**Bachelor Thesis**

# Optimisation of static and dynamic demand on a feeder line

Andrea Roso

| | |
|---|---|
| Date of Submission: | 17. June 2024 |
| Advisors: | Prof. Dr. Marie Schmidt |
| | Kendra Reiter, M.Sc. |

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

# Contents

# 1 Abstracts

## 1.1 Zusammenfassung

In dieser Bachelorarbeit wird die Simulation von [9] erweitert, um Abkürzungen darstellen zu können sowie statische und dynamische Nachfrage abbilden zu können. Es werden zwei Heuristiken definiert, ein Greedy-Algorithmus beruhend auf der Arbeit von [9] und ein Clustering-Algorithmus. Daraufhin wird verglichen wie gut der Greedy-Algorithmus für die statische und dynamische Nachfrage abschneidet im Vergleich zum Clustering-Algorithmus für die statische Nachfrage im Verbund mit dem Greedy-Algorithmus für die dynamische Nachfrage. Dabei werden gleichmäßig verteilte Anfragen von 10 bis hoch zu 300 Anfragen berücksichtigt mit 1, 4, 7 und 10 Bussen. Der Anteil der statischen Nachfrage liegt fest bei 60%, der Anteil der Richtungen liegt bei 50%.
Nach Auswertung der Ergebnisse legt nahe, dass es in diesem Szenario keinen Unterschied zwischen den beiden Algorithmen gibt. In Chapter 7 wird darauf eingegangen, in welchen möglichen Szenarien es einen Unterschied geben könnte.

## 1.2 Abstract

In this bachelor's thesis the simulation by [9] was expanded by shortcuts and reflecting the combination of dynamic and static demand. Two heuristics were defined, a greedy algorithm based on [9] as well as a clustering algorithm. The greedy algorithm used for dynamic and static demand was compared to the clustering algorithm for static demand connected to the greedy algorithm for dynamic demand. The requests are uniformly distributed, ranging from 10 requests up to 300 requests. The number of buses varied from 1, 4, 7 and 10. The percentage of static demand was fixed at 60%, while the percentage of directions was distributed equally.
After evaluation of the results, it seems plausible that in this scenario, no single algorithm is better. In Chapter 7 scenarios and possible improvements to the clustering algorithm are suggested, to further analyse if there are differences between the two algorithms.

# 2 Introduction

There already has been a lot of study on the dial-a-ride problem, in which customers request transportation from a pick-up location to a drop-off location with several different variations. Most systems have in common, that they are on-demand, so that the buses only drive to or stop at certain stops if customers requested it and not just because they are on their schedule. This requires information from the passengers like their timeframe or the location [10].
These dial-a-ride problems can be classified according to multiple properties. One property is the location of the stops. There are variations that look at stop-based systems, in which the buses can only stop at predefined locations, and variations that look at door-to-door systems, in which the customers can freely request any location. In this thesis, we will take a closer look at a stop-based system, more specifically a feeder line, in which either the pick-up location or the drop-off location is defined as the train station[10]. Requests from a stop to the train station are defined as outbound and requests from the train station to a stop are defined as inbound as proposed by [2].
Furthermore the timing can be used as classification. In the static scenario, the schedule is fixed after planning is completed and can't be changed. In a dynamic scenario, it can be changed even during operations. In the dynamic offline scenario, it can change until a service has started. In the dynamic online scenario, it can change after a service has already started [10]. There is also a distinction regarding the availability of a pre-planned schedule. Fully flexible systems don't have anything preplanned, whereas in semi-flexible systems an existing schedule is changed to fit new demand [10]. In this thesis, the combination of fully-flexible static and semi-flexible dynamic-online timing will be the focus.
Lastly, the objective of the optimisation can be used as a property: minimizing the passenger travel time is called passengers' perspective whereas minimizing costs or maximizing profits is called operator's perspective. Both perspectives need to be addressed, either as objectives or as constraints [10]. In this thesis, the operator's perspective is the goal of the optimization whereas the passengers' perspective is included as a set of boundaries.
In this thesis, a simulation implemented by [9] is expanded to be able to consider requests in static as well as dynamic time frames. Furthermore, two heuristics are applied to the expanded simulation for inserting requests and planning the buses accordingly. One heuristic is a greedy algorithm inserting the request in the first appropriate bus, while the second is a clustering algorithm that tries to bundle requests that have similar locations and time frames. The goal of this thesis is to find out, if one heuristic is better when considering the operator's perspective while also taking a look at the passengers' perspective.

# 3 Literature overview

Pratelli et al. [8] analyzed a route deviation bus problem where an existing schedule is deviated to satisfy dynamic demands, using ILP. Pei et al. [7] focused on a system with A-stops that are always visited by the bus and B-stops that are only visted when demanded with a combination of the traversal method and tabu search algorithm where the traversal method is used for a smaller number of B-stops between two A-stops and the tabu search for a greater number of B-stops between two A-stops. Inturri et al. [4] also looked at a semi-flexible dynamic system using an agent-based model combined with information from a GIS. Passengers are rejected if the distance to their pick-up stop is too long and left unsatisfied if their waiting time is too long. Regarding the deviation from the standard route, there are three choices. Either the bus decides randomly to drive on the deviated route, all buses drive on deviated routes except for a random percentage or each bus is assgined to a deviated route except for a random percentage. Their results showed that from a passengers' perspective the last strategy with no randomness resulted as the best strategy. Wang et al [11] use rules to decide if a request to an existing system should be accepted and, if accepted, then re-optimise their existing schedule.

For the static demand Pan et al. [6] proposed an heuristic algorithm derived from the gravity model.

Regarding the efficiency of dial-a-ride solutions [5] discussed creating a metric to compare different dial-a-ride systems, also including metrics to prove their permissibility in German cities as part of the public transport system. They combined deviations, used vehicle capacity and percentage of empty kilometers to a "top" metric called system efficiency. In their most simple metric, this top metric is calculated by simply dividing the sum of the distance for the direct path for all passengers by the sum of the distance effectively driven.

## 3.1 Puppes thesis

The simulation developed in this thesis is based on the existing simulation of Leo Puppe described in his bachelor's thesis "Vergleich von einer anfragebasierten und fahrplanbasierten Minibuslinie durch Simulation" [9] and the problem description above. There are some similarities between his thesis and this thesis. As described in **??** one request is associated with one person and only seating is considered, no other needs like luggage or wheelchairs.
As described in Chapter 4 a fixed number of buses operate on a line with a fixed number

and order of stops. Between arriving at a stop and leaving a stop, the bus needs to wait for the people to get on and off the bus (loading factor). The pick-up and drop-off stops are chosen arbitrarily, as long as they are not equal. All buses start at the first stop on the route at the beginning of the simulation. When skipping a stop the time needed to drive is not reduced, only the loading factor is removed. The passenger requests are made also at a random time during the simulation, in other words Puppe only considered dynamic demand. Their desired departure time is also chosen randomly (after the request is made) and the simulation has to consider a maximum waiting time from the start of the desired departure time until the pick-up time as well as a maximum travel time from the start of the desired departure time until the drop-off time.
In his simulation, he compared a conventional bus line based on existing time schedules with a greedy algorithm based on the dynamic requests with the result that on lines with low demand the greedy algorithm resulted in better outcomes than the conventional line.

# 4 Problem formulation

## 4.1 General approach of the simulation

The goal of this bachelor thesis is the examination of the optimisation for a dial-a-ride line leading to/from a common destination (referred to as a train station) combining the dynamic and static approach. This is done by implementing a simulation described in Chapter 5 and comparing two different heuristics for inserting requests and planning the buses. One heuristic is a greedy algorithm, the other is a clustering algorithm.

## 4.2 Simulation

**Bus** In this setting, there is a bus line serviced by $m$ buses leading to and from a train station. Each bus has the capacity for a number of passengers $c$. $c$ is the same for every bus.

**Line and driving times** The bus moves on a line with $n$ predefined stops, one of which is the train station at the end of the line. Traffic does not change, so the time required to drive from one stop i to the next stop j, $driving\_time_{i,j}$ is always the same, although it can be different between different stops. As we are on a line, the order of the stops is consistent, but stops can be skipped. In this case, the driving time from the previous stop i-1 skipping stop i to the next stop j is predetermined and can be shorter: $driving\_time_{i-1,j} \leq driving\_time_{i-1,i} + driving\_time_{i,j}$. The time from one stop i to a stop j is the same regardless of the direction: $driving\_time_{i,j} = driving\_time_{j,i}$.
The driving time is also subject to the parameters minimum driving time $minimum\_driving\_time$, maximum driving time $maximum\_driving\_time$ and maximum shortcut saving time $maximum\_shortcut\_saving\_time$:
$minimum\_driving\_time \leq driving\_time_{i,j} \leq maximum\_driving\_time$.
If the trip is a shortcut between station i-1,j skipping station i the following is also valid:
$minimum\_driving\_time \leq driving\_time_{i-1,j}$ and
$driving\_time_{i-1,i} + driving\_time_{i,j} - maximum\_shortcut\_saving\_time \leq driving\_time_{i-1,j} \leq driving\_time_{i-1,i} + driving\_time_{i,j}$. The actual $driving\_time$ between stops is calculated when initializing the simulation, the $driving\_time$ is random within the boundaries described above. This is done to reflect the variability of length between stops in real bus routes.

**Loading time** At each stop where passengers are picked up and/or dropped off, the bus needs to wait for them to enter or leave the bus. This is reflected in the loading time. The bus can wait longer than required, so the stopping time can exceed that minimum: $departuretime\_stop_i \geq arrivaltime\_stop_i + loading\_time$

**Static and dynamic period** Passengers can call and demand to be transported by a bus either to or from the train station. There are two time periods, in which the customer can call: The static period is until the day before the transport. The dynamic period is during the day of the transport. For dynamic transports the time between their request and the time they want to leave has to take into account the lead time: $\forall dynamic\_requests : request\_time \leq earliest\_pickup\_time + lead\_time$

**Direction of requests** Passengers can call for two different directions. Inbound means a request from the train station to another stop. Outbound means a request from a stop to the train station.

**Request parameters** Aside from the direction and the time period of the request, there are four important parameters to each request $r_k$: the earliest pick-up time $earliest\_pickup\_time_k$, the latest drop-off time $latest\_dropoff\_time_k$, the pick-up stop $pickup\_stop_k$ and the drop-off stop $dropoff\_stop_k$. The $earliest\_pickup\_time_k$ of course has to be before the $latest\_dropoff\_time_k$: $earliest\_pickup\_time_k < latest\_dropoff\_time_k$ Associated with this is the parameter of maximum travel time $maximum\_travel\_time$, that describes the maximum time a passenger is willing to wait from their earliest pick-up time to their latest drop-off time: $latest\_dropoff\_time - earliest\_pickup\_time \leq maximum\_travel\_time$. This parameter is defined in correlation to the driving time between the pick-up stop and the drop-off stop: $maximum\_travel\_time = driving\_time_{pickup\_stop_k, dropoff\_stop_k} \cdot patience\_factor$
On inbound requests, passengers specify their earliest pick-up time and their drop-off stop.
On outbound requests, passengers specify their pick-up stop. For static requests they specify their latest drop-off time and for dynamic requests their earliest pick-up date.
From this information, the simulation calculates the other two necessary parameters taking into account the direction of the request and the maximum travel time.
Passengers start waiting when their earliest pick-up time begins regardless of their request being scheduled yet. Requests are only canceled, when it is impossible to fulfill them while observing all constraints (related to the request and the buses).

**Busstate** At a time t, each bus has a list of its previous stops and associated requests, a list of its next stops it needs to service and associated requests and a list of requests it is currently transporting. At each stop the lists and its state are updated. The number of free seats, the direction the bus is currently driving in and its state (waiting/driving/at station) are calculated when needed from this information.

**Passengers** Passengers will remain on the bus that picked them up, they will not need to change vehicles. One request is also always associated with only one person.
Passengers only need space for themselves. Wheelchairs, luggage and other additional space factors are not considered.

**Operator Conditions** The vehicles operate on a line around the train station with a depot near the train station. Buses need to drive from the depot to the line at the beginning of their drivers' day and back at the end of their drivers' day. To simplify, the train station is equal to the depot and additional routes associated with driving to/from the depot without passengers are neglected.
The drivers have flexible work hours, so that the operator doesn't pay a fixed amount but based on hours driven and waiting. Driving is defined as the time the bus is not at a stop, but between stops. Waiting is the time the bus is at a stop other than the train station. When at the train station, the drivers are assumed to take a break and are therefore neither driving nor waiting.

## 4.3 Parameters

For the simulation described above, a number of parameters can be chosen: The duration of the simulation. The number of buses $m$, their capacity $c$, the *loading_time*. The number of stops $n$, the *minimum_driving_time*, the *maximum_driving_time* and the *maximum_shortcut_saving_time*. For requests, the percentage of static requests and the total number of requests can be varied as well as the *lead_time*, the *patience_factor* as well as the seed for generating random requests.

## 4.4 Heuristics

Both heuristics are explained in more detail in Chapter 5.

**Greedy Algorithm** The greedy algorithm tries to insert the request into the first bus it fits into. The buses leave their current stops as soon as possible and only wait for assigned requests.

**Clustering Algorithm** The clustering algorithm tries to insert the request into the bus where it fits the most. The best fit is determined by considering if the bus stops on the request's drop-off stop or pick-up stop anyway, if it stops close to it and by the constraints of the other requests transported by the bus.
With this algorithm the buses try to wait as long as possible at the first or last station minus buffer time to wait for possible future unknown requests. If the capacity of a bus is reached, the bus leaves as soon as possible.
The buses don't stay at their current stop when empty, but either return to the train

station or drive to the very last stop on the route.

## 4.5 Success Criteria

### 4.5.1 Carrying rate

One of the most important criteria for success is the number of requests that could be transported in comparison to the number of requests that were rejected. This is shown in the carrying rate: $carrying\_rate = \frac{\#cancelled\_requests}{\#cancelled\_requests + \#transported\_requests}$. The carrying rate is calculated for all requests as well as for static and dynamic requests separately.

### 4.5.2 Passenger Criteria

For passengers, two criteria are important. The first one is the average total passenger time. This is the time from the earliest pick-up time of a request to their actual drop-off time averaged over all requests that are dropped off. The other criterion is the average total transport time. This is the time a passenger spends on the bus including waiting and driving times, also averaged over all requests that are either currently being transported or are dropped off. Like the carrying rate, the passenger criteria are evaluated for all requests as well as for static and dynamic requests separately.

### 4.5.3 Operator Criteria

In order to assess the efficiency of the solution, there needs to be way to assess its efficiency. On the cost side, the most important and variable factors are the costs for the vehicle and for the staff. Some costs are fixed and are not considered in the calculation of the efficiency. Other costs are variable and are influenced directly by planning the routes, especially by accepting or rejecting passengers. These are the working hours for the bus drivers and fuel/electricity used by the bus as well as wear and tear related costs influenced by the distance driven. Even while waiting at a stop, the costs for the vehicle are assumed to be in effect to reflect costs for a running motor and the driver. Often the vehicle costs are calculated by distance driven and not the time, however in this thesis the costs are calculated by time. This is done to simplify calculations (staff costs are per time, not per distance) and to better reflect waiting times. Also the costs can then be better compared to the travel or booked time of the passengers (see below). As described above, when the bus is at the station, the driver is assumed to take a break. There will therefore not be any variable costs, while a bus is standing at the station.
In order to assess requests, two metrics are important which are following Liebchen et al [5] but adapted to time instead of distance: detour factor and empty running time. For the detour factor the sum of the total transport time for each passenger is divided by the sum of the booked time for each passenger. The booked time is the driving time between the request's pick-up stop and drop-off location: $detour\_factor = \frac{total\_transport\_time}{\sum_{request\ r} driving\_time_{pickup\_stop_r, dropoff\_stop_r}}$.

The running time of a bus b $running\_time_b$ is the time it spends away from the train station. $running\_time$ is this time summed over all busses.
The empty running time is the time of a bus away from the train station during which the bus carries no passengers divided by its entire running time summed over all buses: $empty\_running\_time = \sum_{bus\ b} \frac{running\_time_b: freeseats=capacity}{running\_time_b}$.

Aside from that the bus utilization is considered: $bus\_utilization = \frac{\sum_{trip:stop_i,stop_j} \frac{\#transported\_requests}{capacity} \cdot (driving\_tim}{\sum running\_time}$

# 5 Simulation

The source code of the simulation is accessible for users of the University of Würzburg at `https://gitlab2.informatik.uni-wuerzburg.de/s392111/line-based-minibus-simulation` on the branch "thesis".

## 5.1 Differences to simulation by [9]

The simulation in this line is based on an existing simulation developed by [9]. It is an incremental time progression simulation in which the state of the simulation is updated based on what is happening at that time, either a bus arriving or departing from a stop or a request that becomes known, starts waiting, can be assigned to a bus or is now impossible to fulfill.
In this thesis, the line used is a feeder line. The passengers only get to pick one stop, the other is always the train station at the end/beginning of the route. If they are able to skip a stop, not only do buses not incur loading time, but the driving time can also be reduced. Static demand is considered as well as dynamic demand while the maximum waiting time is not considered.

In order to get a working basis, refactoring of the code of the existing simulation created by [9] was done to allow the creation of additional classes that include static demand, shortcuts and different insertion strategies. This was mainly done by using the underlying structure of interfaces instead of implemented classes and applying it more thoroughly. If needed, methods from implemented classes were added to their interfaces. This allows to switch implementations easily when needed, while still keeping other implementations that remain valid from Puppe's simulation. This way, the simulation can be expanded in future works.

## 5.2 Short description of the classes in the simulation

The simulation has two main parts: the core, where all calculations take place, and visualization. Visualization is not described in detail.

**Bus** Buses have an associated route, a list of previous actions and next actions as well as a list of requests (=passengers) that are transported on the bus at the current time of the simulation. Actions change from the list of next actions to the list of previous actions when their timestamp is equal to the time in the simulation. When this happens, the

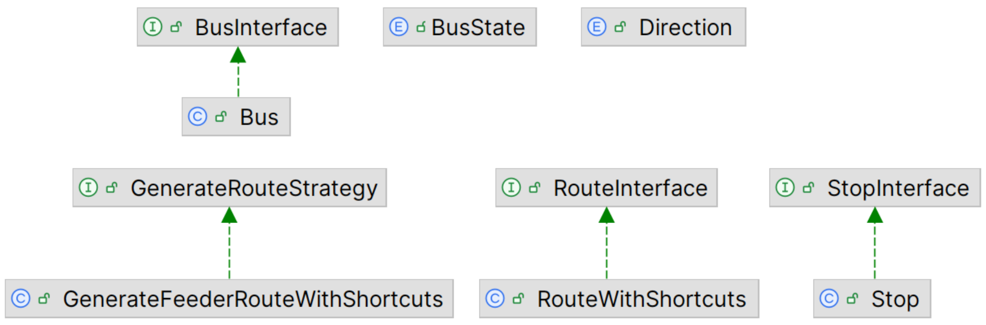

**Fig. 5.1:** UML of Interfaces and Classes for Buses, Routes and Stops including BusState, Direction and the strategy to generate new routes

list of transported requests is updated and requests are either removed, if the bus arrives at their destination stop, or added if it departs from a pick-up-stop and the request is scheduled to be transported on this bus. For Buses there is only one implementing class. A Bus also always has a BusState. It can either be "DRIVING", if its last action was the departure from a stop or be "AT_STATION" if its last action was arriving at the train station or be "WAITING" if its last action was arriving at any other stop.
A Bus has a direction that can either be inbound or outbound. This is calculated based on the corresponding time and its list of next and previous actions. If the bus is at the train station with no next actions planned, its direction is INBOUND. If the pair of (departure, arrival) has the first stop with a lower index than the index of the next stop (=closer to the train station) it is also INBOUND. In other cases, the direction is OUTBOUND.

**Stop** A Stop on a route has a name and an index. If the index is 0, its name is "train station", in all other cases the name is equal to the index. The lower the index, the closer it is to the train station.

**Route** A Route consists of a list of stops as well as a map of driving times as described in Section 4.1. It is generated at the start of the simulation by the GenerateRouteStrategy. In case of this simulation, the GenerateFeederRouteWithShortcuts is used which has the parameters number of stops, maximum driving time between two stops, minimum driving time between two stops and maximum shortcut saving time when using shortcuts. After instantiating the stops, a random driving time between the minimum driving time and the maximum driving time for neighboring stops is generated. After that the time saving by using shortcuts is calculated by taking either the time without using a shortcut reduced by a random number between 0 and the maximum shortcut saving time or the minimum driving time, if the result is lower than that. The driving time between two stops, with or without shortcuts, is the same regardless of the direction. The driving time between stops is generated during the initialization of the simulation and remains the same throughout its runtime.

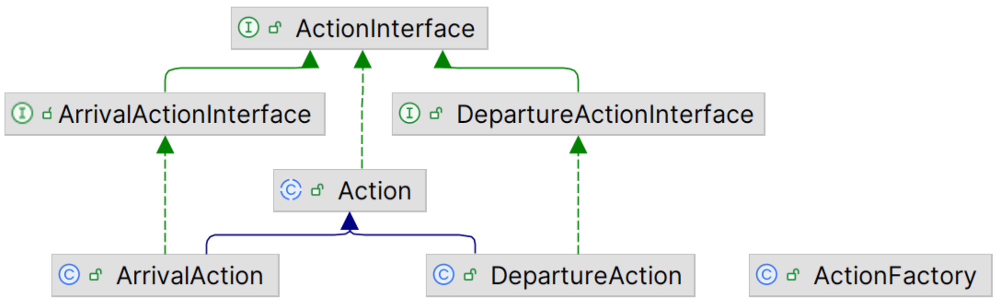

**Fig. 5.2:** UML of Interfaces and Classes forActions

**Actions** Actions can either be arrivals at (ArrivalAction/Interface) or departures from (DepartureAction/Interface) a stop at a certain time. The time can be in the future as well as the present or the past. ArrivalActions have no additional information. DepartureActions have an associated BoardRequestStrategy that contains a list of requests to be boarded at this time and stop. In this simulation the BoardRequestStrategy BoardRequests was chosen which simply sets a list of requests it receives as parameter as the list of requests to be boarded.

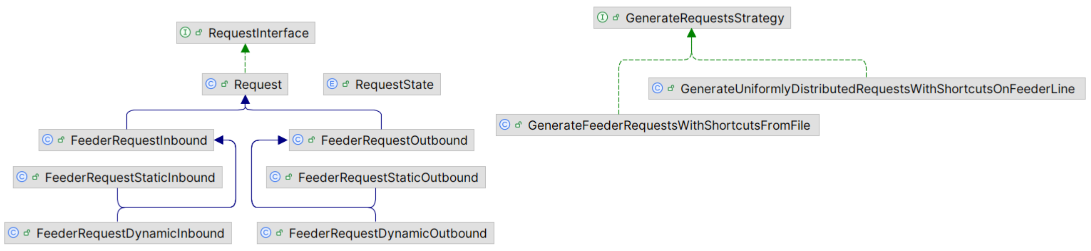
RequestInterface
Request
RequestState
FeederRequestInbound
FeederRequestOutbound
FeederRequestStaticInbound
FeederRequestStaticOutbound
FeederRequestDynamicInbound
FeederRequestDynamicOutbound
GenerateRequestsStrategy
GenerateUniformlyDistributedRequestsWithShortcutsOnFeederLine
GenerateFeederRequestsWithShortcutsFromFile

**Fig. 5.3:** UML of Interfaces and Classes for Requests and the strategy to generate requests

**Requests** In general, Requests always have a pick-up and drop-off stop, an earliest pick-up time and a latest drop-off time. They are also made at a certain time, the request time. Each Request also has a RequestState. The RequestState is one of the following:

- UNKNOWN: it is prior to the request time

- KNOWN: after the request time, but before the earliest pick-up time
- WAITING: after the earliest pick-up time, but not on a bus yet
- PROCESSING: currently being transported by a bus to the drop-off stop
- FINISHED: arrived at drop-off stop
- CANCELLED: not being transported at all

In this simulation, we differentiate furthermore between static and dynamic as well as inbound and outbound requests. Inbound requests only require a drop-off stop, a request time, an earliest pick-up time and a maximum travel time. The pick-up stop is always the train station. The latest drop-off time is calculated from their earliest pick-up time and their maximum travel time. This is the same for dynamic and static inbound requests, however for static requests the request time is always 0.
Outbound requests only require a pick-up stop, the drop-off stop is always the train station. For static outbound requests the latest drop-off time and the maximum travel time is required. The request time is always 0 and the earliest pick-up time is calculated by the other parameters. For dynamic outbound requests, the latest drop-off time is calculated by the required earliest pick-up time and maximum travel time. The request time also has to be specified.
Requests can either be read from a file or be generated based on seed with uniform distribution. The file is mostly for debugging and testing purposes and not described further.
For generating requests randomly a GenerateRequestStrategy is used. It requires the total number of requests, the percentage of static requests, the seed as well as the maximum travel time. For static and dynamic requests, approximately half are expected to be inbound and outbound using a random boolean generated from seed to determine it. For static inbound requests the earliest pick-up time is somewhere between the start of the simulation and its end minus the maximum travel time. For static outbound requests, the latest drop-off time is somewhere between the start and end of the simulation. For dynamic requests, the earliest pick-up time is sometime between the request time and the simulation duration minus the maximum travel time. The stops are chosen without constraints.

**Events** Events are mostly used for visualizing the simulation. Every time something significant happens, like the time changes, a request is delivered or a bus arrives at a stop, a new Event is generated describing what exactly happend including the corresponding objects (e.g. a BoardRequestEvent contains the simulation, the request, the stop and the bus).

**Settings** The Settings describe the constraints of the simulation. To be more precise, the Settings define the duration of the simulation, the loading factor, the number of available buses and their capacity as well as the strategy to generate new routes and

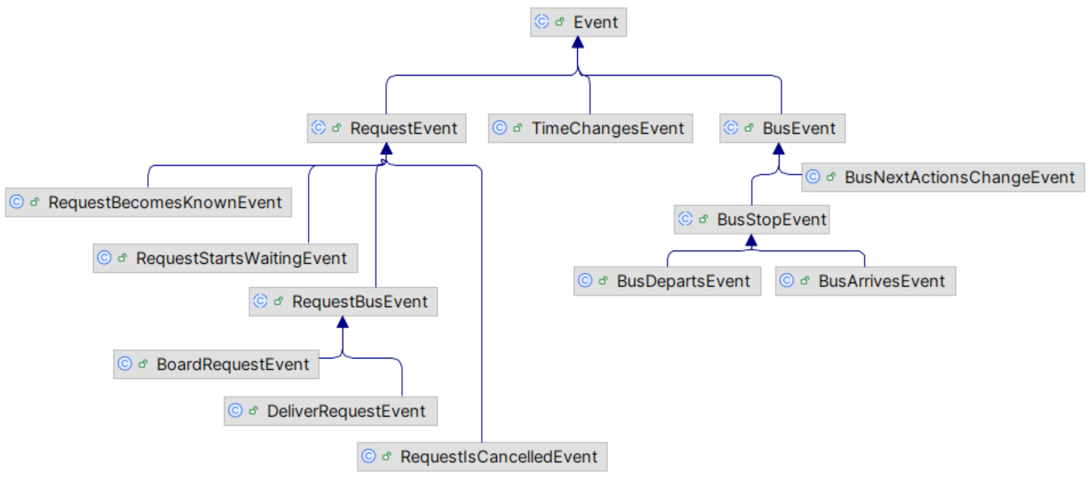

**Fig. 5.4:** UML of Interfaces and Classes for Events

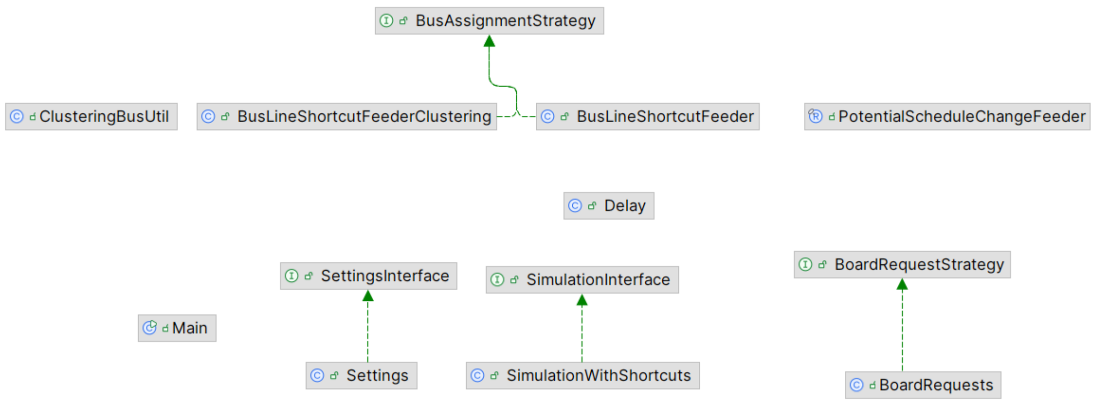

**Fig. 5.5:** UML of Interfaces and Classes for Settings, Simulation, BusAssignmentStrategy, BoardRequests

requests and the strategy to assign buses.

**BusAssignmentStrategy** The BusAssignmentStrategies are used to plan the next actions of the buses. In this simulation, we choose two different strategies, a greedy strategy used for all buses (Section 5.3) based on the greedy algorithm implemented by [9] and clustering used for static requests and greedy for dynamic and remaining requests (Section 5.4. The strategy has to adhere to some rules while planning the next actions of buses:

- departures and arrivals must be alternating

- if the list is not empty, the last action must be an arrival
- the stopping times must be observed between arrivals and departures
- the driving times must be observed between departures and arrivals
- a bus can only turn when empty
- the latest arrival times for all requests in the list must be observed

as defined by [9]

**Simulation** The simulation is the class that actually executes the simulation. It initializes the routes, buses, requests and strategies. After that the simulation updates the request states, plans buses using the BusAssignmentStrategy from the corresponding Settings and updates Buses if their action is at the current time. It also calls the visualizer, if defined. The simulation also has a method for setting the next actions of the buses called by the BusAssignmentStrategy, that checks if all constraints are considered and if the syntax of the list of next actions is correct. It additionally keeps track of a variety of information for evaluating the different BusAssignmentStrategies.

## 5.3 Static and dynamic demand combined on a feeder line with shortcuts using a greedy algorithm

In this section the approach for the greedy algorithm is described. It is based on the approach used by [9], but changed to use the advantages of the boundaries set by the feeder line.
It is tried to plan requests, meaning assigning unassigned requests at or after their earliest pick-up time to a bus and adding actions to the bus accordingly, during each tick of the simulation. However, only requests are considered that are made before or at the current time in the simulation and that are not yet assigned to a bus already. Also requests that are impossible to fulfill as their latest drop-off time is before the current time plus the time it would take to drive directly from their pick-up location to their drop-off location are not considered as well.
Generally, insertion of the request is tried for each bus. To make it more efficient, the buses are sorted and the buses near to the request's pick-up location at an appropriate time are tried first. This results in two lists: buses near the request and other buses. In themselves the two lists of buses are sorted by their order in the list of buses in the Simulation class. After iterating over all buses in this order and saving potential schedule changes in a list, the first bus with a valid schedule change will receive the request.
Regarding the departure time from the pick-up stop (and all further stops if timing was changed) the bus leaves at the earliest time possible.

### 5.3.1 Combination of static and dynamic demand

The differentiation between static and dynamic requests is done in this strategy by the order in which requests are scheduled. The static requests are scheduled first and only after that dynamic requests are added. In the simulation the time static requests become visible is at 0. It is possible for dynamic requests to also be visible at 0, but they are still scheduled after static requests.

### 5.3.2 Inbound Requests

Inbound requests are requests that take passengers from the train station to their drop-off location. The following Algorithm 1 is performed for each bus in the order described above (Section 5.3). The buses in the list that is first tried as described above are defined as buses that are at the train station during the interval of earliest pick-up time to latest drop-off time minus direct travel time. The second list includes all buses not included in the first list. The first bus to return a non-empty list of next actions with correct syntax and considering all constraints such as latest drop-off time of all their requests is awarded the new request.
As parameters, the algorithm takes the request with its drop-off stop, earliest pick-up time and latest drop-off time as well as the bus with its current transported requests, its capacity, its list of previous actions and the list of next actions in the future that are planned for the bus.
Stop a is smaller than Stop b if its index is smaller.
During the following algorithm, it is always tested, if the boundaries of time and bus capacity can be observed. If not the function returns with an empty list. First it is tested in line 2, if the bus has any next actions. If not, the request is inserted directly in line 3and the function returns. If there are new actions, all actions before the earliest pick-up time or before the pick-up stop are taken unchanged in line 4-5. This logic is taken from [9]'s work. If after that, there are no more next actions, the request is again inserted directly and the function returns in line 9-11. If not, it is tested if the bus has its last actio before it turns and before the request's drop-off stop in line 12-18. If yes, the drop-off stop is added to the bus' route and the function returns. This is also taken from [9], described in the function integrateRequestWhileDriving of the class OnDemandBusLinePuppe.
If the bus stops at the request's drop-off stop anyway, the request is also inserted and the function returns as described in line 19-24. If the function hasn't returned yet, it means that the bus either drives past the request's drop-off stop or it turns before reaching the drop-off stop. In this case in lines 25-29, it is tried to insert the drop-off stop with the additional delay. If the constraints of all requests assigned to the bus can still be observed and the bus has free seats, the request is added and the function returns. If not, the first next action is transferred to the list of new actions in line 30 and the algorithm starts from the top in line 1/2 until either the request can be inserted or it is impossible to fulfill the time boundaries and the function returns with an empty list.

```
Algorithm 1: PlanInboundRequestForOneBusForGreedyAlgorithm(Request r, Bus b)
   Output: List of next actions for the bus
1  while True do
2      if b.listNextActions.empty() then
3          return new List(arrive at r.pickupStop, depart from r.pickupStop
             boarding r, arrive at r.dropoffStop)
4      listOfNewNextActions = new List()
5      listOfNewNextActions.add(b.listNextActions.transferWhile(action.time <
         earliestPickupTime || action.stop != r.pickupStop))
6      if r.latestDropoffTime < b.listNextActions.first.time +
         drivingTime(r.pickupStop, r.dropoffStop) then
7          // the request can't be integrated with this bus using this algorithm
8          return List.empty();
9      if b.listNextActions.empty() then
10         listOfNewNextActions.add(new List(arrive at r.pickupStop, depart from
             r.pickupStop boarding r, arrive at r.dropoffStop))
11         return listOfNewNextActions
12     // the bus doesn't turn before its last action and its last action is before the
         new dropoff stop
13     if b.listNextActions.getLastActionBeforeTurn.stop < r.dropoffStop and
         b.listNextActions.ActionsBeforeTurn.size == b.listNextActions.size then
14         add request to departure from trainstation in listNextActions
15         listOfNewNextActions.add(b.listNextActions).add(arrival at
             r.dropOffStop)
16         if arrival at r.dropoffStop.getTime() > r.latestDropoffTime ||
             !b.hasFreeSeats then
17             return List.empty()
18         return listOfNewNewActions
19     if b.listNextActions.ActionsBeforeTurn() includes action with the stop
         r.dropOffStop then
20         add request to departure from trainstation in listNextActions
21         listOfNewNextActions.add(listNextActions)
22         if arrival at r.DropoffStop.time > r.latestDropoffTime || !b.hasFreeSeats
             then
23             return List.empty()
24         return listOfNewNewActions
25     // at this point, the bus has to drive past the newDropoffStop or turn before
         reaching the newDropoffStop if b.hasFreeSeats then
26         delay = added time by inserting r.DropoffStop between neighbors or
             before turn if for all existing requests and r, latestDropOffTime is
             observable then
27             add request to departure from trainstation in b.listNextActions
28             listOfNewNextActions.add(b.listNextActions.transferWhile(action.stop
                 < neighbor || action.getStopBeforeTurn).add(r.dropOffStop)
29             listOfNewNextActions.add(b.listNextActions.map(add delay to
                 actions)) return listOfNewNextActions
30     listOfNewNextActions.add(b.listNextActions.remove(0))
```

### 5.3.3 Outbound Requsts

Outbound requests are requests that take passengers from their pick-up location to the train stop. Like for inbound requests, the following Algorithm 2 is tried for each bus. Near buses in this case are buses that are at the pick-up stop or further away from the train station than the pick-up stop in the interval of earliest pick-up time and latest drop-off time minus direct driving time. Also again, the first bus to return a non-empty list of next actions with correct syntax and observing all constraints is awarded the request.
During the following algorithm, it is always tested, if the boundaries of time and bus capacity can be observed. If not the function returns with an empty list. First, it is tested in line 2 if the bus has next actions. If not, the request is inserted directly and the function returns. After that, all actions before the earliest pick-up time, before the pick-up stop or in the wrong direction are accepted unchanged in line 5. If after that, the bus only has one or fewer actions, the request is also directly inserted after the last action and the function returns in line 8-11. Until here, this is the same logic as applied in [9]. After that, all actions that are further away from the train station than the pick-up stop are accepted unchanged in line 12. If the next stop is the request's pick-up stop, the request is inserted and the function returns in line 13-16. If not, it is tried to insert a detour to the request's pick-up stop. If the latest drop-off times of all requests assigned to the bus can be observed, the request is assigned to the bus and the appropriate actions are added to the list, the function returns in line 17-22. If not, the bus drives past the stop and all outbound actions are accepted unchanged in line 23. After that the algorithm starts from top in line 2, either until the request can be inserted or it is impossible and the function returns with an empty list.

## 5.4 Static and dynamic demand combined on a feeder line with shortcuts using a clustering algorithm for static demand and greedy algorithm for dynamic demand

Like in the greedy algorithm, buses are divided into buses that are near the request and other buses. Different to it, the buses are sorted more thoroughly (described for inbound and outbound requests separately below) taking into account information from the earliest pick-up time until latest drop-off time minus driving time. Buses that are considered near the request keep their original order. Other buses are sorted by the difference of their index to the desired stop (drop-off stop for inbound requests, pick-up stop for outbound requests). This way buses that drive closer to the desired stop are handled first. Inserting the request with the algorithms presented below is tried in the described order until successful.
Regarding the departure time of buses from the stop, until all seats are filled, the bus departs at the latest possible time minus a buffer time. The latest possible time is the time when the latest drop-off time for all requests assigned to the bus can still be observed. When all seats are filled, the bus leaves at the earliest possible time.

**Algorithm 2:** PlanOutboundRequestForOneBusForGreedyAlgorithm(Request r, Bus b)

```
Output: List of next actions for the bus
while True do
    if b.listNextActions.empty() then
        return new List(departure current stop, arrival at r.pickupStop,
          departure r.pickupStop, arrival at r.dropoffStop)
    listOfNewNextActions = new List()
    listOfNewNextActions.add((b.listNextActions.transferWhile(action.time <
      earliestPickupTime or action.stop != r.pickupStop or b.Direction ==
      INBOUND))
    if b.listNextActions.get(0).time > r.latestDropOffTime -
      drivingTime(r.pickupStop, r.dropoffStop) then
        return List.empty()
    if b.listNextActions.size <= 1 then
        listOfNewNextActions.add(b.listNextActions)
        listOfNewNextActions.add(departure last stop, arrival at r.pickupStop,
          departure r.pickupStop, arrival at r.dropoffStop)
        return listOfNewNextACtions
    listOfNewNextActions.add(b.listNextActions.transferWhile(action.stop >
      r.pickupStop))
    if b.listNextActions.get(0).stop == r.pickupStop and b.hasFreeSeats then
        insert request at departure from r.pickupStop
        listOfNewNextActions.add(b.listNextActions)
        return listOfNewNextActions
    if b.hasFreeSeats then
        delay = added time by inserting r.DropoffStop between neighbors or
          before turn
        if for all existing requests and r, latestDropOffTime is observable then
            add r.pickupStop between neighboring stops in b.listNextActions
            listOfNewNextActions.add(b.listNextActions.map(add delay to
              actions))
            return listOfNewNextActions
    listOfNewNextActions.add(b.listNextActions.transferWhile(action.direction
      == OUTBOUND))
```

If the last planned action of a bus is the arrival at a stop that is not the train station, it is either returned to the train station if the stop is in the first 75% of stops or to the last stop on the route if it is in the last 25%.

### 5.4.1 Combination of static and dynamic demand

The simulation tries to plan the static demands, first inbound, then outbound, with clustering during initialization before the first tick. Successfully planned requests are added to a list of accepted requests that is handled by the assignment strategy. After that, all unassigned static requests and all dynamic demand is handled the same way as described in Section 5.3.

### 5.4.2 Inbound Requests

For inbound requests, buses are considered near the request if they have free seats, are inbound at the time and stop at the drop-off stop after visiting the train station. For the other buses only buses that have free seats, are inbound and stop at the train station at the time are considered. All other buses are not taken into account. The time for which the conditions above are true is saved and given to the algorithm as a parameter. Algorithm 3 takes the request, the bus and this time as a parameter and tries to insert the request for the given bus. If successful and the drop-off stop is the last stop on the route, the bus is either returned to the train station or to the last stop. If the bus is returned to the train station, the requests waiting there are re-ordered if necessary using Algorithm 4.
If the bus stops at the request's drop-off stop anyway, the request is inserted directly and the function returns in lines 2-5. Balancing the requests in this case is not necessary.
If the bus has no further actions the request is inserted, the bus departing at the latest possible time minus the buffer, as well as the trip to either the last stop or the train station in lines 6-13. If the bus is returned to the train station Algorithm 4 is called.
If that is not possible, the request is assigned to the bus anyway and all actions before the request's drop-off stop are accepted unchanged in lines 14-16. Then the drop-off stop is inserted and the following actions are delayed accordingly in line 20.
There are less cases to consider, because the buses are sorted and selected more thoroughly.

For balancing the buses at the train station, the bus with the highest number of different drop-off stops is chosen in line 1 of Algorithm 4. If no bus has requests, with more than one different drop-off stop, there is no need for balancing and the function returns in line 2-3.
The requests of the other bus are then sorted by the index of their drop-off stop beginning at the stop neighboring the train station in line 4. The requests are then transferred to the newly arriving bus in 6-8 until the list of requests for the new bus is bigger than the list of requests for the other bus.
The actions of both buses are adjusted accordingly in their timing as well as in them

happening at all in line 9. After that the new bus is returned either to the train station or to the last stop on the route in lines 10-15. If necessary, the balancing happens again for the new arrival at the train station.

**Algorithm 3:** PlanInboundRequestForOneBusFoClusteringAlgorithm(Request r, Bus b, time t)

```
Output: List of next actions for the bus
1  listNewNextActions = List.empty()
2  if b has request with r.dropoffStop as dropoffStop at t then
3      listNewNextActions.add(b.listNextActions)
4      insert r into list of requests for action at time t in listNewNextActions
5      return listNewNextActions
6  if b.listNextActions.isEmpty then
7      add trip from r.pickupStop to r.dropoffStop to listNewNextActions
8      if r.dropoffStop.index < 3 /4 · route.stops.size then
9          add trip to trainstation to listNewNextActions
10         BalanceBusesAtTrainStation(b, b.getLastAction.time)
11     else
12         add trip to last stop on route to listNewNextActions
13     return listNewNextActions
14 listNewNextActions.add(b.nextActions.transferWhile(action.time < time))
15 add request to departure from trainstation
16 listNewNextActions.add(b.nextActions.transferWhile(action.stop <
     r.dropoffStop))
17 listNewNextActions.add(insert r.dropoffStop)
18 if arrival at r.dropOffStop.time > r.latestDropOffTime then
19     return List.empty()
20 listNewNextActions.add(b.nextActions.map(add delay))
21 if latestArrivalTime for existing requests not observed then
22     return List.empty()
23 return listNewNextActions
```

### 5.4.3 Outbound Requests

For outbound requests, buses are considered near the request if they are outbound, have free seats and are at the pick-up stop. For the other buses only buses that are further away from the train station than the pick-up stop and have free seating are considered. All other buses are not taken into account.
If the bus has no next actions, the request is inserted directly and the buses at the train station are balanced as described above in line 2-5.
If the bus is in the list of nearestBuses, then the bus is at the pick-up stop and the

**Algorithm 4:** BalanceBusesAtTrainStation(Bus b, time t)

```
1  otherBus = bus that is at train station at t, has requests with more than one
     drop-off stop; if more than one is present, take bus that has drop-off stops
     furthest away from train station
2  if there is no other bus then
3      return
4  requestListOtherBus = otherBus.requestsAtTime sorted by drop-off stop
5  requestListNewBus = List.empty()
6  while requestListNewBus.size < requestListOtherBus.size do
7      stop = requestListOtherBus.get(0).getStop
8      requestListNewBus.add(requestListOtherBus.transfer(dropoffStop == stop))
9  adjust actions and their timing for both buses according to new/removed
     requests
10 if b.lastStop.index < 3 /4 · buses.size then
11     add return trip to train station for b
12     if b.listNextActions.last.time < simulationDuration then
13         BalanceBusesAtTrainStation(b, b.listNextActions.last.time)
14 else
15     add trip to last station on route for b
```

request can be inserted directly, the function returns in line 6-9.
After that all actions before the parameter time, are accepted unchanged in line 11. If there are no next actions after that, the request is inserted directly, the buses at the train station are balanced and the function returns at lines 11-14.
If that is not the case, all actions before the pick-up stop are accepted unchanged in line 15. Then a detour to the pick-up stop is added and the following actions are accepted with the corresponding delay, the function returns in lines 16-20.

## 5.5 Detailed list of origin of classes and methods

If nothing saying otherwise is listed below, the class and its methods are directly taken from [9].

### 5.5.1 package "core"

**Settings** This class was largely implemented by [9]. In this thesis getGenerateRequestsStrategy() as well as the doc-comment for the class was added.

**Simulation** The logic in this class was largely implemented by [9]. In this thesis, classes were replaced by their respective interfaces for Request/RequestInterface, Bus/BusInterface,

**Algorithm 5:** PlanOutboundRequestForOneBusFoClusteringAlgorithm(Request r, Bus b, time t)

```
Output: List of next actions for the bus
listNewNextActions = List.empty()
if b is in nearestBuses and b.listNextActions.isEmpty then
    listNewNextActions.add(trip from r.pickupStop to r.dropoffStop)
    BalanceBusesAtTrainStation(b, b.listNextActions.last.time)
    return listNewNextActions
if b is in nearestBuses then
    insert r into list of requests of b.listNextActions at t
    listNewNextActions.add(b.listNextActions)
    return listNewNextActions
listNewNextActions.add(b.listNextActions.transferWhile(action.time < time))
if b.listNextActions.isEmpty() then
    listNewNextActions.add(trip from r.pickupStop to r.dropoffStop)
    BalanceBusesAtTrainStation(b, listNewNextActions.last.time)
    return listNewNextActions
listNewNextActions.add(b.listNextActions.transferWhile(action.stop >
  r.pickupStop))
listNewNextActions.add(detour to r.pickupStop)
listNewNextActions.add(b.listNextActions.map(add delay))
if !(latestDropoffTime observed for all requests) then
    return List.empty()
return listNewNextActions
```

Action/ActionInterface, Stop/StopInterface and the doc-comment for the class was added. Also the signature of setNextActions, checkNextActionsSyntax, checkForbiddenTurns, checkCurrentlyTransportedRequests and checkDrivingAndSstoppingTimes was changed from return type void to return type boolean without changing the way it works as well as renaming of some methods for clarity. Additionally, the methods getCarryingRateForStatic, getCarryingRateForDynamic, getAverageStaticRequestTransportTime, getAverageDynamicRequestTransportTime, getCalculationTime, getAverageTotalPassengertime, getAverageTotalPassengerTimeForStaticDemand, getAverageTotalPassengerTimeDynamicDemand, detourFactor, detourFactorStatic, detourFactorDynamic, emptyRunningTime, getProcessedFinishedStaticRequests, getProcessedFinishedDynamicRequests were added with the body of throwing an UnsupportedOperationException. The class was not used in this thesis, but kept for future uses and had therefore to be adapted to new requirements.

**SimulationWithShortcuts** The following description is when compared to the Simulation class implemented by [9] changed last with the commit 011d03a9c77e48f24e44f7614d87157310c0ff9b. As described in Simulation above, classes were changed to their respective interfaces. All lists are changed from LinkedList to ArrayList due to performance. Methods that are not mentioned, are not implemented.
The initializeSimulation method is deleted and its content transferred to the constructor. After sorting the requests comparing the request time like in Simulation, they are also sorted by their earliest pick-up time. when initializing the BusAssignmentStrategy, nextActionsChangeable is set to true, so that the static requests can be planned when using the clustering algorithm. For the execute method, a timer is added to get the calculation time and the planBuses method is extracted directly into the execute method instead of an independent method. For updateBusesAndREquests/executeBusNextAction the checkEmptyTurn method is deleted, because it is checked prior to that already. For updateCancelledRequests the check for maximum passenger waiting time is deleted and replaced by a check if the latest drop-off time is still later than the current time plus the driving time between the pick-up stop of the request and the drop-off stop. The deliverRequests method was refactored for readability while keeping the logic the same, only the order of events was switched so that the BusArrivesEvent is added before the DeliverRequestEvents. The boardRequests method was refactored to remove most checks and instead simply board the requests of the relevant DepartureActionInterface due to them being made prior to this method. Also the BusDepartsEvent is added before the BoardRequestEvent. The setNextActions method has a new return type boolean instead of void. The check if the BusInterface parameter is class Bus is deleted like the casting of the ActionInterfaces to Actions. If the checks are passed, the list of ActionInterfaces given as a parameter replaces the nextActions list of the bus and the method returns true. Otherwise it returns false. The checkNextActionsSyntax/isNextActionsSyntaxCorrect method was refactored to return false instead of throwing errors, if the syntax is not correct. The rest of its logic is maintained, but was refactored for readability. The methods checkDrivingAndStoppingTimes, check-

ForbiddenTurns and checkCurrentlyTransportedRequests were merged into one method areConstraintsKept. They were also refactored to return a boolean instead of throwing exceptions, while the rest of the logic was kept. For checkDrivingAndStoppingTimes the check for a trip to itself is removed as it is checked in isNextActionsSyntaxCorrect. For checkForbiddenTurns the check to prevent a too short driving time to first destination was removed, as this is checked in checkDrivingAndStoppingTimes anyway and it is irrelevant to the method. For checkCurrentlyTransportedRequests the check if the bus is empty before turning is removed, as this is done before. The methods updateKnownRequests and updateWaitingRequests are kept as implemented by [9]. The getters getAllRequests, getAllVisibleRequests, getBuses, getRoute, getSettings and getTime are kept unchanged from [9]. The method getAverageWaitingTime is refactored for readability and consolidated into two methods, but the logic is kept from [9]. getAverageRequestDrivingTime, getTotalBusDrivingTime is refactored and consolidated into one method each for readability while keeping the logic from [9] .getAverageRequestTransportTime and getTotalUtilization is also refactored and consolidated into one method for readability, this time also changing the way it is calculated. getCarryingRate and getUtilizedBuses are transferred unchanged from [9]. getAverageStaticRequestTransportTime and getAverageDynamicRequestTransportTime are added in this thesis. getCarryingRateForStatic and getCarryingRateForDynamic are added in this thesis, but get their logic heavily from getCarryingRate by [9]. getAverageTotalPassengerTime, getAverageTotalPassengerTimeForStaticDemand, getAverageTottalPassengerTimeDynamicDemand, detourFactor, detourFactorStatic, detourFactorDynamic, emptyRunningTime, getProcessedFinishedStaticRequests, getProcessedFinsihedDynamicRequests and getCalculationTime are added in this thesis.

**package defaultstrategies**

**package boardrequests**

**BoardNoRequest** getRequestsToBoard is renamed to getNewRequestsToBoardForBus, method getPlannedRequestsToBoard is added returning an empty list as well as the doc-comment. This class is not used in this thesis, but kept for future use.

**BoardUntil** getRequestsToBoard is renamed to getNewRequestsToBoardForBus, LinkedList is changed to ArrayList for performance reasons and getPlannedRequestsToBoard is added throwing an UnsupportedOperationException. Also added doc comments and made the field maxCapacity final. This class is not used in this thesis, but kept for future use.

**BoardRequests** Replaced LinkedList with ArrayList for performance reasons, renamed getRequestsToBoard to getNewRequestsToBoardForBus and changed the parameter List<? extends RequestInterface> to List<RequestInterface>. Also add doc-comments and make the field plannedRequestsToBoard final. This class is used when planning buses in this thesis, the rest of the class is written by [9].

**package generaterequests**

**GenerateRequests, GenerateRequestsWithImportantStop** The only thing changed in these classes is that the classes Route, Request and Stop are changed to their interfaces, LinkedLists changed to ArrayLists and doc-comments to the class are added. The rest is unchanged from [9]. The classes are not used in this thesis, but kept for future use.

**GenerateUniformlyDistributedRequests** Like in the classes before, Request, Route and Stop are changed to their interfaces as well as LinkedList to ArrayList, doc-comments are added. The method generateRequestsAsRequest is added. The class is not used in this thesis, but kept for future use.

**GenerateFeederRequestsWithShortcutsFromFile** This class is added in this thesis, its general structure is taken from GenerateRequests by [9], but the constructor is newly created and the parseJson method is also rewritten completely to account for static and dynamic demand.

**GenerateUniformlyDistributedRequestsWithShortcutsOnFeederLine** This class is added in this thesis, its general structure is taken from GenerateUniformlyDistributedRequests by [9], but the constructor and generateRequests is heavily modified to account for static and dynamic demand in both directions.

**package generateroute**

**GenerateSimpleRoute** Like in the classes before, Stop is changed to StopInterface as well as LinkedList to ArrayList, doc-comments are added. The class is not used in this thesis, but kept for future use.

**GenerateFeederRouteWithShortcuts** This class is added in this thesis.

**package enums**

**BusState** "STOPPING" is renamed to "WAITING", the state "AT_STATION" is added, the rest is as written by [9].

**Direction** UP is renamed to INBOUND and DOWN is renamed to OUTBOUND. deltaIndex and associated functions are deleted as well as change because it is not used. Doc-comments are added.

**RequestState** A long description is added to the doc-comments, visible and mustBe-Planned are made final, the rest is as written by [9].

**events**

**BoardRequestEvent, BusArrivesEvent, BusDepartsEvent, BusDepartsEvent, BusEvent, BusNextActionsChangeEvent, BusStopEvent, DeliverRequestEvent, Event, RequestBecomesKnownEvent, RequestBusEvent, RequestEvent, RequestIsCancelledEvent, RequestStartsWaitingEvent, TimeChangesEvent** For all events, classes like Bus, Request, Simulation, Stop, Action are changed to their corresponding interfaces and doc-comments are added to the class. toString methods are added to BoardRequestEvent, DeliverRequestEvent and RequestIsCancelledEvent. The rest ist as written by [9].

**package factories**

**ActionFactory** Like in other classes Stop was replaced by StopInterface and doc-comments are added to the class. Also checks if something was an instanceof Stop are removed. The rest is as written by [9]

**package interfaces**

**package <u>actions</u>**

**ActionInterface** Doc-comments are added to the interface itself as well as its methods. The methods setRequestsProcessedByThisAction and getRequestsProcessedByThisAction are added. getTime and getStop are by [9].

**ArrivalActionInterface** Only doc-comments are added to this interface, the rest is as written by [9].

**DepartureActionInterface** Only doc-comments are added to this interface as well as its method getBoardRequestStrategy. The rest is as written by [9].

**package <u>objects</u>**

**BusInterface** The Directions and BusStates are renamed according to their enum. The methods transferNextActionToPreviousAction, getIndex, getWriteableCurrentlyTransportedRequests and setNextActions are added including their doc-comments and a doc-comment for the interface. The List<? extends SomeInterface> are replaced by List<SomeInterface> in the method signatures. The rest is as written by [9]

**RequestInterface** The methods cancel and nextRequestState are added with their doc-comment as well as a doc-comment for the interface itself. The rest is as written by [9].

**RouteInterface** The Directions are renamed according to their enum, List<? extends StopInterface> is replaced with List<StopInterface>, getFirstStop and getDirectDrivingTime are added as well as doc-comments to the interface. The rest is as written by [9]

**SettingsInterface** Only the doc-comments are chnged slightly, the rest is as written by [9].

**SimulationInterface** Doc-comments are added to methods and interfaces, existing doc-comments are clarified. List<? extends SomeInterface> is replaced by List<SomeInterface>. The methods getAllRequests, execute, getAverageWaitingTime, getAverageRequestDrivingTime, getAverageRequestTransportTime, getCarryingRate, getTotalUtilization, getTotalBusDrivingTime, executeBusNextAction, getCarryingRateForStatic, getCarryingRateForDynamic, getAverageStaticRequestTransportTime, getAverageDynamicRequestTransporttime, getCalculationTime, getAverageTotalpassengerTime, getAverageTotalPassengerTimeForStaticDemand, getAverageTotalPassengerTimeDynamicDemand, detourFactor, detourFActorStatic, detourFActorDynamic, emptyRunningTime, getProcessedFinishedStaticRequests, getProcessedFinishedDynamicRequests and getUtilizedBusses are added as well as their doc-comments. checkDrivingAndStoppingTimes, checkForbiddenTurns are deleted. checknextActionsSyntax is renamed to isNextActionsSyntaxCorrect, its return type changed from void to boolean. checkCurrentlyTransportedRequests is renamed to areConstraintsKept, its return type changed from void to boolean and its doc-comment expanded to describe what is checked. The return type of setNextActions is changed from void to boolean. The rest is as written by [9].

**StopInterface** An index is added to the stop as well as doc-comments. The rest is as written by [9].

**package strategies**

**BoardRequestStrategy** getRequestsToBoard is renamed to getNewRequestsToBoardForBus. The method getPlannedRequestsToBoard is added as well as doc-comments for the interface and its methods. The rest is as written by [9].

**BusAssignmentStrategy** Only author-doc comments are added to this interface, the rest is as written by [9].

**GenerateRequestsStrategy** In this interface, Request, Route are replaced by RequestInterface and RouteInterface and doc-comments added to the interface and its methods. The rest is as written by [9].

**GenerateRouteStrategy** In this interface, Route is replaced by RouteInterface and doc-comments are added to the interface and its methods. The rest is as written by [9].

**VisualizeStrategy** In this interface, Simulation is replaced by SimulationInterface and author-doc comments are added. The rest is as written by [9].

#### package objects

**Action** In this class, Stop, Request are changed to their corresponding interface and author doc-comments are added. Also each action gets an unique id and the hashCode/equals methods are overwritten using only the id as relevant field. This is to ensure the correctness of actions when used in Sets. The rest is as written by [9].

**ArrivalAction** In this class, Stop is changed to StopInterface and author doc-comments are added. The rest is as written by [9].

**Bus** In this class, the naming of Direction and BusState are changed according to the enums. Also Action, Request, Route, Stop are replaced by their corresponding interface and LinkedLists are replaced by ArrayLists. Like in Action, each Bus gets a unique index at instantiation and the hashCode/equals methods are overwritten using only the index as the relevant field. The methods getBusState and getTravelDirection are changed to account for the BusState AT_STATION. The methods getWriteableNextActions and getWriteablePreviousActions are deleted and replaced by transferNextActionToPreviousActions to enforce cohesion and loose coupling. In the method getBusState comparing Stops with == is replaced by comparing with the .equals method. The rest is as written by [9].

**DepartureAction** In this class, only author doc-comments are added and Stop is replaced by StopInterface. The rest is as written by [9].

**FeederRequestDynamicInbound, FeederRequestDynamicOutbound, FeederRequestInbound, FeederRequestOutbound, FeederRequestStaticInbound, FeederRequestStaticOutbound** These classes are added in this thesis.

**Request** In this class, Stop is replaced by StopInterface and author doc-comments are added. The method setRequestState is deleted because only nextRequestState and cancel are used to set the RequestState. Like in Action and Bus, each Request gets a unique id and its equals/hashCode method are overwritten only based on this id. The rest is as written by [9].

**Route** In this class, the naming of Direction is changed according to the enum, Stop is replaced by StopInterface and LinkedLists with ArrayLists. The methods withDirectionIndependentDrivingTimes, getLastStop and getDetailedDrivingTimes are deleted because they are not used. The method getDrivingTimeVector is refactored to remove the necessity of being an instance of Stop and not just a StopInterface, while keeping the basic logic of the method as written by [9]. The method getDirection and getStops-BeforeTurn are refactored to remove the necessity of being an instance of Stop while keeping the basic logic of the method as written by [9]. The rest of the class is as written by [9]. This class is not used in this thesis, but kept for future use.

**RouteWithShortcuts** This class is largely based on Route as written by [9]. The check for correctness of the route in the constructor and its corresponding method are removed. The method getDrivingTimeVector, getStopsBeforeTurn and getDirection are refactored completely including the logic and their helper methods are removed, because it can be simplified. The rest of the class is as written by [9] in Route.

**Stop** An index with getter is added to the stop and a singleton TrainStation with getter. The rest is as written by [9]

### 5.5.2 Package "strategies.busassignment"

**AdditionalEffort** Only author doc-comments are added to this class, the rest is as written by [9]. This class is not used in this thesis, but kept for future use.

**BusLineShortcutFeeder** This class is largely based on OnDemandBusLinePuppe with some adjustments.
The list of assignedRequests and rejectedRequests is changed to a Set. The lists are only added to and searched, a Set does the same thing with a better performance.
A Map with buses as key and ClusteringBusUtil as value is added in order to find appropriate buses faster and more easily. It is initialized in initializeStrategy. The rest of the method is as written by [9].
The method assignBusToRequest is extracted into planBuses directly. The method to distinguish between nearest buses and other buses is rewritten completely, including logic changes making use of the added map described above and boundaries in place by being a feeder line.
As in OnDemandBusLinePuppe the simulation first tries to insert the request into all buses, first the nearest buses and then the other buses, and collects a list of PotentialScheduleChangeFeeder (PotentialScheduleChange in OnDemandBusLinePuppe) without actually inserting the request. After that the list of PotentialScheduleChange-Feeders is iterated until the request is inserted successfully, if possible. OnDemand-BusLinePuppe handled this with exceptions, while BusLineShortcutFeeder handles it with methods returning a boolean. In BusLineShortcutFeeder, the map with information regarding the buses is updated after a successful insertion. Different to OnDemand-BusLinePuppe, requests are only added to rejected requests when it is impossible to fulfill them while observing the latest drop-off time.
When inserting a request on a nearest bus, each action before the earliest pick-up time is left unchanged like in OnDemandBusLinePuppe. On inbound requests, the actions before the pick-up stop are also skipped like in OnDemandBusLinePuppe. For outbound

requests, actions in the other direction are skipped instead. After that the logic in BusLineShortcutFeeder deviates from OnDemandBusLinePuppe using the boundaries set by the feeder line. The same is true when inserting a request on other buses: each action before the earliest pick-up time is left unchagned like in OnDemandBusLinePuppe. After that the logic deviates using boundaries set by the feeder line.
The method checkPlannedRequests is transferred from OnDemandBusLinePuppe as written by [9] with only a few checks removed.
The checks busCapacityObservable and isLatestArrivalTimesObservedForExistingRequests are written newly in this thesis. The helper methods getIndexAfterTurn, getBusDirectionAtTime are also written newly in this thesis. Methods for inserting new actions are refactored but largely keep their logic as written by [9].

**BusLineShortcutFeederClustering** This class is written newly in this thesis.

**ClusteringUtil** This class is added completely in this thesis.

**ConventionalBusLine** Only author doc-comments are added to this class and List<? extends StopInterface> / ListIterator<? extends StopInterface> are replaced by List<StopInterface>/ListI
The rest is as written by [9]. This class is not used in this thesis, but kept for future use.

**Delay** Only author doc-comments are added to this class, the rest is as written by [9].

**OnDemandBusLinePuppe** The name of the class itself is changed from OnDemandBusLine3 to OnDemandBusLinePuppe. The naming of Direction is changed according to the enum, author doc-comments are added and LinkedLists are replaced by ArrayLists. Also like in the classes above, Request, Bus are replaced by their corresponding interfaces. There is some refactoring for readability while keeping the logic. this class is not used in this thesis, but kept for future use and for reference as large parts of BusLineShortcutFeeder are based on this class. The rest is as written by [9].

**PotentialScheduleChange** Only author doc-comments are added to this class, the rest is as written by [9].

**PotenitalScheduleChangeFeeder** This is the same as PotentialScheduleChange, but without the requestDelay.

### 5.5.3 Package "visualization"

**console/ConsoleVisualizer** Only author doc-comments are added into this class and Simulation is replaced by SimulationInterface. The rest is as written by [9]. This class is not used in this thesis, but kept for future use.

**gui/Connectionpane, gui/SimulationPane, gui/StopPane** Only author doc-comments are added into these classes. The rest is as written by [9]. They are not used in this thesis, but kept for future use.

**gui/DiagramAdapter** Only author doc-comments are added into this class, wayTime-DiagramPane is made final and Simulation is replaced by SimulationInterface. The rest is as written by [9].

**gui/WayTimeDiagramPane** Doc-comments are added to this class and Simulation, ArrivalAction and DepartureAction are replaced by their corresponding interface. Some code is refactored without changing the logic. The rest is as written by [9].

### 5.5.4 class "Main"

Old comments are removed, classes are called by their interfaces where possible and the old classes are replaced by the new ones for this thesis (e.g. GenerateFeederRoute-WithShortcuts instead of GenerateSimpleRoute).
For the comparison of results new criteria are added and the appendNewLine is changed to work better with Excel when separating data into columns. The method getStrat-egyAndRequestCount is added as well as getLongAverage. The rest is as written by [9].

# 6 Evaluation

## 6.1 Parameters for the simulation

As described in [9] the duration of one tick is a tenth of a minute or in other words 6 seconds. The simulation lasts for 1.000 ticks, so about 100 minutes.

**Buses and route** The simulation uses a route with 10 stops. Each bus can carry 4 passengers at a time.
Regarding the loading factor [1] showed for several larger bus models with four or five doors that for one to 4 persons getting in or out of the bus, the bus needed to stop from about 1 second to about 4 seconds. As our smallest time unit is one tick, the loading factor is one tick / 6 seconds.
In [3] the maximum recommended walking distance to a stop is given with 600m, so roughly 1.200km between two stops. At 30 km/h this translates to 2,4s driving time. It is also quite common to have buses that connect several cities without stops in between cities, we assume that they are at most 2min apart from each other. This means a minimum driving time of 6 seconds/1 tick (because it is our smallest time unit) and 120s/20 ticks. The maximum time saved from shortcuts is half of that, so 60s/10 ticks. The total duration for all stops is at maximum $9 trips \cdot 20 ticks + 8 \cdot 1 tick = 188 ticks$.

**Requests** The lead time for dynamic requests has to be at least 15 minutes, so 900s/150 ticks. The patience factor for calculating the maximum travel time is 4.
The stop is chosen uniformly distributed over all stops except the train stop. Whether the request is inbound or outbound, is decided by a random uniformly distributed boolean. The percentage of static requests is 60%. For inbound requests, the earliest pick-up time is randomly uniformly distributed chosen over the interval from the beginning of the simulation until the simulation duration minus its maximum travel time. For outbound requests, the latest drop-off time is chosen randomly uniformly distributed from the maximum travel time until the simulation duration.
The request time for dynamic requests is calculated randomly uniformly distributed over the whole simulation duration minus their maximum travel time. For the earliest pick-up time the following has to be met: $request\_time + lead\_time \leq earliest\_pickup\_time \leq simulation\_duration - request\_time - lead\_time - maximum\_travel\_time$. The exact time in the interval is also calculated randomly and uniformly distributed. Following that, the latest drop-off time is $latest\_dropoff\_time = earliest\_pickup\_time + maximum\_travel\_time$.

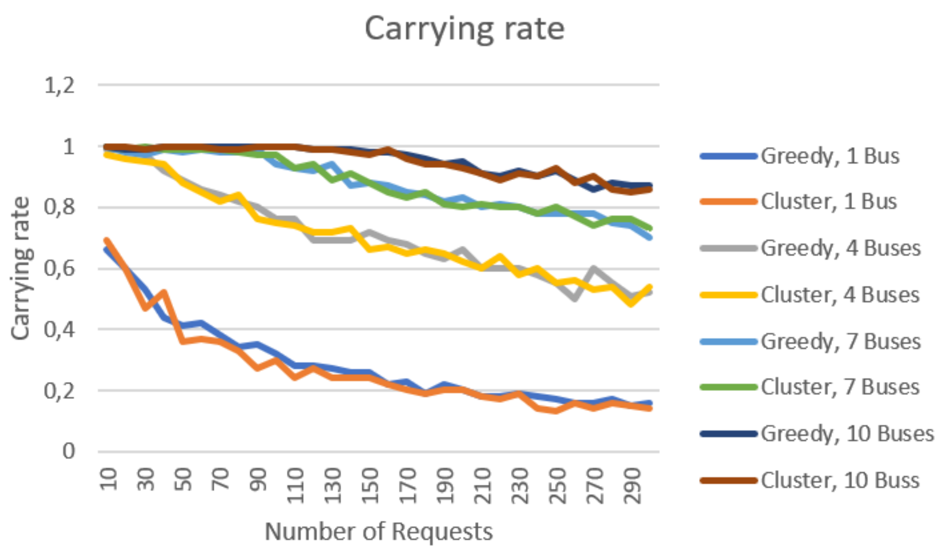

**Fig. 6.1:** Carrying rate for greedy and clustering algorithm for 1, 4, 7 and 10 buses

**Variable parameters** The simulation was tried for a variation of requests, from 10 requests up to 300 requests. For each step the number of requests was increased by 10 and the simulation for the step itself was done 10 times with a different seed for the generation of the requests for each repetition. The results of those 10 simulations per step are averaged.
The exact time and stops of the requests were chosen randomly with a uniform distribution, so each repetition for each step has different requests. The fixed parameter of 60% static requests and 40% dynamic requests remained the same.
The quantity of buses was varied at 1, 4, 7 and 10 buses.
The driving time between the stops and the time saved by shortcuts is also generated randomly for each repetition, so that the times can vary between our fixed parameters described above.

## 6.2 Evaluation of Results

### 6.2.1 Carrying Rate

The carrying rate over all was higher with an increased number of buses.

**Carrying rate for static and dynamic demand** The carrying rate for static demand is significantly higher than for dynamic demand as can be seen in Figure 6.2 and Figure 6.3.

**One Bus** When using only one bus, the greedy algorithm has a higher carrying rate than the clustering algorithm except at 10 and 40 requests. For both algorithms the carrying rate when using one bus drops from above 65% for 10 requests to between

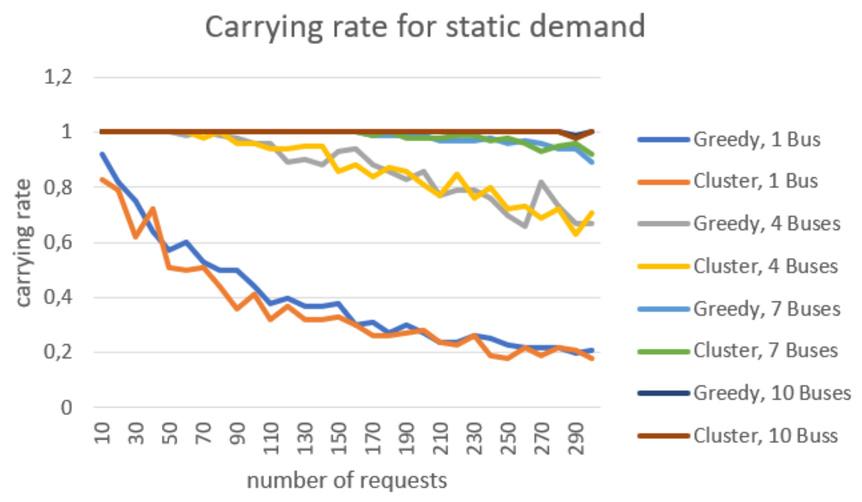

**Fig. 6.2:** Carrying rate for static demand for greedy and clustering algorithm for 1, 4, 7 and 10 buses

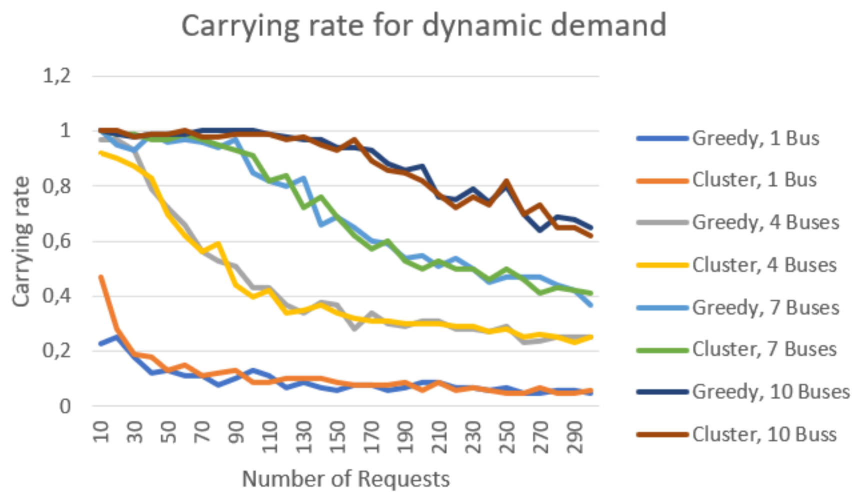

**Fig. 6.3:** Carrying rate for dynamic demand for greedy and clustering algorithm for 1, 4, 7 and 10 buses

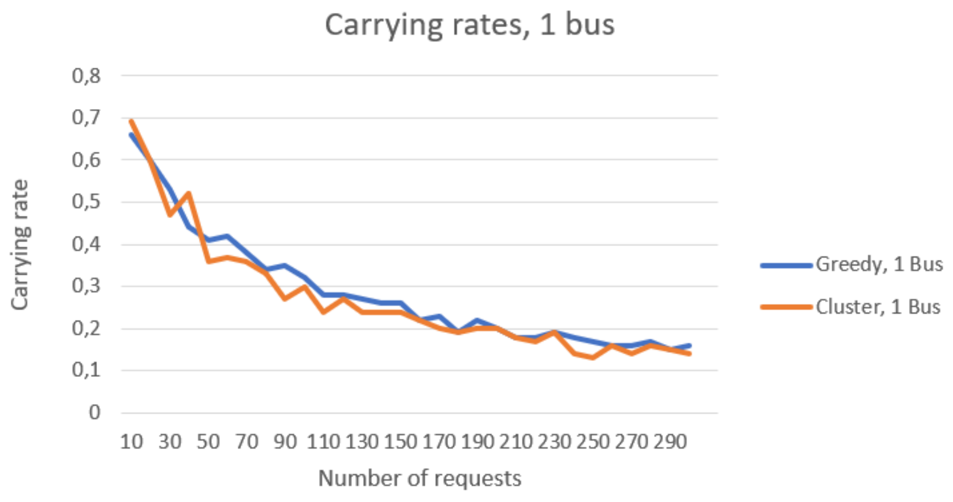

**Fig. 6.4:** Carrying rate for greedy and clustering algorithm for 1 bus

13% and 19% beginning at 210 requests. After the limit of 210 requests is reached, the carrying rate stays in this interval for both algorithms.
For one bus, the difference between the carrying rate for static and dynamic demand was higher the less requests were made. The difference when comparing clustering static and dynamic was less severe than when using the greedy algorithm for low request counts (36 Percent Points vs. 69 Percent Points at 10 requests). But with more requests, the difference between clustering and greedy algorithm becomes less severe (e.g. difference of 22 percent points at 160 requests for both algorithms).

**Four Buses** When using four buses, both algorithms perform roughly the same. The decline also doesn't flatten at the end unlike when using one bus. The carrying rate starts at 99% for the greedy algorithm and 97% for the clustering algorithm at 10 requests and drops to 52% for the greedy algorithm and 54% for the clustering algorithm at 300 requests. The first drop below 90% is at 50 requests for both algorithms.
Regarding the difference between static and dynamic demand, for up to 20 requests the difference is under 10 percent points. Then the difference increases drastically up to 60 percent points for clustering algorithm at 130 requests and 66 percent points for the greedy algorithm at 160 requests. It decreases slowly after that to 42 percent points for the greedy algorithm and 46 percent points for the clustering algorithm. For both static and dynamic demand, no one algorithm is better, not even if split into low and high demand.

**Seven Buses** When using seven buses, both algorithms perform roughly the same. Starting with 100% for both algorithms at 10 requests, the carrying rate drops more slowly at the beginning. The first drop below 90% is at 130 requests for the clustering algorithm and at 140 requests for the greedy algorithm. At 300 requests, the greedy

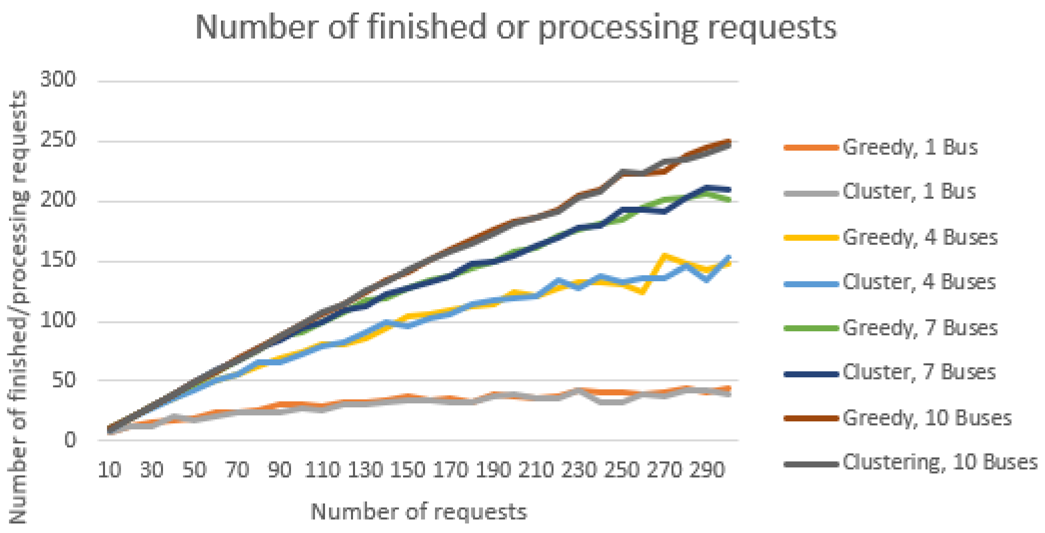

**Fig. 6.5:** Number of finished or processing requests for greedy and clustering algorithm for 1, 4, 7 and 10 buses

algorithm has a carrying rate of 70% and the clustering algorithm a carrying rate of 73%.
When using seven buses, the difference between dynamic and static demand stays below 10 percent points until 100 requests are made. After that the difference increases to up to 54 percent points for the clustering algorithm and up to 53 percent points for the greedy algorithm. Again, no one algorithm is better.

**Ten Buses** The pattern for seven buses to stay at a high rate for few requests, is stronger when using 10 buses. Both algorithms start at 100%. The first time they drop below 98% is at 140 requests for the clustering algorithm and at 150 requests for the greedy algorithm. They both stay above 90% even at 250 requests and drop to 87% for the greedy algorithm and 86% for the clustering algorithm at 300 requests.
Regarding the difference between dynamic and static demand, the pattern above intensifies. The difference stays below 10 percnet points until 160 requests and after that increases to 36 percent points for the greedy algorithm and 38 percent points for the clustering algorithm.

### 6.2.2 Number of finished or processing requests

As can be seen in Figure 6.5 the number of finished or processing requests increases with the number of total requests and also with the number of buses. It is independent from the algorithm used.
Adjusted for their percentage, more static requests are accepted than dynamic requests. As can be seen in Figure 6.6 when using 7 or 10 buses, the adjusted number of requests is almost equal at start, so lower than 0,1, but rises after that to up to 1,68. For one

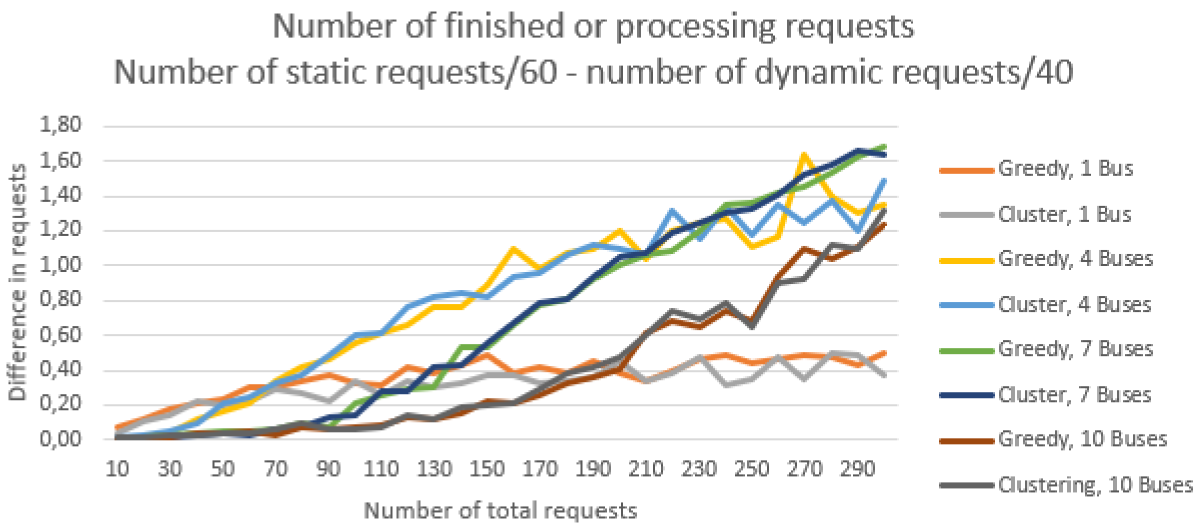

**Fig. 6.6:** Difference between static and dynamic requests for number of finished or processing requests for greedy and clustering algorithm for 1, 4, 7 and 10 buses adjusted for their percentage

bus, it is higher than 0,1 already at 20 requests, but never increases to more than 0,5. For four buses, it is higher than 0,1 at 50 requests for both the greedy and the clustering algorithm, but doesn't flatten like with one bus, but increases steadily to up to 1,64 for the greedy algorithm and 1,49 for the clustering algorithm.

### 6.2.3 Average Total Passenger Time

The average passenger time as defined in Section 4.5.2 drops slightly when the number of buses is increased. This is more drastic for the clustering algorithm starting with an average of 65 ticks for one bus and dropping to 46 ticks for 10 buses as can be seen in Figure 6.8. The greedy algorithm starts with an average of 55 ticks for one bus and drops to 45 ticks for 10 buses. For 4 to 10 buses, the average passenger time is similar. Regarding the difference between static and dynamic requests and the number of buses, there is only a significant difference when using one bus. For one bus the average passenger time for a static request is higher than for a dynamic request, over all request numbers the average passenger time is 56 ticks for static requests and 47 ticks for dynamic requests when using the greedy algorithm, 67 ticks for static requests and 52 ticks for dynamic requests when using the clustering algorithm.
An increase in number of requests correlates with an increased average passenger time. When using the greedy algorithm and 4 to 10 buses, there is a switch starting at 120 requests that static reqeusts take more or about equal time than dynamic requests. The change is more pronounced when using 4 buses. When using 7 and 10 buses, the difference between average passenger time for static and dynamic requests is less for 120 requests and up. For the clustering algorithm, the graph looks similar, but the graph

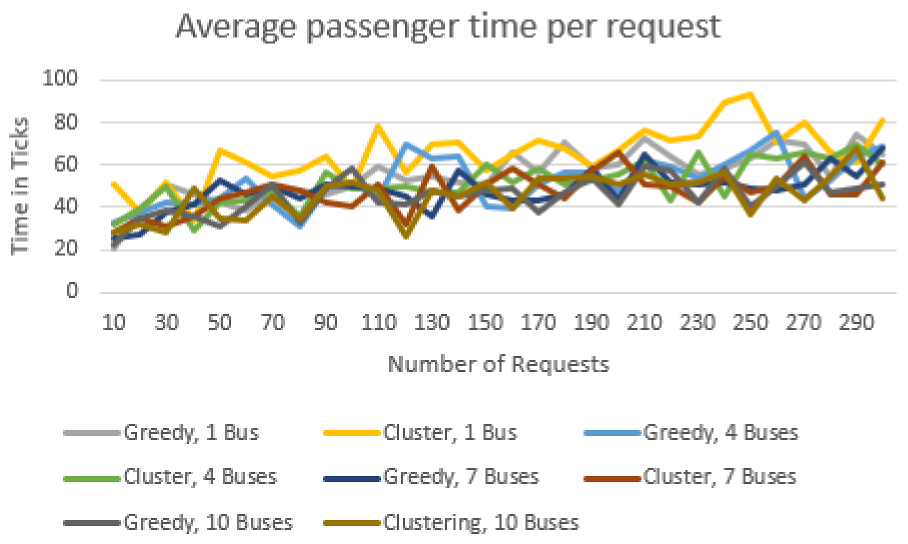

**Fig. 6.7:** Average passenger time for greedy and clustering algorithm for 1, 4, 7 and 10 buses

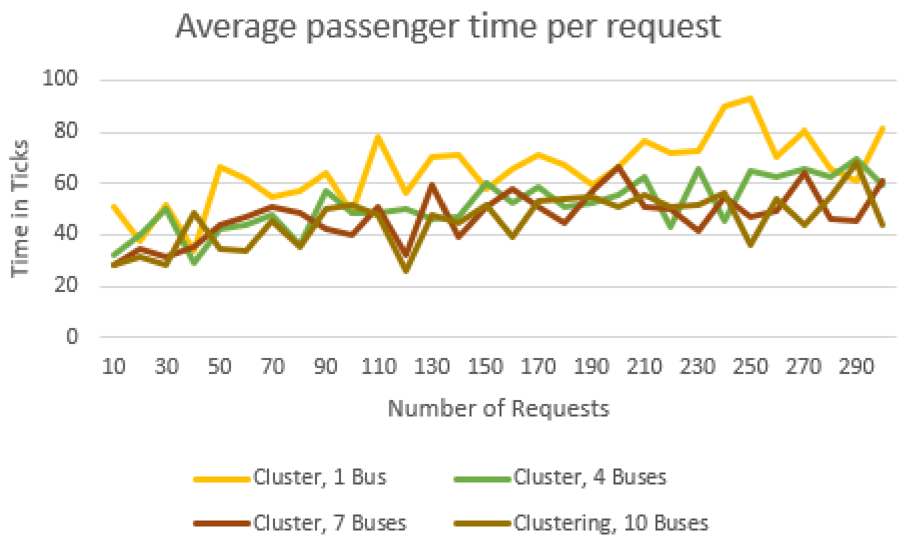

**Fig. 6.8:** Average passenger time for clustering algorithm for 1, 4, 7 and 10 buses

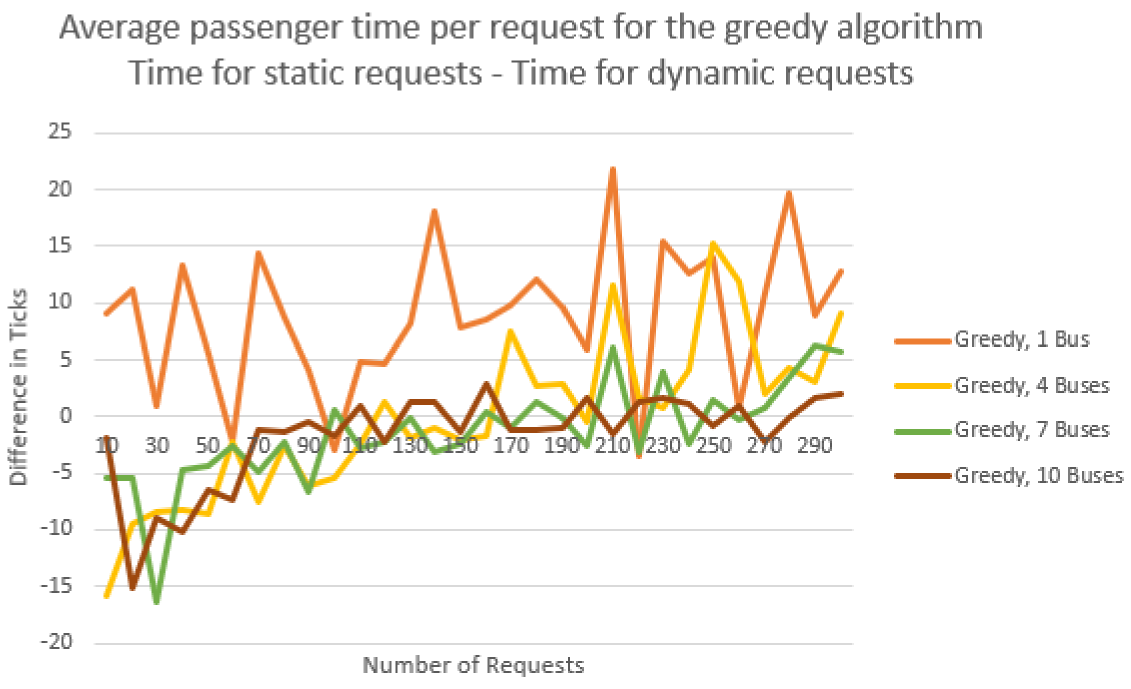

**Fig. 6.9:** Difference in average passenger time between static and dynamic requests for greedy algorithm for 1, 4 , 7 and 10 buses

varies more.

### 6.2.4 Average Total Transport Time

The average transport time as defined in Section 4.5.2 remains largely independent from the number of buses. The lowest average transport time is the clustering algorithm with 10 buses at 120 requests, the minimum of all other scenarios is at most 4,41 minutes slower. The longest average transport time is the clustering algorithm with one bus at 250 requests. The other scenarios are at most 5,92 minutes faster. The average transport time also only increases a little when more requests are added as can be seen in Figure 6.12, in which the figure was reduced to 4 buses for better readability. The choice of algorithm also doesn't affect the transport times very much, there is no clear pattern which algorithm is better in which context.

### 6.2.5 Detour Factor

For the detour factor, there is no significant differene between the two algorithms as can be seen in Figure 6.13. Both start with a detour factor of 0,1 or higher for 10 requests and then drop to 0,01 for 4 buses or more or 0,03 for one bus.
With more buses, the detour factor decreases more quickly related to number of requests and reaches 0 when using 10 buses.

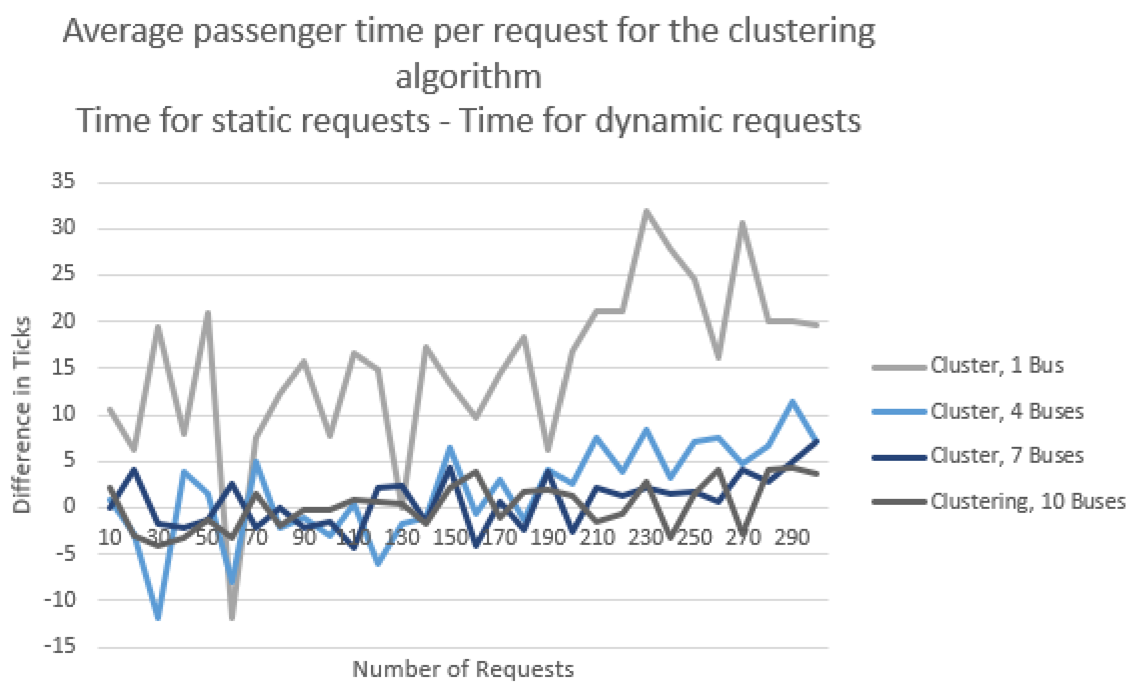

**Fig. 6.10:** Difference in average passenger time between static and dynamic requests for clustering algorithm for 1, 4 , 7 and 10 buses

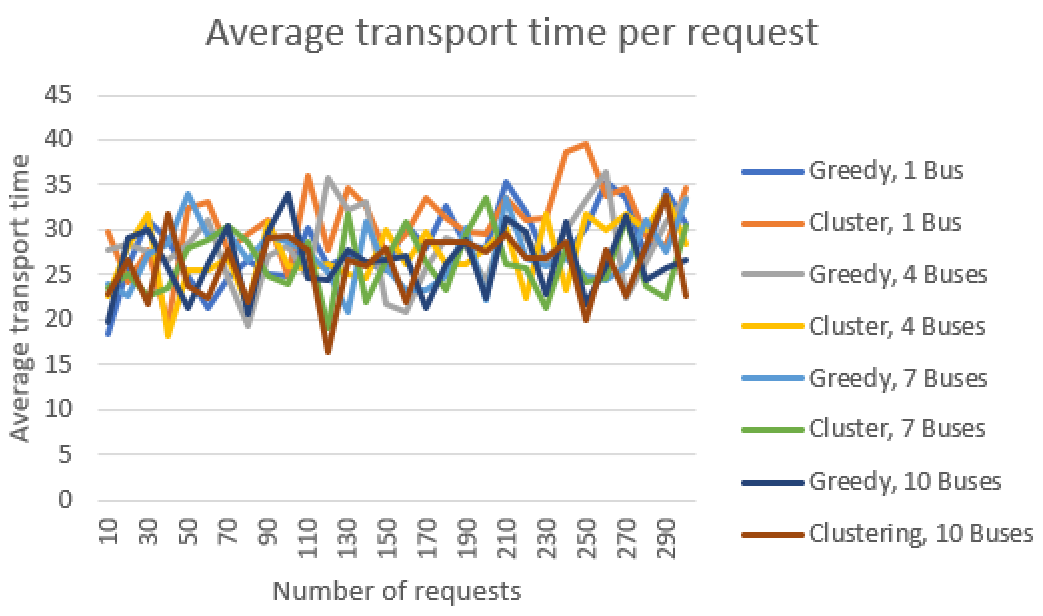

**Fig. 6.11:** Average transport time for greedy and clustering algorithm for 1, 4, 7 and 10 buses

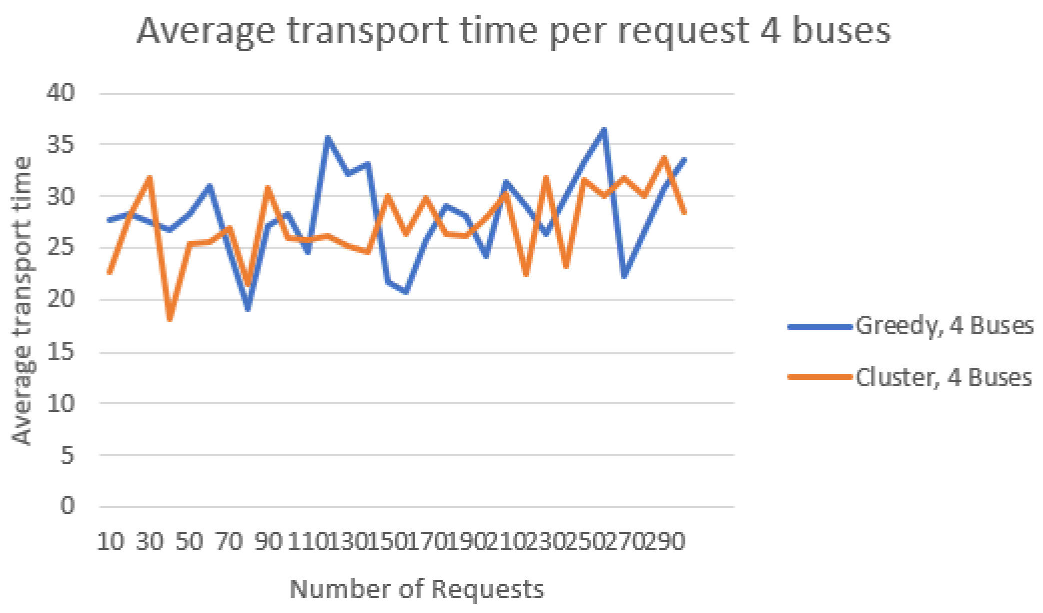

**Fig. 6.12:** Average transport time for greedy and clustering algorithm for 4 buses

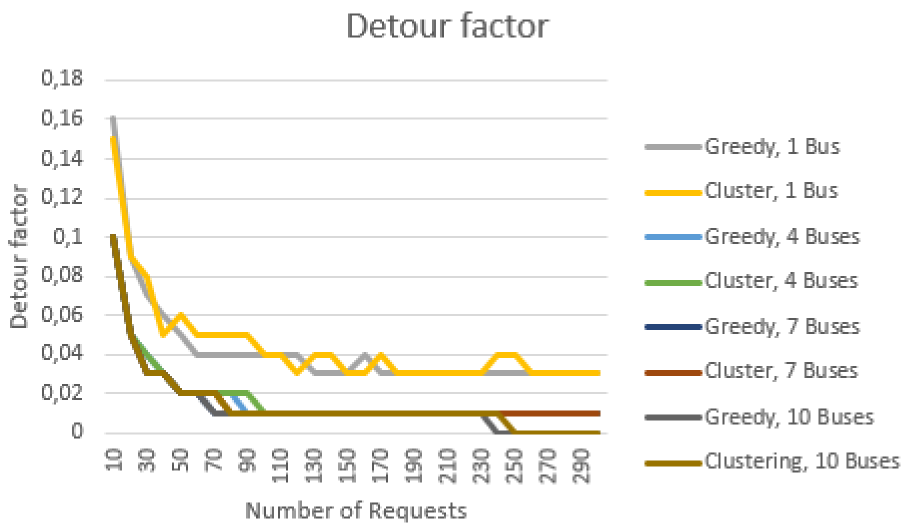

**Fig. 6.13:** Detour factor for greedy and clustering algorithm for 1, 4 , 7 and 10 buses

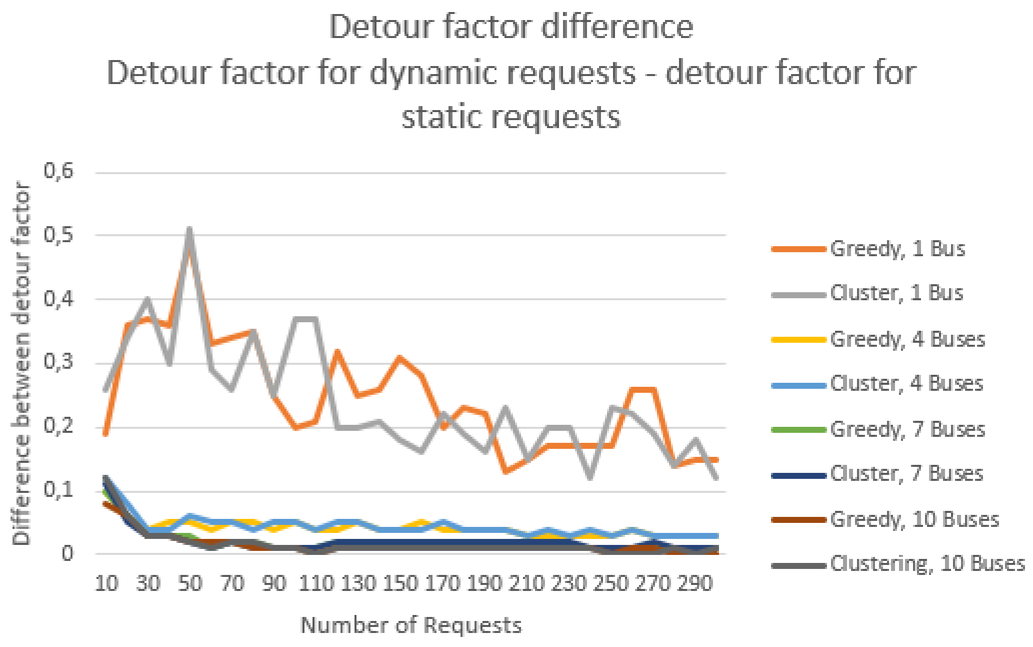

**Fig. 6.14:** Difference in detour factor between dynamic and static requests for greedy and clustering algorithm for 1, 4 , 7 and 10 buses

The detour factor is higher or equal for dynamic requests compared to static requests. This is most pronounced when using one bus at 50 requests: the difference is 0,49 for the greedy algorithm and 0,51 for the clustering algorithm. For 4 to 10 buses, the maximum is 0,12 for 10 requests when using 4 buses and the clustering algorithm, dropping steadily after that. When more buses are used, the difference between dynamic and static requests decreases as can be seen in Figure 6.14.

### 6.2.6 Empty Running Time

The empty running time remains between 28% and 40% as can be seen in Figure 6.15. It is largely independent from the algorithm used. Beginning at 170 requests, the empty running time is slightly lower when using one or four buses than when using seven or 10 buses and the clustering algorithm as can be seen in Figure 6.16.

### 6.2.7 Bus Utilization

The average bus utilization increases with the number of requests and decreases with the number of buses used. It is independent from the type of algorithm used as can be seen in Figure 6.17. In Figure 6.18 it is shown that except for at 10 requests, all buses that were available, were also used during the simulation.
The total bus driving time increases with both the number of requests and the number of buses used. It is also largely independent from the type of algorithm used as evident

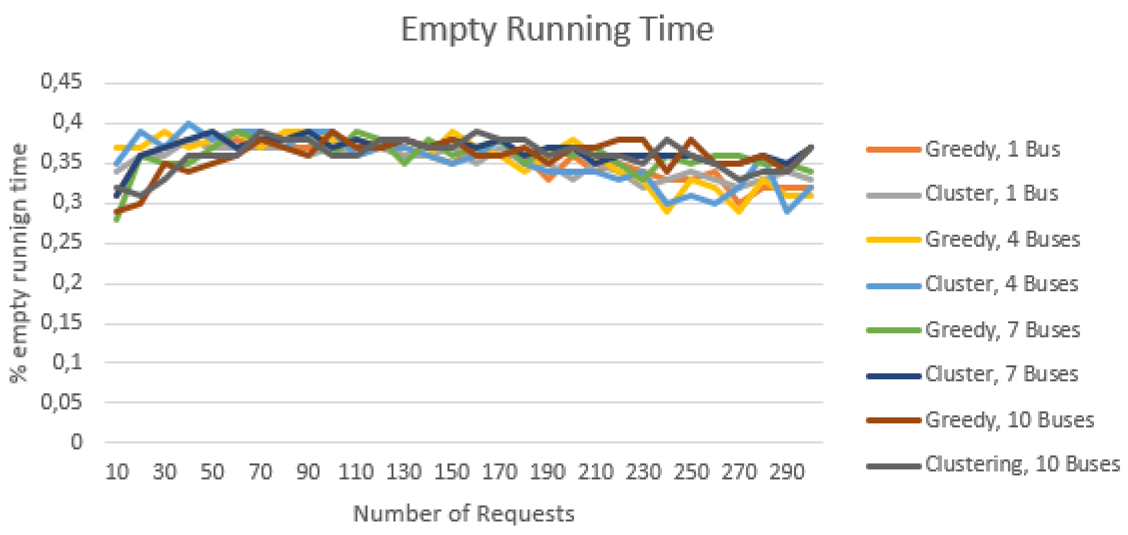

**Fig. 6.15:** Empty running time for greedy and clustering algorithm for 1, 4 , 7 and 10 buses

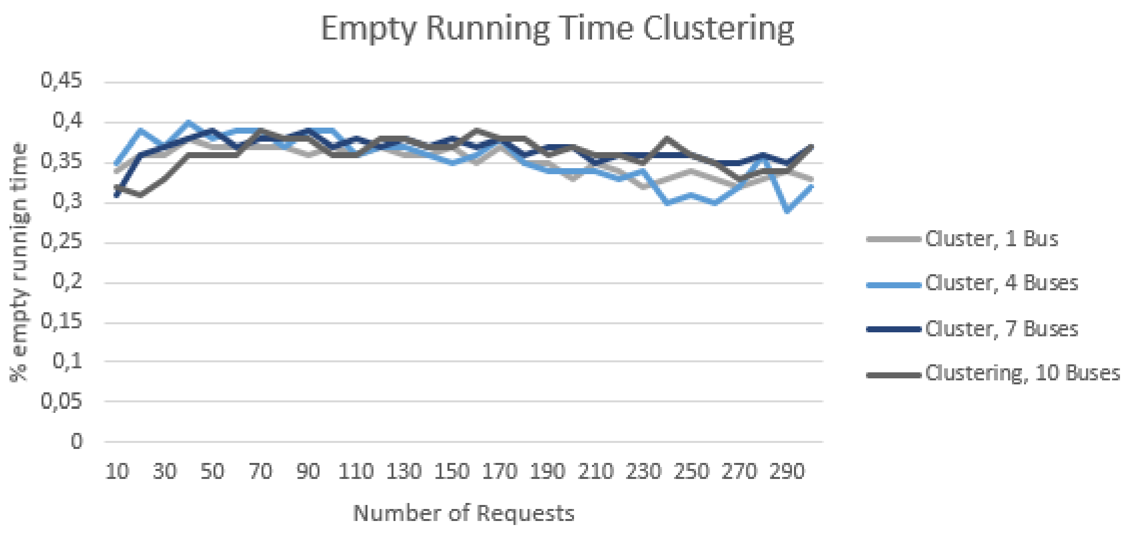

**Fig. 6.16:** Empty running time for clustering algorithm for 1, 4 , 7 and 10 buses

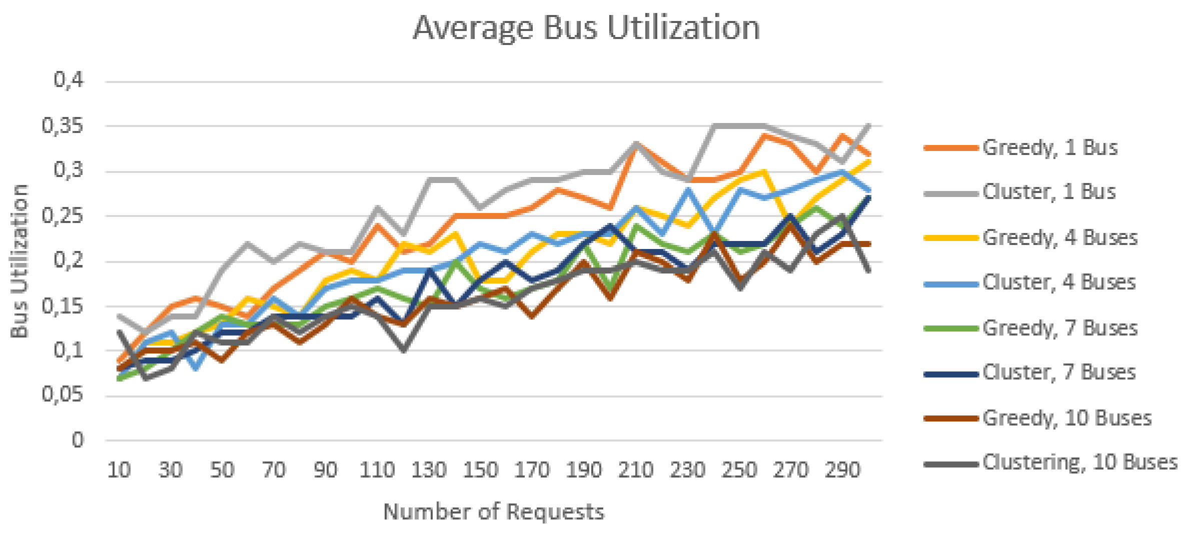

**Fig. 6.17:** Average bus utilization for greedy and clustering algorithm for 1, 4 , 7 and 10 buses

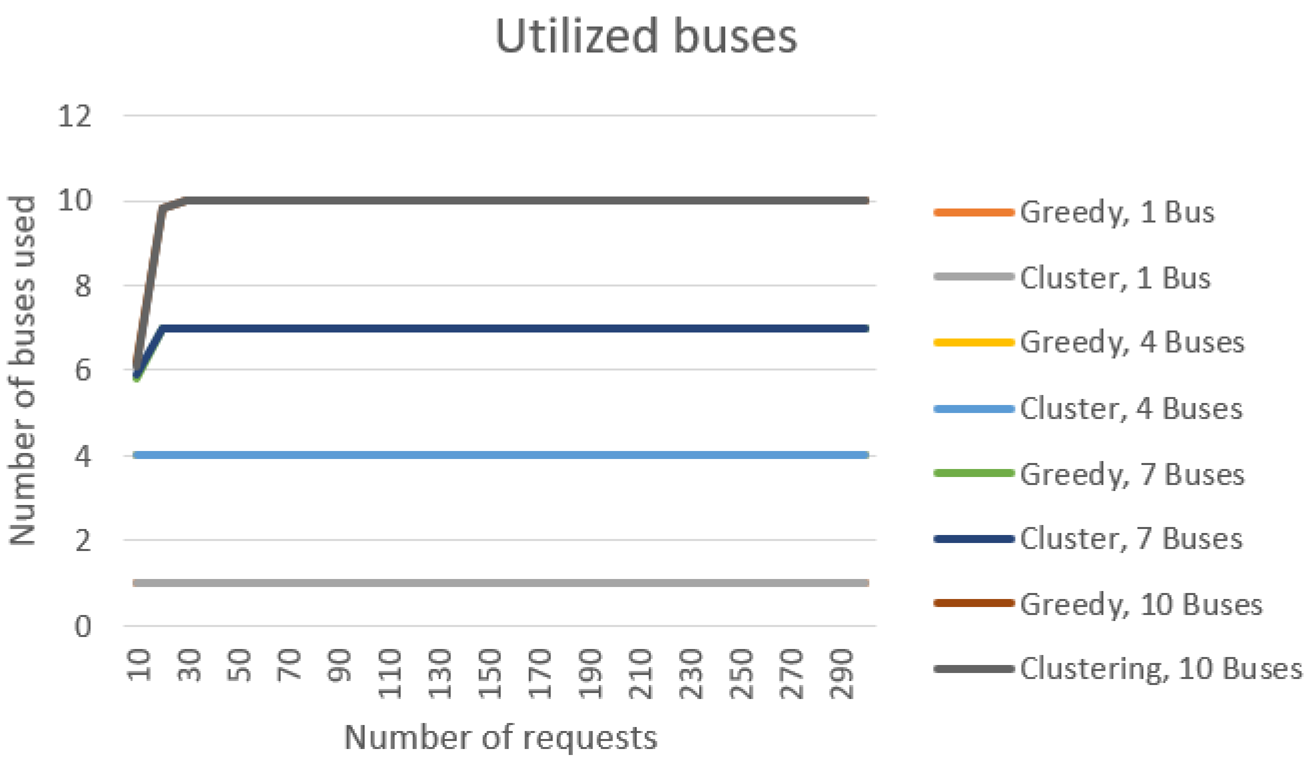

**Fig. 6.18:** Average utilized buses for greedy and clustering algorithm for 1, 4 , 7 and 10 buses. The numbers for the greedy and clustering algorithm are so similar, that only the lines for the clustering algorithm are visible.

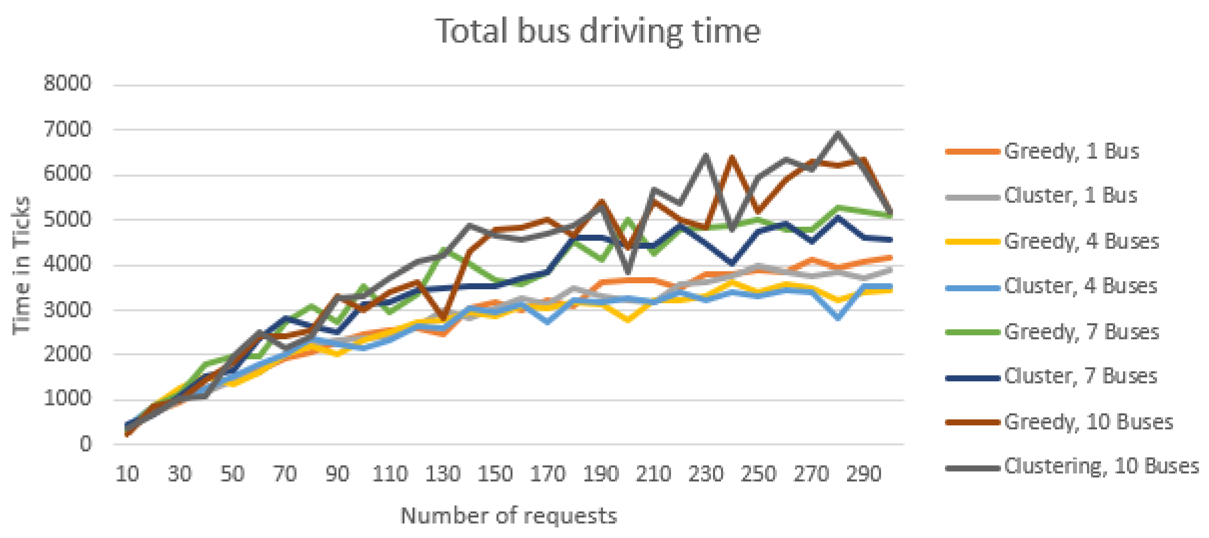

**Fig. 6.19:** Total bus driving time for greedy and clustering algorithm for 1, 4 , 7 and 10 buses.

in Figure 6.19.

### 6.2.8 Calculation Time

As shown in Figure 6.20 the average calculation time increases with the number of requests and the number of buses used. It is independent from the type of algorithm used.

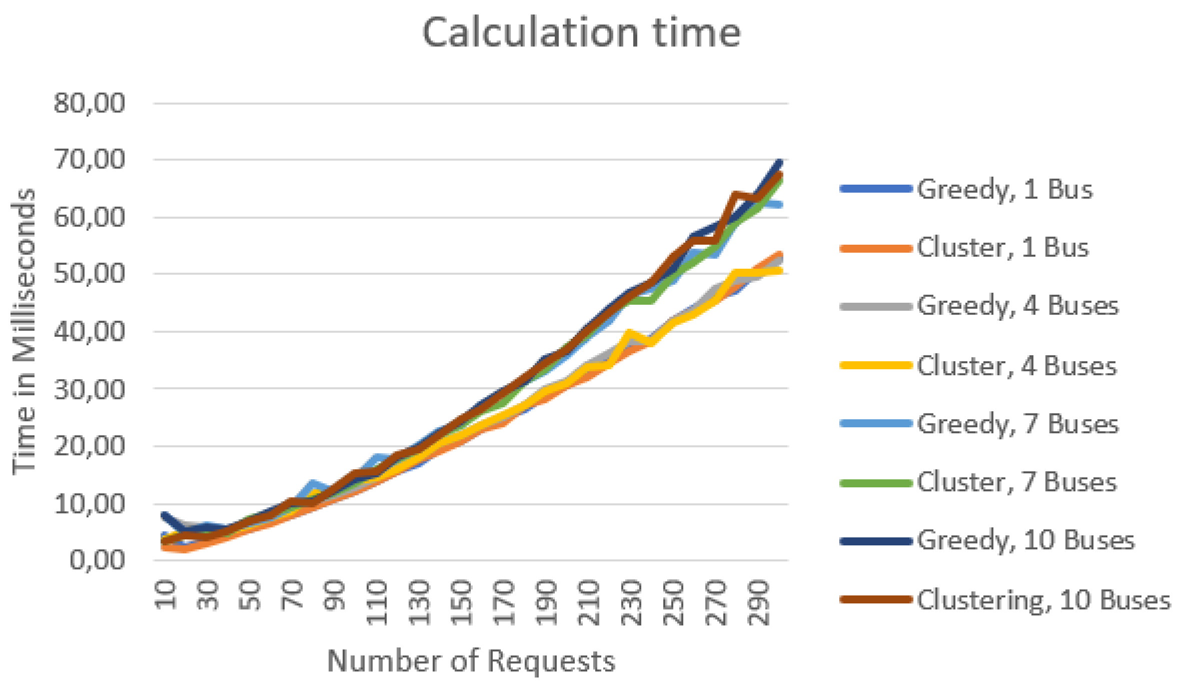

**Fig. 6.20:** Average calculation time for greedy and clustering algorithm for 1, 4 , 7 and 10 buses

# 7 Discussion

## 7.1 Carrying rate and number of finished or processing requests

When looking at 4, 7 or 10 buses, there was no discernible difference between the two algorithms. But with one bus, the carrying rate for the greedy algorithm is almost always higher than for the static algorithm. This could be due to the fact, that when using the greedy algorithm, the buses drive as soon as possible, while with the clustering algorithm, they wait as long as possible. In future work, it should be checked whether these numbers are true for the majority of randomly generate requests or if the numbers in this thesis are only because of the randomisation of the requests and coincidence.

The fact that the carrying rate increases with the number of buses and decreases with the number of requests is logical. If more buses are used, there is more space in general and there is more possibility for inserting a new request. Meanwhile, when the number of requests increases, the chance for a request to be impossible to fulfill even with a perfect algorithm, also increases.

The perfect route would have the minimum driving time, 1, for all driving times.
The perfect set of requests would be 4 requests at the train station times the number of buses, all with the earliest pick-up time of 1 and the drop-off stop neighboring the train station. Then again 4 requests at the neighboring stop with the earliest pick-up time 3, and so on and so forth. This means that in 4 ticks, a total of 8 requests times the number of buses could be transported, so for the whole simulation duration 2.000 requests times the number of buses.
If we assume the worst case for the route and the other stop being as far away from the train station as possible, while the distribution of requests would also be always 4 requests times the number of buses at the train station and 4 requests times the number of buses at the stop furthest away from the train station. The driving time from the station to the other stop would require 180 ticks. This means that in 364 ticks 8 requests times the number of buses can be transported, so for the whole simulation 24 requests times the number of buses (when counting processing requests).

With this in mind, it is also logical that the carrying rate drops after a certain threshold is reached. This can be seen for one bus, when comparing the number of finished or processing requests to the carrying rate: the absolute number of requests doesn't increase much after a certain point, while the carrying rate drops when more requests are added as the buses just can't transport more requests.

The fact that the carrying rate for static demand is higher than for dynamic demand is probably due to the fact, that for both algorithms static demand is considered first, simply because it is known at the beginning of the simulation.

## 7.2 Average total transport time and average total passenger time

Regarding the average total transport time, as described in Section 6.2.4, it is largely independent from algorith, number of requests or number of buses. This probably means, that there is not much room for improvement regarding the transport time.
When looking at the average total passenger time it decreases with an increase in buses and increases with an increase in requests. At least for the greedy algorithm, this can be expected as more requests on fewer buses means less flexibility. For the clustering algorithm, this result is somewhat surprising, because the buses should wait longer for newer requests, thus the waiting time should be larger when compared to the greedy algorithm. The reason for this result is probably, that dynamic results are planned with the greedy algorithm, overwriting some of the additional waiting time. It is also noticeable, that passenger time for static requests increase more compared to dynamic requests, the more requests are added. This can probably be explained by them being added earlier than dynamic requests and the algorithm favoring buses that are already near the desired stop. This has the potential to increase the passenger time for requests already planned for the bus, concerning more often static requests as they are planned earlier than dynamic requests.

## 7.3 Detour factor, empty running time and bus utilization

The detour factor decreases with the number of requests and with the number of buses. It is slightly higher for dynamic requests than for static requests which, again, can be explained by dynamic requests being inserted after static requests.
The empty running time is lower when using less buses for high demand. This can be explained by being able to pick more requests that fit into the current bus. The clustering algorithm could probably be deviated to optimise more for empty running time by rejecting requests far away from the station or only accepting requests far away from the station, when enough requests are gathered. The same is true for the bus utilization.

## 7.4 Calculation time

Although the calculation time increases with the number of requests and buses as expected, it stays below 68 milliseconds as presented in the evaluation. However, when the accepted and rejected requests in the heuristic are contained in an ArrayList instead of a HashSet as previously done in [9] this rises to over 23 seconds for 10 buses and

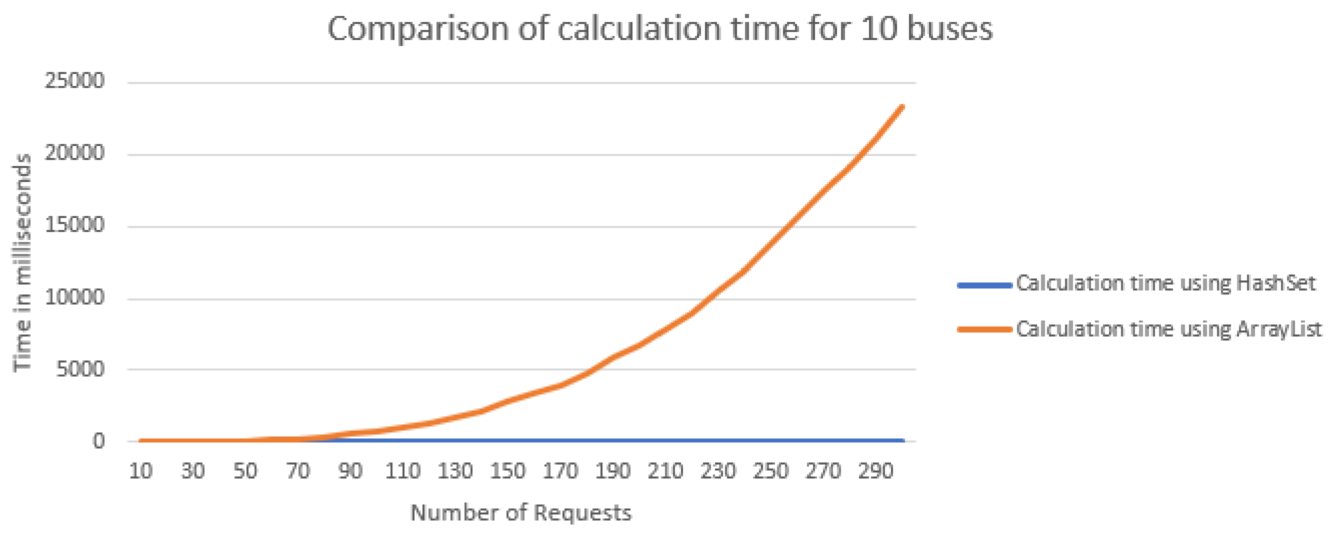

**Fig. 7.1:** Comparison of average calculation time for clustering algorithm for 10 buses when replacing the Sets of accepted and rejected requests with Lists

300 requests when using the clustering algorithm. This shows that the calculation time can sometimes be improved by rather simple measures and refactoring of the code, even without touching the logic of the heuristics.

## 7.5 Topics for further research

### 7.5.1 Variations to the clustering algorithm

In this thesis, the clustering algorithm was only handling static demand, while dynamic demand was always handled by the greedy algorithm. It is possible, that the integration of dynamic demand into the clustering algorithm could change its results for the better. This could then be compared to only the greedy algorithm, or the combined approach used in this thesis.
It would also be possible to further refine the clustering algorithm, e.g. by defining new information that mandates if a bus returns to the station or to the last stop after its last action is done. By strategically rejecting requests far away from the station, based on other known requests, the driving time of the buses as well as the passenger time of requests nearer to the station could possibly be improved.
Another variation of the clustering algorithm would be to only wait for further unknown requests when at the train station, while returning as soon as possible when the bus is not at the station. This could improve the criteria for inbound requests, while the effects on outbound requests would need to be the topic of research.

### 7.5.2 Variations to the parameters

In this thesis, we assumed that the demand is uniformly distributed for inbound and outbound requests over time. As we are looking at a feeder line and thus commute, it is reasonable to assume that traffic at different times of the day will vary towards one direction, e.g. in the morning there are more outbound requests and in the evening there are more inbound requests. It would be interesting to see, if the results comparing the greedy algorithm and the clustering algorithm would then be similar to this thesis or in favor of the clustering algorithm. Additionally, the clustering algorithm could be changed to consider the time of day and previous knowledge about demand patterns.
Similar to this, the demand could be altered, so that there is a higher demand for stops close to the train station, dropping the farther away we get from the station. In this case, especially when considering shortcuts, it is also possible that the clustering algorithm performs better than the greedy algorithm and in this thesis.

# 8 Conclusion

In this thesis, the simulation developed by [9] was expanded to include shortcuts, the combination of static and dynamic demand as well as two heuristics for inserting requests. One heuristic, the greedy algorithm, was based in parts on [9]' s greedy algorithm, while the clustering algorithm was newly developed and only applied to static demand.
The two heuristics were then compared for a range of 10 requests up to 30 requests and for 1, 4, 7 and 10 buses based on several criteria ranging from general aspects like the carrying rate to passenger-centric aspects like the average total passenger time and the average total transport time to operator-centric aspects like the empty running time, the detour factor or the bus utilization and total bus driving time.
In the results, there was no significant difference between the two algorithms, especially for the operator-centric aspects and for the passenger-centric aspects. The carrying rate for one bus even favored the greedy algorithm a bit, although it is not proven that this is not just statistical coincidence. The difference between dynamic requests and static requests was however visible, most probably due to fact that dynamic requests are planned after static requests.
However in Chapter 7 a lot of scenarios were suggested, in which the clustering algorithm could prove better than the greedy algorithm and also some ways to improve the clustering algorithm further. Changing the considered scenario and the simulation in these ways, could prove to bring interesting results and maybe further ideas for designing a way to use mini buses in a demand-responsive approach even including dynamic demand, that can also be realized for implementations outside of the scientific domain and inside the economic and public domain.
Another interesting finding is, that the runtime of the simulation not only depends on the heuristics and algorithms used in the program, but also on the data structures used. By replacing a list with a set, a significant amount of calculation time could be saved.

# Bibliography

[1] Marco Binswanger. Personenfluss im busverkehr: Optimale gestaltung von fahrzeugen zur verbesserung der fahrgastwechselzeiten. Master's thesis, ETH Zürich, 2017.

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

[3] Forschungsgesellschaft für Straßen-und Verkehrswesen. *Empfehlungen für Anlagen des öffentlichen Personennahverkehrs*, 2013.

[4] Giuseppe Inturri, Nadia Giuffrida, Matteo Ignaccolo, Michela Le Pira, Alessandro Pluchino, and Andrea Rapisarda. Testing demand responsive shared transport services via agent-based simulations. *New Trends in Emerging Complex Real Life Problems: ODS, Taormina, Italy, September 10–13, 2018*, pages 313–320, 2018.

[5] Christian Liebchen, Martin Lehnert, Christian Mehlert, and Martin Schiefelbusch. *Betriebliche Effizienzgrößen für Ridepooling-Systeme*. Springer, 2021.

[6] Shuliang Pan, Jie Yu, Xianfeng Yang, Yue Liu, and Nan Zou. Designing a flexible feeder transit system serving irregularly shaped and gated communities: Determining service area and feeder route planning. *Journal of Urban Planning and Development*, 141(3):04014028, 2015.

[7] Mingyang Pei, Peiqun Lin, Jun Du, and Xiaopeng Li. Operational design for a real-time flexible transit system considering passenger demand and willingness to pay. *IEEE Access*, 7:180305–180315, 2019.

[8] Antonio Pratelli, Marino Lupi, Alessandro Farina, and Chiara Pratelli. Comparing route deviation bus operation with respect to dial-a-ride service for a low-demand residential area. *DATA ANALYTICS 2018*, page 151, 2018.

[9] Leo Puppe. Vergleich von einer anfragebasierten und fahrplanbasierten minibuslinie durch simulation. 2023.

[10] Pieter Vansteenwegen, Lissa Melis, Dilay Aktaş, Bryan David Galarza Montenegro, Fábio Sartori Vieira, and Kenneth Sörensen. A survey on demand-responsive public bus systems. *Transportation Research Part C: Emerging Technologies*, 137:103573, 2022.

[11] Zhengwu Wang, Jie Yu, Wei Hao, Jinjun Tang, Qiang Zeng, Changxi Ma, and Rongjie Yu. Two-step coordinated optimization model of mixed demand responsive feeder transit. *Journal of Transportation Engineering, Part A: Systems*, 146(3):04019082, 2020.

Julius-Maximilians-
**UNIVERSITÄT WÜRZBURG**

# Versicherung zur selbständigen Leistungserbringung

Titel der Bachelorarbeit
Optimisation of static and dynamic demand on a feeder line

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):
Prof. Dr. Marie Schmidt, Lehrstuhl 1, Informatik - Fakultät

Eingereicht durch (Vorname, Nachname, Matrikel):
Andrea Rose, 2462592

Ich versichere, dass ich die vorstehende schriftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

☐ Mit dem Prüfungsleiter bzw. der Prüfungsleiterin wurde abgestimmt, dass für die Erstellung der vorgelegten schriftlichen Arbeit Chatbots (insbesondere ChatGPT) bzw. allgemein solche Programme, die anstelle meiner Person die Aufgabenstellung der Prüfung bzw. Teile derselben bearbeiten könnten, entsprechend den Vorgaben der PrüfungsleiterIn bzw. des Prüfungsleiters eingesetzt wurden. Die mittels Chatbots erstellten Passagen sind als solche gekennzeichnet.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Würzburg, 12.6.24 [illegible]
Ort, Datum, Unterschrift