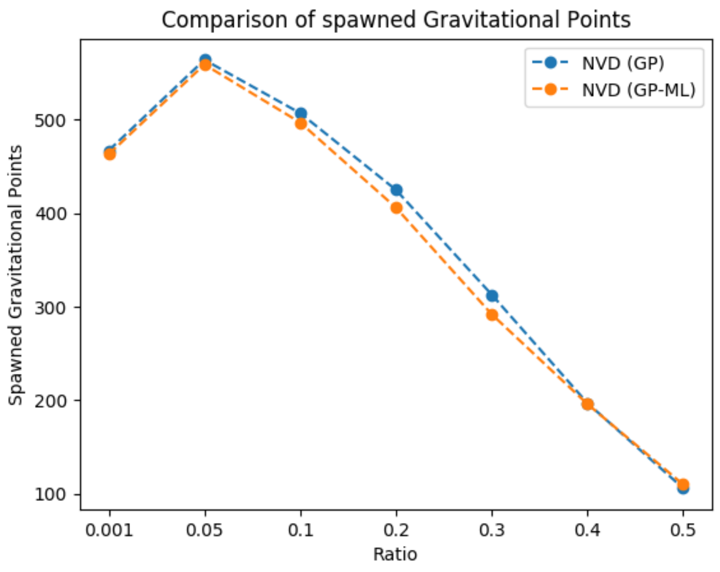


Figure 5.8: Heatmap of a subset of spawned Gravitational Points in Approach 1. The legend displays the number of times a point was spawned at a tile. This is a run with 50 000 passengers and 5000 drivers.

Even though we have demonstrated better performance in the NVD (GP-ML), the resulting numbers are many times only marginally better. This is because an error of 16% still introduces a significant noise to the predictions. Now, the machine learning problem is an instance of univariate analysis where we are predicting the future demand based simply on the levels of demand 32 hours before. However, if we would transform the problem to a multivariate analysis, we would be able to infer much more about the demand. Including information such as humidity, wind speed or traffic conditions significantly affect people's decisions on whether to take a taxi or not. Unfortunately, this project lacks the data to perform the extended analysis, but it is certainly an extension which is expected to introduce improvements to the existing model of gravitational points.

5.4 Challenges

Besides the challenges I have encountered during the implementation of the approaches, there were two other, tied more to the evaluation.

- Making sense of the results once a simulation was complete (more importantly when the results deviated from what was expected).
- Defining an approach which can be used to evaluate all of the implemented algorithms uniformly.

The first challenge was mainly tackled thanks to the Visualizer, described in Section 4.1, an application which helped me to visualize the execution of the fleet manager and observe it's decisions on smaller instances. However, even then the results were sometimes hard to grasp, after which I have generated graphs to help me understand the evolution of metrics throughout the simulation. The graph generation was performed using a Python package Matplotlib [50].

The second challenge is related to how the algorithms concerning the Vehicle Routing Problem (VRP) are evaluated. If we look at the current research, we would observe that the algorithm's performances are benchmarked on two standartised sets of problems. The first one is the Solomon's problem set [51] and the Gehring & Homberger's extended benchmark set [52]. The problem with both of these sets is that they have been created in order to estimate the performance of a Vehicle Routing Problem with Time Windows (VRPTW) and the passengers in the sets range from 25 up to a maximum of 1000 passengers, while the drivers are usually spawned in number of tens. Our on-demand fleet manager works on a much larger scale and it is a modified instance of a Dynamic Vehicle Routing Problem with Time Windows (DVRPTW). We need to test the system during several days of simulation with tens of thousands of passengers and drivers.

As a solution, we have used an approach tied more specifically to our problem [49] [53] which is based on the ratio of drivers and passengers further described at the start of Chapter 5. We also leverage the fact that we have access to the Uber's data set and we don't need to generate random passengers requests throughout the area, which wouldn't allow us to implement algorithms such as the spawning of gravitational points, as they need a certain pattern to follow. The is also an option to perform this inversely, to fix the number of drivers are vary the number of passengers, but it is not expected to perform with much difference.

Chapter 6

Conclusion and future work

This chapter aims to give a concise summary of the work done in this project, alongside with possible future improvements or work which can be done to extend the content of the project.

6.1 Conclusion and Contributions

- Implementation of the NVD and IH algorithms, which tackle the problem of scheduling an exclusive taxi. Further reasoning about possible optimizations of IH and their subsequent implementation.
- Implementation of the AGD algorithm and it's optimizations used for scheduling of shared taxis, according to the Lyft Engineering [13].
- Design and implementation of a non-standard scheduling approach in form of Gravitational Points. The project proposes 2 different ways of spawning the gravitational points and dispatching drivers to them, each demonstrating significant improvements in the measured metrics.
- Creation of The Simulator framework used to simulate an execution of a fleet manager and The Visualizer for visualizing the decisions made by the fleet manager.
- Comparison and throughout analysis of all of the above mentioned approaches, evaluated on a real world data set under different conditions controlled by the driver to passenger ratio.

6.2 Future Work

This section is divided into two parts, first one describing the plans which can be implemented by directly extending the work I have done. The second part includes more ambitious and complex proposals which still overlap with the topic discussed in the project, but would require more time to complete.

6.2.1 Immediate Extensions

- **Multivariate analysis in Approach 2** - First mentioned in Section 5.3.3, the performance of Approach 2 can be improved by enriching the data we provide to our LSTM networks. We have measured a prediction error of 16% under an instance of a univariate analysis. The prediction error is expected to decrease if we would pose the machine learning problem as a multivariate analysis, as we would be able to include additional information along with the time series data of the demand, such as the wind speed, humidity or information regarding the traffic situation.

- **Events Prediction in Approach 2** - When we are spawning gravitational points and we use the neural networks to predict the demand in the next time window we always act to some kind of a linear pattern seen in the past. However, there are millions of events (such as concerts, conferences, exhibitions or many more) going on every day. It is expected to see a larger passenger demand at the start and end of these events, which in turn introduces an unexpected element to our predictions. Usually, the exact location and time of these events can be known beforehand, and we can even estimate the size of passenger influx which such an event would produce. The next step would be to extend our system with a part which can find out about such events in a scheduled area and alter the predicted demand according to the size, time and location of the event.

- **Add Non-deterministic Elements** - It would be interesting to see how non-deterministic elements such as a driver or user rejecting the scheduled match or driver logging out at the end of a shift affects the performance of the proposed implementations.

- **Area Clustering** - As described in Section 2.4.3, the fleet manager does not operate on a global scale as this would cause extremely high processing and memory demands. It rather works on a basis of larger cities or areas. The Area Clustering extension would explore what are the demographic aspects which determine what shapes and sizes of an area are most optimal for deploying a fleet manager.

6.2.2 Further Extensions

- **Gravitational Points for Users** - We are already spawning gravitational points for drivers, why not use the same approach to suggest the users a nearby location where they can move in exchange for a cheaper ride? This way, we would be able to aggregate users to common locations which would help us schedule and predict the demand even more effectively. We would need an 'auctioneering' system because also want to determine what are good conditions of giving such an offer to the user. It can also further tackle common problems such as one way lanes, the driver might need to make an extra detour of several kilometers just because the user is in the middle of a one way lane, however if we would be able to ’nudge’ the user to the end of the lane we would introduce a benefit to both parties.

- **On-demand food delivery** - Food delivery shares many common traits with an on-demand ride transportation network. The main difference is that we would be solving a different VRP problem, as we would have many restaurants scattered throughout an area where the driver needs to stop first before arriving to the ’passenger’. Also, since the requests first arrive to a restaurant where they need to cook the food, we are also given a larger time window to plan.

Bibliography

[1] Ride-sharing users statistics. https://www.statista.com/outlook/368/100/ride-sharing/worldwide/. (accessed 28 January 2018). pages 1

[2] How does ride-hailing compare with taxis? https://lasvegassun.com/news/2016/jun/13/how-does-ride-hailing-compare-with-taxis. (accessed 1 February 2018). pages 1

[3] Michael E. Webber F. Todd Davidson. Using only uber or lyft is cheaper than owning a car for 25americans. http://www.businessinsider.com/uber-or-lyft-could-be-cheaper-than-owning-a-car-2017-10. (accessed 30 January 2018). pages 1

[4] 13 cities that are starting to ban cars. http://www.businessinsider.com/cities-going-car-free-ban-2017-8. (accessed 28 January 2018). pages 1

[5] https://www.uber.com/partner/madd/. (accessed 30 January 2018). pages 1

[6] Uber official website. https://www.uber.com. (accessed 30 January 2018). pages 1

[7] Lyft official website. https://www.lyft.com. (accessed 30 January 2018). pages 1

[8] Lyft - how are passengers and drivers paired? https://help.lyft.com/hc/en-us/articles/115012926847. (accessed 27 January 2018). pages 1, 17

[9] How the matching algorithm works in the on-demand economy? http://nextjuggernaut.com/blog/matching-algorithm-works-demand-economy-part-three-user-journey-series/. (accessed 28 January 2018). pages 2, 10, 55

[10] San franciscos love affair with ride-hailing. https://www.drivesweden.net/en/smart-mobility-news-and-comments/san-franciscos-love-affair-ride-hailing. (accessed 28 January 2018). pages 2

[11] Heres why business travelers are ditching taxis. http://time.com/money/4543758/business-travelers-ditching-taxis-uber-lyft/. (accessed 28 January 2018). pages 2

[12] Is uber slipping in the ride-hailing market? `https://www.forbes.com/sites/michaelgoldstein/2017/10/24/is-uber-slipping-in-the-ride-hailing-market`. (accessed 28 January 2018). pages 2

[13] Timothy Brown. Matchmaking in lyft line. `https://eng.lyft.com/matchmaking-in-lyft-line-9c2635fe62c4`. (accessed 29 January 2018). pages 5, 17, 27, 31, 59, 72

[14] `http://www.autonews.com/article/20171105/INDUSTRY_REDESIGNED/171109910/future-lies-in-fleet-management`. (accessed 30 January 2018). pages 6

[15] Wilson N. and Colvin N. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959. http://www.jstor.org/stable/2627477, (accessed 23 January 2018). pages 7

[16] G. B. Dantzig and J. H. Ramser. Computer control of the rochester dial-a-ride system. Technical report, MIT, 1977. (accessed 23 January 2018). pages 7

[17] H. N. Psaraftis. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*, 14(2), 1980. (accessed 25 January 2018). pages 8

[18] Ingrid Lunden. Uber says that 20 percent of its rides globally are now on uberpool. `https://techcrunch.com/2016/05/10/uber-says-that-20-of-its-rides-globally-are-now-on-uber-pool/`. (accessed 27 January 2018). pages 8, 22

[19] `https://twitter.com/uberuk/status/425594403210924033`. (accessed 30 January 2018). pages 8

[20] A. L. Medaglia Pillac V., Gendreau M. Ch. Guret. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013. https://www.cirrelt.ca/DocumentsTravail/CIRRELT-2011-62.pdf, (accessed 26 January 2018). pages 9, 13

[21] et. al. Ichoua S. Diversion issues in real-time vehicle dispatching. *Transportation Science*, 34(4):426 – 438, 2000. https://pubsonline.informs.org/doi/abs/10.1287/trsc.34.4.426.12325, (accessed 25 January 2018). pages 9

[22] Khouadjia M. R. and Sarasola B. et. al. Metaheuristics for dynamic vehicle routing. *Studies in Computational Intelligence*, 433:265–289, 2013. (accessed 25 January 2018). pages 9

[23] Varone S. and Janilionis V. Insertion heuristic for a dynamic dial-a-ride problem using geographical maps. *Optimisation et Simulation*, 2014. (accessed 25 January 2018). pages 10

[24] J. Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573 – 586, 2003. (accessed 30 January 2018). pages 11

[25] Glpk. `https://www.gnu.org/software/glpk/`. (accessed 26 May 2018). pages 12

[26] Carpooling is totally coming back this time. `https://www.citylab.com/transportation/2017/09/is-carpooling-making-a-comeback/539979/`. (accessed 30 January 2018). pages 12

[27] Robert Geisberger, Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Vlker. Fast detour computation for ride sharing. *OpenAccess Series in Informatics*, 14:88–99, 01 2010. (accessed 26 May 2018). pages 13

[28] A.E. Rizzoli A.V. Donati R. Montemanni, L.M. Gambardella. Ant colony system for a dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 10:327343, 2005. (accessed 26 January 2018). pages 14

[29] Indemand official website. `https://tryindemand.com/`. (accessed 30 January 2018). pages 17

[30] Juggernaut official website. `http://nextjuggernaut.com/`. (accessed 30 January 2018). pages 17

[31] Uberrush official website. `https://rush.uber.com`. (accessed 30 January 2018). pages 17

[32] `https://www.quora.com/How-does-Uber-assign-rides-to-drivers`. (accessed 29 January 2018). pages 17

[33] jsprit. `https://github.com/graphhopper/jsprit`. (accessed 30 January 2018). pages 18

[34] Open-vrp. `https://github.com/mck-/Open-VRP`. (accessed 30 January 2018). pages 18

[35] Hipster4j official website. `http://www.hipster4j.org/`. (accessed 30 January 2018). pages 18

[36] Optaplanner official website. `https://www.optaplanner.org/`. (accessed 30 January 2018). pages 18

[37] Libretaxi. `https://github.com/ro31337/libretaxi`. (accessed 30 January 2018). pages 19

[38] Opencabs. `https://github.com/rtnpro/opencabs`. (accessed 30 January 2018). pages 19

[39] Libretaxi fms. `https://github.com/ro31337/libretaxi/blob/master/src/response-handlers/submit-order/notify-drivers-response-handler.js`. (accessed 30 January 2018). pages 19

[40] Optimizing a dispatch system using an ai simulation framework. https://www.uber.com/newsroom/semi-automated-science-using-an-ai-simulation-framework/. (accessed 31 January 2018). pages 21, 33

[41] Uber tlc foil response. https://github.com/fivethirtyeight/uber-tlc-foil-response. (accessed 12 April 2018). pages 39

[42] Osrm api documentation. http://project-osrm.org/docs/v5.5.1/api. (accessed 23 May 2018). pages 47, 50

[43] Websockets documentation. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. (accessed 27 May 2018). pages 49

[44] Reactjs official website. https://reactjs.org/. (accessed 27 May 2018). pages 50

[45] Mapbox official website. https://www.mapbox.com/. (accessed 27 May 2018). pages 50

[46] Node.js official website. https://nodejs.org/en/. (accessed 27 May 2018). pages 51

[47] Flask official website. http://flask.pocoo.org/. (accessed 27 May 2018). pages 51

[48] Redis official website. https://redis.io/. (accessed 27 May 2018). pages 51

[49] R. Jayakrishnan Neda Masoud. A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system. *Transportation Research Part B: Methodological*, 106:218–236, 2017. (accessed 17 May 2018). pages 53, 71

[50] Matplotlib official website. https://matplotlib.org/index.html. (accessed 20 May 2018). pages 71

[51] Solomon's problem set. http://w.cba.neu.edu/~msolomon/problems.htm. (accessed 20 May 2018). pages 71

[52] Gehring and homberger benchmark. https://www.sintef.no/Projectweb/TOP/VRPTW/Homberger-benchmark. (accessed 20 May 2018). pages 71

[53] J.Y. Park J. Jung, R. Jayakrishnan. Design and modeling of real-time shared-taxi dispatch algorithms. *Transportation Research Board*, 2013. (accessed 20 May 2018). pages 71