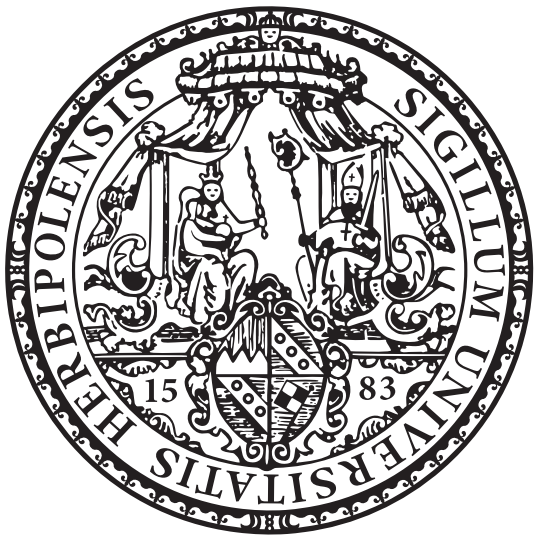
Master Thesis

Line-based Dial-A-Ride Problem with Transfers

Jonas Barth

Date of Submission: 16. June 2025
Advisors: Prof. Dr. Marie Schmidt
Kendra Reiter, M. Sc.



Julius-Maximilians-Universität Würzburg
Chair of Computer Science I
Algorithms and Complexity

Abstract

The *line-based Dial-a-Ride Problem* (liDARP) is a variation of the well-known *Dial-a-Ride Problem* (DARP), in which vehicles respond to requests on-demand but are constrained to operate along a predefined line. In this thesis we extend the liDARP to multiple lines by enabling users to transfer between them. We establish a comprehensive problem definition for the *line-based Dial-A-Ride Problem with Transfers* (liDARPT) and formulate an event-based MILP model to solve instances optimally. Implementing and testing the model shows that problem instances with up to eight requests an hour and 70 requests in total can be solved to optimality in under 15 minutes, where on average 99.7% of requests are accepted. While the system efficiency and acceptance rate are generally better in the DARP setting for medium sized instances, our tests reveal that the liDARPT approach scales better with more requests.

Zusammenfassung

Das *line-based Dial-A-Ride Problem* (liDARP) ist eine Variation des bekannten *Dial-A-Ride Problems* (DARP), in dem Fahrzeuge nachfragebasiert auf einer Buslinie fahren. In dieser Arbeit erweitern wir diesen Ansatz auf mehrere Linien zu einem Netzwerk, indem wir es Nutzern ermöglichen zwischen Fahrzeugen umzusteigen. Wir formulieren eine umfassende Problemdefinition für das *line-based Dial-A-Ride Problem with Transfers* (liDARPT) und stellen ein ereignisbasiertes MILP-Modell auf, mit dem Probleminstanzen optimal gelöst werden können. Nachdem wir das Modell implementiert und getestet haben, können wir feststellen, dass alle Testinstanzen mit bis zu acht Anfragen pro Stunde und 70 Anfragen insgesamt optimal in unter 15 Minuten gelöst werden können, wobei im Durchschnitt 99.7% aller Anfragen akzeptiert werden. Zwar sind Systemeffizienz und Akzeptanzrate im klassischen DARP in der Regel höher, jedoch skaliert der liDARPT-Ansatz besser bei zunehmender Anzahl an Anfragen.

Contents

1. Introduction

This thesis explores the capabilities of large on-demand bus networks for public transport, by discussing an algorithm to solve problem instances optimally and comparing the results to a different transport method.

Most people in modern society own personal cars, with numbers still rising in first world countries like Germany, see [fS24]. As this report shows, on average a group of 1,000 people owns roughly 580 cars. These are used for every day commutes, which leads to the vehicle occupancy typically being very low, in many cases being completely empty besides its driver. However, a lot of these trips could be shared in theory, due to similar origins and destinations or possible pick-ups along the routes.

This car dependency becomes even clearer when looking at the shares of motorized vehicles used in passenger transport. According to [Umw24], in their most recent evaluation done in 2021, the share of individual transport in Germany lies at 79.3%. Public transport alternatives only managed a quota of 4.3%.

It is well known that emissions in freight or individual transport is one of many factors leading to climate change caused by humans. The institution Climate TRACE reports transportation in general as the sector responsible for the 4th most CO_2e emissions in 2023 with 14.19%, see [TRA24]. These numbers call for closer attention and research towards new solutions in public transport to improve efficiency. Among other things, this could be accomplished by enhancing user-friendliness.

One heavily researched idea in the literature is constituted in the *Dial-A-Ride Problem* (DARP). Here, small public buses can roam freely within an area and pick-up and deliver users depending on their specific requests that are transmitted some time in advance. In most problem formulations the buses can be shared by multiple users, thus increasing the efficiency of the tours. Generally, this process can also be described as *Ridepooling*. There are multiple services like this available nowadays, for example in some communities in Germany. Typically they are referred to as *Rufbus*, but also more specifically *Holmich-App* in Wuppertal, see [Flo20] and *Call-Heinz* in the rural region of Schweinfurt, [Sch24]. Requests can be submitted easily through dedicated smart phone applications or by phone.

In [RSS24], the authors proposed the *line-based Dial-A-Ride Problem* (liDARP) as a variation of the standard DARP. The main difference being that a bus is now assigned to a specific line, a series of stops it will always travel along. The bus will change directions and pick-up users based on the passenger demand.

Obviously, for this to be efficient and attractive for users, we have to assume that the predefined line connects a number of different areas, with easy access for as many users as possible and without overly extensive deviations. Under this assumption, the hope is to encounter less detours and thus more predictable routes and planning for users

compared to standard DARP. As the requests should be more cohesive, they would be easier to match, which should help us from an algorithmic point of view and more importantly, improving the efficiency of the entire system, meaning more shared rides.

Generally, this approach could be especially interesting for bus networks in rural regions. Since the overall demand for public transport is rather low, because of the lower population density. Due to this, bus companies tend to reduce the frequency of buses. Which again hurts the flexibility when using these services, creating a negative feedback loop. In theory he liDARP approach could fix some of these issues, by offering more flexibility to users and a higher occupancy of the buses per ride.

In this thesis we want to look further into the liDARP. Especially, extending previous work by not only looking at a single line, but instead into bigger, overlapping networks and enabling the users to transfer buses at certain stops. For this, we establish an event-based *mixed-integer linear programming* (MILP) model, similar to [RSS24], where, depending on a network of lines and corresponding buses, as well as a set of requests from users, we try to find a routing of the buses and users, so that the overall number of transported users is maximized. By enabling transfers between buses we have to look at the whole network simultaneously, instead of just one line at a time. The network could also contain cycles, which leads to a number of different routing possibilities for a single request. Considering these extensions, we refer to this problem as the *line-based Dial-a-Ride Problem with Transfers* (liDARPT).

The additional factors for this problem can lead to a lot more inconveniences and especially a bigger model, compared to the liDARP, which we aim to tackle within this thesis. In the process of this thesis, we implement a software framework, that is built specifically for solving the liDARPT. For this, we implement an event-graph, similar to [RSS24], an approach used for the DARP first by [GKS22]. By utilizing this framework, different lines and sets of requests are tested and evaluated. We also compare these results to the variation of employing standard DARP with the same set of requests, while still solving optimally.

In the next section, Section 1.1 we will look into previous work in this area and similar approaches of enabling transfers in a DARP formulation, followed by giving a mathematical definition of the liDARPT in Section 1.2. In Chapter 2, the basic event-based linear model of [RSS24] is explained in detail, as well as the necessary adaptations made for enabling transfers. Afterwards the software implementation is discussed, where different techniques are employed to ensure acceptable runtimes, Chapter 3. The used parameters for testing and generating data are presented in Chapter 4 and the results are evaluated. We conclude our findings and give an outlook on possible ways to extend this research in Chapter 5.

1.1. Related Work

The well-known *Vehicle Routing Problem with Time Windows* (VRPTW) can be seen as the earliest framework, that incorporates similar challenges as we are discussing here. It is a very broad and overarching problem, that can be specified into many other more

detailed scenarios. In its original form multiple locations are given, that are supposed to be visited by a vehicle, within their respective time windows. A lot of research was conducted in the 1990s and early 2000s, for example in [DDS92] and [TLZO01], where the authors looked at solving instances exact via linear programming or used heuristic methods, respectively. *Ridepooling* is very similar and constitutes a special case of the VRPTW.

As the DARP is the corresponding mathematical formulation for *Ridepooling*, it is an extensively researched topic. It can be considered in a static context, where all requests are known in advance. However, a dynamic setting is also possible. In this case user requests come in as time goes on and routes have to be adjusted on the fly. Most of the research can also be distinguished by either solving the problem in an exact manner or using heuristics, especially metaheuristics. We will cover static and exact approaches in this section, as that is what we will focus on in this work. Firstly, a general overview of previous research can be found in [HSK$^+$18].

The 3-index formulation from [CL07] was hugely influential in the field. The authors engineered a MILP on a location-based graph, with an integer variable for every path between two points and bus. Furthermore, they suggest a branch-and-cut algorithm to solve it and do so for instances with up to 50 requests. Only shortly after [RCL07] improved on this work by proposing a 2-index variable formulation of this MILP. They accomplish this by removing the indexing of the cars, only tracking the activation of an edge and inferring the routes from that. This approach was considered the state-of-the-art for a long time.

Recent notable findings were achieved by [RF21]. In their formulation, the routes of the buses were split up into multiple fragments and the MILP is constructed on the selection of those, instead of single paths. This leads to less variables and thus was shown to have faster running times. Finally, the event-based approach shown in [GKS22] and [GKP$^+$24] resulted in greatly reduced running times and solved instances of up to 80 requests in less than 7 seconds optimally. In an event-based graph, a node represents not only a location, but also a specific subset of passengers in the vehicle at this point. This allows for many combinations in theory, but reduces the number of arcs greatly, with the right preprocessing steps.

In [RSS24] the liDARP was established. For the specific problem variation multiple MILP formulations were tested, with the event-based strategy coming out on top. The authors managed to solve instances with up to 50 requests in under a second. In this paper, only a single line was considered. A big challenge for this thesis, will be trying to enable a network of lines with transfers.

Most of the work, on the classic DARP with transfers, has focused on establishing few transfer points in the network, see [MLP14]. Here, the authors declare the Dial-a-Ride with Transfers(DARPT), where they consider the static setting and use the given demand to generate transfer points, to minimize the overall travel duration. Only at these points passengers can change vehicles. A methaheuristic called *Adaptive Large Neighborhood Search* was employed to solve the instances. When comparing results of optimal solutions of the DARP, to heuristic ones of DARPT on the same instances, it could be seen that the latter achieved slightly better performance on average. The

instance sizes ranged from 16 up to 96 requests.

The authors in [DQ13] also worked with the DARPT framework. To solve the problem an insertion-heuristic was introduced. When using their specific heuristic, enabling transfers lead to 11.9% more accepted requests compared to standard DARP. In their experiments the demand consisted of 24 to 144 requests.

Finally, [GN23] recently proposed a model definition, where interchange is possible at all pick-up and drop-off positions. They solved the problem exactly via Branch-and-Cut. As expected this lead to longer computation times, so they also tested the metaheuristic tabu-search for larger instances. With the second approach, solutions for up to 200 requests were found in less than 30 minutes.

1.2. Problem Description

In this section, we will present a mathematical definition of the liDARPT. We consider a set of buses K, with each bus driving on a line $l \in L$. Every line l consists of an ordered sequence of stops $(s_1, \dots, s_{|l|})$. Each bus is assigned to a single line by $\sigma(k) \in L$. However, one line can have multiple buses and we consider all of them to have the same maximum capacity of passengers $c_l \in \mathbb{N}$. A bus can drive along the line in both directions. It is allowed to skip bus stops, however for simplification a bus is only permitted to turn around when empty. If a bus starts or ends its tour, it always drives to the depot s_{depot} of its corresponding line. The depot can be one of the stops of the line, or a different location altogether. For two stops on a line s_i, s_j we define function t to calculate the travel time necessary for driving directly to one from the other: $t(s_i, s_j) \in \mathbb{R}$. The triangle inequality holds for all stops s_i, s_j, s_m and t:

$$t(s_i, s_j) \leq t(s_i, s_m) + t(s_m, s_j).$$

As input we receive a set of customer requests R, each request $r \in R$ consists of a number of persons $q_r \in \mathbb{N}$, that want to be transported. Furthermore, a request is attached to a pick-up r^+ and a drop-off r^- and for both there is a time window $[e_{r^+}, \ell_{r^+}]$ and $[e_{r^-}, \ell_{r^-}]$ in which the related action needs to occur. Additionally, each request has a maximum ride time $\mathcal{L}_r$, which the users overall traveling time, from the first pick-up to final drop-off, must not exceed. Generally, we assume a fixed service time π, this is the amount of time needed to handle all combined users getting in or out of the vehicle at a stop. No matter if that is only one user being picked-up or multiple users leaving, while others enter the vehicle.

We aim to find a plan Q_k for every bus k, i.e., a sequence of stop actions $(\rho_1, \rho_2, \dots)$. An action ρ consists of a number of properties: It is identified by the stop location $o_\rho \in \sigma(k)$, corresponding time of arrival e_ρ as well as departure ℓ_ρ. To track the routes each user takes, there is also a set of requests for drop-off D_ρ and a set of requests for pick-up P_ρ, attached to each stop action. Both of these sets consist of requests, of which there can never be more than the capacity of the bus, $|D_\rho|, |P_\rho| \leq c_{\sigma(k)}$. For the plan to be feasible, all travel times have to be respected. Meaning the following holds for all subsequent stop events ρ_i, ρ_j in the plan Q_k:

Identifier	Description
K	set of buses
k	bus
$\sigma(k)$	function that assigns a line to bus k
c_l	capacity of buses on line l
L	set of lines
l	line
s_i	i-th stop location of a bus line
s_{depot}	start and end location of bus tour on a line
$t(s_i, s_j)$	distance function for two stops
R	set of requests
r	request
q_r	number of passengers for request r
r^+, r^-	pick-up, drop-off action for request
$[e_{r^+}, \ell_{r^+}]$	pick-up time window
$[e_{r^-}, \ell_{r^-}]$	drop-off time window
$\mathcal{L}_r$	maximum travel time of request r
π	service time
Q_k	plan for bus k
ρ_i	i-th stop action of plan Q_k
o_ρ	location of stop action ρ
e_ρ, ℓ_ρ	arrival, departure time of stop action ρ
P_ρ, D_ρ	set of requests picked up, dropped off at action ρ
p_r	decision variable for request r

Tab. 1.1.: Variable Description

$$t(o_{\rho_i}, o_{\rho_j}) \leq e_{\rho_j} - \ell_{\rho_i}.$$

In this problem formulation it is possible to deny some requests. This can be achieved by defining a decision variable $p_r \in \{0, 1\}$ for every request $r \in R$. If $p_r = 1$, the request r counts as accepted and the above detailed plan would need to ensure the following properties. First we can generate a route for r, which refers to a sequence of stop action tuples $((\rho_1, \rho_2), (\rho_3, \rho_4), \ldots, (\rho_{m-1}, \rho_m))$. The intuition behind this being, that every tuple describes the users' path on a single line. They enter the bus at the first entry and step out at the second entry of the tuple. So the composition of tuples forms the entire route a user follows, using one or multiple buses, to get to their destination. These conditions are laid out mathematically below:

- for every tuple (ρ_i, ρ_{i+1}): $r \in P_{\rho_i}$ and $r \in D_{\rho_{i+1}}$ and $\rho_i, \rho_{i+1} \in Q_k$ for a $k \in K$
- for action ρ_1: $r \in P_{\rho_1}$, $o_{\rho_1} = r^+$ and $\ell_{\rho_1} \in [e_{r^+}, \ell_{r^+}]$
- for action ρ_m: $r \in D_{\rho_m}$, $o_{\rho_m} = r^-$ and $e_{\rho_m} \in [e_{r^-}, \ell_{r^-}]$
- for every consecutive pair of tuples in the route $(\rho_{i_1}, \rho_{i_2}), (\rho_{j_1}, \rho_{j_2})$: $o_{\rho_{i_2}} = o_{\rho_{j_1}}$ and $\ell_{\rho_{j_1}} \geq e_{\rho_{i_2}} + \pi$
- $\mathcal{L}_r \geq e_{\rho_m} - \ell_{\rho_1} + \pi$

As we already feature hard time constraints in this formulation, we do not consider this in our optimization goal. Instead we focus on transporting as many users as possible, while secondly minimizing the overall traveled distance by the buses. We assume the travel time to be directly proportional to the distance, thus we simply minimize the travel duration here. This can be achieved by using a penalty W added to the sum for every denied request and leads to the following objective function:

$$\min \sum_{k \in K} \sum_{i=1}^{|Q_k|-1} t(o_{\rho_i}, o_{\rho_{i+1}}) + W \sum_{r \in R} (1 - p_r). \tag{1.1}$$

2. Event-Based Model

The goal is to find an optimal solution for the liDARPT. For that, the standard approach is to generate a MILP model based on the given network and requests. However, as we have already discussed in Section 1.1, different model compositions are possible. For the liDARP, multiple approaches were tested by [RSS24] and the method that lead to the fastest computation times was the event-based formulation. Thus, in this chapter we will focus on this strategy and formulate a model suitable for the liDARPT based on it.

This method was introduced for the DARP by [GKS22]. It involves an extra step where events have to be created first, which are then linked to each other in an event graph. An event can be explained as a specific user allocation to a bus at any point in time. Typically, it is either a pick-up or drop-off event, meaning users of one request are joining or leaving the bus at each event, while the other remaining passengers stay in the vehicle.

For a line l we denote an event as a tuple $(v_1^+, v_2, \ldots, v_{c_l})$ or $(v_1^-, v_2, \ldots, v_{c_l})$ for the case of pick-up or drop-off, respectively with $v_i \in R \cup 0$. We write $v_i = 0$ to indicate the absence of users. In the simplest case $q_r = 1$ for all requests and each element v_i of the tuple corresponds to a single seat in the vehicle. However, as we allow for different numbers of users for requests, this is not generally true and we have to check that the vehicles maximum capacity is not exceeded for an event. The requests $v_2, \ldots, v_{c_l}$ are always written in ascending order, so there is only one event for a request action and specific set of other requests. Finally, there is an idle event $(0, 0, \ldots, 0)_l$ added as well for every line $l \in L$, which symbolizes a bus starting and stopping at its depot.

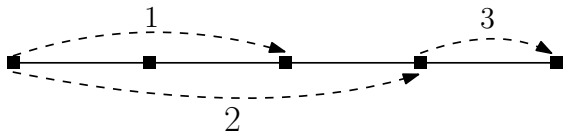


Fig. 2.1.: Example of a network and requests.

To gather a better understanding of this approach, we look at a simple example involving a single vehicle with a capacity of four. In Figure 2.1, a single line and stop locations are shown. Furthermore, three requests $\{1, 2, 3\}$ and their corresponding routes are depicted. By accounting for local constraints, like two requests that travel along completely distinct parts of the line, or timing constraints, many events can be eliminated in preprocessing. Thus, only *feasible* events are left. After eliminating events based on local properties, all possible events left for this instance are listed below.

- $(0, 0, 0, 0)$
- $(1^+, 0, 0, 0)$, $(1^+, 2, 0, 0)$, $(1^-, 0, 0, 0)$, $(1^-, 2, 0, 0)$
- $(2^+, 0, 0, 0)$, $(2^+, 1, 0, 0)$, $(2^-, 0, 0, 0)$, $(2^-, 3, 0, 0)$
- $(3^+, 0, 0, 0)$, $(3^+, 2, 0, 0)$ $(3^-, 0, 0, 0)$

The event graph $G = (V, A)$ uses directed edges, referring from one event to a possible subsequent event, meaning that the bus could drive from the first to the second, while carrying out the corresponding pick-up or drop-off actions. The MILP model is required to then find a path through the event graph for every bus, that starts at and returns to its line's idle event. A number of additional constraints are needed as well to enforce the number of picked up requests and correct timing of stops.

By employing some basic conventions, we can disregard even more events and arcs. For example, for two or more actions happening at the same stop, the drop-off actions are always handled first. As we will discuss later in Chapter 3 every action of a request receives its own time window. We can use these time windows to process events in descending order of their time window end. Not only does this shrink the number of edges in the event graph, it is also an important technique that helps enforce time windows in our model, on which we will elaborate in Section 2.2. Consequently, if there are multiple requests available for pick-up at a certain stop, we consider just one specific sequence they can be picked up in, by keeping only a specific subset of edges in the graph. In the example from Figure 2.1 the events $(3^+, 2, 0, 0)$, $(2^-, 3, 0, 0)$ can be eliminated, without restricting the set of achievable routes, as they do not follow our first convention.

Finally, there is the possibility of using time constraints to exclude some events or arcs from the event graph. This could happen for example if two requests are planning their travel for completely different points in time. So the combination of both of the requests' users in the same vehicle will never be possible. Similarly, we can leave out some edges from the graph, as we can see for the example in Figure 2.1, by assuming the earliest departure time of request 3 is later than the latest arrival times of both of the other requests. In conclusion, only the arc back to the depot is necessary after event $(3^-, 0, 0, 0)$. The resulting event graph is shown in Figure 2.2.

At first glance, the event-based approach can lead to a very big graph size. As shown in [GKS22] the number of nodes is constrained by

$$V \leq |L| + 2|R| \sum_{i=0}^{\max(c_l)-1} \binom{|R|-1}{i}. \tag{2.1}$$

This is due to the fact that we always consider one depot node $(0, 0, \ldots, 0)_l$ for every line $l \in L$, and for every request in R and both of its actions, we count all possible subsets in R of size at most $\max(c_l) - 1$, with $\max(c_l)$ denoting the highest capacity among all lines. For $|R| > |L|, \max(c_l)$, the number of nodes is bounded by $\mathcal{O}(|R|^{\max(c_l)})$.

In their work the authors of [GKS22] conducted similar assessments for the number of edges and arrived at an upper bound of $\mathcal{O}(|R|^{\max(c_l)+1})$. By comparison, in the standard

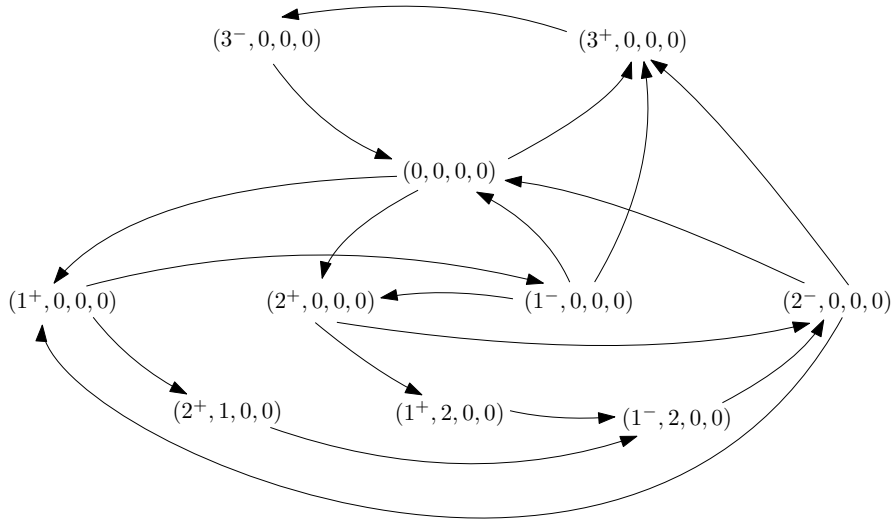


Fig. 2.2.: Event graph for the instance in Figure 2.1 after preprocessing and pruning unnecessary events.

location-based approach, the MILP model primarily receives a graph of the network and the necessary stops for the given requests. Thus, the graph is in $\mathcal{O}(|R|)$. However, most of the information on requests, like their time windows, or enforcing the maximum capacity of the vehicles has to be handled explicitly.

Furthermore, we generally consider networks with a more sparse demand, meaning only small buses are needed with $4 \leq \max(c_l) \leq 6$. Coupled with the use of preprocessing steps, where many events and edges can be eliminated in a lot of problem instances, the MILP model often takes less time for computing the optimal solution, compared to the location-based strategy, as shown by [RSS24].

2.1. Base Model

First, we will inspect the basic MILP model used for the liDARP as shown in [GKP$^+$24], but using our own notation. Additionally to the variables p_r for each request r, every edge $a \in A$ in the event graph receives a corresponding binary variable x_a. Also, for every pick-up or drop-off action we add a continuous variable B to the model. This variable will depict the exact departure time of the vehicle after the action takes place.

Overall, the formulation is guided by the location-augmented event based model described in [GKP$^+$24]. The authors used the special approach of sharing the time variables B for all events of the same action. So we construct the model in a way, where a variable can be part of many different equations, but we guarantee that only very few of them become active, while most others become inactive and will not be affecting the final result. Thus, less variables are needed compared to other proposed formulations.

For a node $v \in V$ we denote its incoming and outgoing edges by $\delta^{\text{in}}, \delta^{\text{out}}$ and to further simplify the notation $\varphi(r) = \{r^+, r^-\}$ is used to represent the set of both actions of r.

The depot node $(0, \dots, 0)$ is abbreviated to $\mathbf{0}$ and the set V_{r^+} stands for all events in V, that correspond to a pick-up action of r.

$$\min \quad \sum_{a \in A} t_a x_a + W \cdot \sum_{r \in R} (1 - p_r) \tag{2.2a}$$

s.t.

$$\sum_{a \in \delta^{\text{in}}(v)} x_a - \sum_{a' \in \delta^{\text{out}}(v)} x_{a'} = 0, \qquad \forall v \in V, \tag{2.2b}$$

$$\sum_{a \in \delta^{\text{in}}(v) \colon v \in V_{r^+}} x_a = p_r, \qquad \forall r \in R, \tag{2.2c}$$

$$\sum_{a \in \delta^{\text{out}}(\mathbf{0})} x_a \leq |K|, \tag{2.2d}$$

$$\begin{aligned} B_{g^*} \geq B_{r^*} + \pi + t(o_{r^*}, o_{g^*}) \qquad & \forall r^*, g^* \in \{\varphi(r), \varphi(g) \mid r, g \in R\} \colon , \\ - M(1 - \sum_{\substack{(v,w) \in A \colon \\ v_1 = r^* \wedge w_1 = g^*}} x_{(v,w)}), \qquad & ((r^*, \dots), (g^*, \dots)) \in A, \end{aligned} \tag{2.2e}$$

$$B_{r^+} \geq e_0 + t(o_{\mathbf{0}}, o_{r^+}) \sum_{(\mathbf{0},w) \in A \colon w_1 = r^+} x_{(\mathbf{0},w)}, \quad \forall (\mathbf{0}, w) \in A, \tag{2.2f}$$

$$e_{r^*} \leq B_{r^*} \leq \ell_{r^*}, \qquad \forall r^* \in \{\varphi(r) \mid r \in R\}, \tag{2.2g}$$

$$B_{r^-} - B_{r^+} - \pi \leq \mathcal{L}_r, \qquad \forall r \in R, \tag{2.2h}$$

$$x_a \in \{0, 1\}, \qquad \forall a \in A, \tag{2.2i}$$

$$p_r \in \{0, 1\}, \qquad \forall r \in R \tag{2.2j}$$

The objective function in Equation (2.2a) was discussed before in Section 1.2. We make use of a penalty W big enough, so that the number of accepted requests is maximized first and the secondary goal is to find a plan of minimal duration. The typical flow constraint can be seen in Equation (2.2b). This needs to be upheld to make sure we receive a path for every vehicle. Next, Equation (2.2c) is employed to guarantee that a request r is picked up by a vehicle if and only if $p_r = 1$. In Equation (2.2d), we assure that exactly $|K|$ paths start at depot node $\mathbf{0}$, which will also end there because of the flow constraint.

For adjacent actions in the event graph, we use Equation (2.2e) to take travel duration into account for the B variables. The inequality is disabled by a big-M factor, if there is no edge activated between the two actions. Additionally, Equation (2.2f) guarantees the correctness of the departure times for the very first actions on the paths. Finally, the time windows of each action are ensured by Equation (2.2g) and the maximum travel time in Equation (2.2h).

Due to the fact, that we aim for a lexicographic objective function, it is of importance to find a W big enough, so that the number of accepted users in maximized first. Still, we do not want to set this value arbitrarily large, as this could hurt the performance of the model. With the following Lemma 2.1 we conclude a relatively small upper bound for W, that we will also give a short proof for.

Lemma 2.1. *For $W = 2NR + 1$, where N is the total time needed to travel along each line from start to end, the minimal solution OPT of our MILP model has the maximum number of accepted requests.*

Proof. By contradiction:
Assume there exists a different solution ALT with more accepted requests. We know that the difference in penalty is at least $2NR+1$. However, we also know that $OPT \leq ALT$. This means, that the difference in traveled time has to be at least $2NR + 1$.

$$\begin{aligned} & ALT \geq OPT \\ \Leftrightarrow & \sum_{x \in A} t_a x_a^{ALT} \geq \sum_{x \in A} t_a x_a^{OPT} + 2NR + 1 > 2NR + 1 \end{aligned}$$

This implies that the overall duration traveled in ALT is at least $2NR + 1$. However, any plan that minimizes the traveled duration cannot be longer than $2NR$. In this worst-case scenario every request in R needs to travel through the entire network, using all of the lines. So, for every line a bus has to travel along the whole line to get into position to pick-up the request and then drive all the way back for delivery. Leading to an upper bound of $2NR$ for the travel duration. By exceeding the upper bound we reached a contradiction. Thus, there can not be a solution with more accepted users than in OPT. □

2.2. Adaptations

In the liDARPT multiple buses on different lines have to be coordinated at the same time, while users of requests are able to transfer from one bus to another. Furthermore, due to cycles in the network there can be different *route options* for a single request. We interpret a route option as one possible path through the network along one or more lines from pick-up to drop-off location of the request.

The various route options have to be found within a separate preprocessing step. As in larger networks the number of possibilities can be quite large, we can make the compromise of only exploring some variations. For example, we identify the shortest possible routing option for a request and then only explore variations with at most one extra transfer. For a request $r \in R$, we consider a set of indexes $\Omega(r) = \{1, 2, \dots\}$ and every r_i with $i \in \Omega(r)$ as a route option for r. Additionally, each routing option r_i consists of multiple *splits* $r_{i,j}$ with $j \in \omega(r_i)$. A split describes the users traveling along a specific bus line, before arriving at the destination or switching to another bus line afterwards. Accordingly, every split $r_{i,j}$ possesses a pick-up $r_{i,j}^+$ and drop-off action $r_{i,j}^-$ and corresponding locations $o_{r_{i,j}^+}$, $o_{r_{i,j}^-}$.

An example for these concepts is pictured in Figure 2.3. Three different bus lines are shown, each drawn in their own color, that form a network with a cycle. Furthermore, a request r and its pick-up o_{r^+} and drop-off location o_r^- are marked. There are two

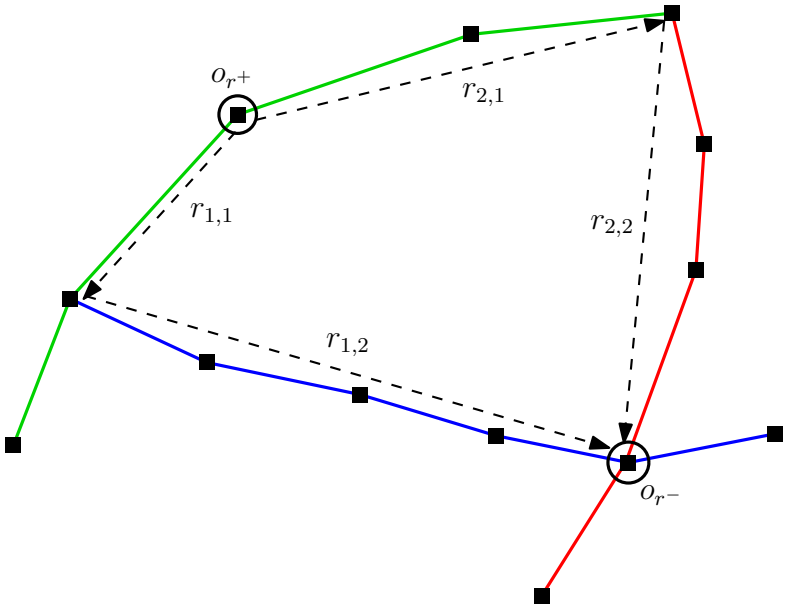


Fig. 2.3.: Example of a network with circles and a single request r with two route options.

sensible ways for the users of request r to travel to their destination. Both route options consist of two splits in this example, as the users would have to transfer buses once in each option. It should be noted, that multiple route options can also occur in the case of overlapping lines. Meaning that, multiple lines run along two or more stops. So if the users of the request would have to transfer between these lines, there are now different options for transfer locations.

We declare $R_\Omega = \{r_{i,j} \mid r \in R \wedge i \in \Omega(r) \wedge j \in \omega(r_i)\}$ as the set of all splits of requests. Moreover, the set $R^*_\Omega = \{r^*_{i,j} \in \varphi(r_{i,j}) \mid r_{i,j} \in R_\Omega\}$ entails all pick-up and drop-off actions of splits of requests.

Using this expanded notation is necessary, but can also grow the corresponding MILP model quite extensively. Here, we need a variable for the timing of every splits pick-up $B_{r^+_{i,j}}$ and drop-off $B_{r^-_{i,j}}$ for every split $r_{i,j} \in R_\Omega$. Besides, additional decision variables $x_{r_i} \geq 0$ for all $r \in R$ and $i \in \Omega(r)$ are needed for the model to select one of the given route options by setting its value to one, while the other option variables are kept at zero. The variables can be declared as binary or relaxed as a continuous variable, without constraining the result, see Lemma 2.2. Additionally, we simplify the notation by using the inverse function $\sigma^{-1}(l)$ to find the set of buses that traverse line $l \in L$.

$$\min \quad \sum_{a \in A} t_a x_a + W \cdot \sum_{r \in R} (1 - p_r) \tag{2.3a}$$

s.t.

$$\sum_{a \in \delta^{\text{in}}(v)} x_a - \sum_{a' \in \delta^{\text{out}}(v)} x_{a'} = 0, \qquad \forall v \in V, \tag{2.3b}$$

$$\sum_{a \in \delta^{\text{in}}(v) \colon v \in V_{r^+_{i,j}}} x_a \geq x_{r_i}, \qquad \forall r_{i,j} \in R_\Omega, \tag{2.3c}$$

$$\sum_{a \in \delta^{\text{out}}(\mathbf{0})} x_a \leq |\sigma^{-1}(l)| \qquad \forall l \in L, \tag{2.3d}$$

$$\begin{aligned} B_{g^*_{m,n}} \geq B_{r^*_{i,j}} + \pi + t(o_{r^*_{i,j}}, o_{g^*_{m,n}}) \qquad & \forall r^*_{i,j}, g^*_{m,n} \in R^*_\Omega \colon , \\ - M(1 - \sum_{\substack{(v,w) \in A \colon \\ v_1 = r^*_{i,j} \wedge w_1 = g^*_{m,n}}} x_{(u,w)}), \qquad & ((r^*_{i,j}, \dots), (g^*_{m,n}, \dots)) \in A, \end{aligned} \tag{2.3e}$$

$$B_{r^+_{i,1}} \geq e_0 + t(o_{\mathbf{0}}, o_{r^+_{i,1}}) \sum_{(\mathbf{0},w) \in A \colon w_1 = r^+_{i,1}} x_{(\mathbf{0},w)}, \quad \forall r_{i,1} \in R_\Omega, \tag{2.3f}$$

$$e_{r^*_{i,j}} \leq B_{r^*_{i,j}} \leq \ell_{r^*_{i,j}}, \qquad \forall r^*_{i,j} \in R^*_\Omega, \tag{2.3g}$$

$$B_{r^-_{i,|\omega(r_i)|}} - B_{r^+_{i,1}} - \pi \leq \mathcal{L}_r, \qquad \forall r \in R, \forall i \in \Omega(r), \tag{2.3h}$$

$$B_{r^+_{i,j}} \geq B_{r^-_{i,j-1}}, \qquad \forall r_{i,j} \in R_\Omega \colon j \neq 1, \tag{2.3i}$$

$$\sum_{i \in \Omega(r)} x_{r_i} = p_r, \qquad \forall r \in R, \tag{2.3j}$$

$$x_{r_i} \geq 0 \qquad \forall r \in R, \forall i \in \Omega(r), \tag{2.3k}$$

$$x_a \in \{0, 1\}, \qquad \forall a \in A, \tag{2.3l}$$

$$p_r \in \{0, 1\}, \qquad \forall r \in R \tag{2.3m}$$

The majority of constraints as well as the objective function can be found in the model formulation of the liDARP, see Section 2.1. The main difference here is that most constraints for the liDARPT are formed over splits, while before it was sufficient to declare them for requests. One of the constraints with bigger changes is Equation (2.3c). It is used to ensure that all splits of the selected route option are actually picked up. The "greater equal" sign is used instead of equality, because some splits and their events are part of multiple route options. This way, the model does not become infeasible whenever these are selected. Because in the liDARPT multiple lines, with different depots are considered, Equation (2.2d) has to be augmented, by defining it separately for each line and adjusting the right-hand side to match the number of busses, see Equation (2.3d).

If we would not enforce a specific order for pick-up events, as we have noted in the beginning of Chapter 2, our formulation of Equation (2.3e) could lead to inconsistencies. When a request is picked up at its first stop with other requests, the end of its time window could be disregarded. This is due to the fact, that $B_{r^+_{i,1}}$ is not necessarily the

time of departure of the bus, instead we might have to wait for the other requests that are picked up afterwards. By ordering events so that those which need to leave later are handled first, the time windows of subsequent events fit within those of previously handled events. Thus, we cannot exceed the time window for any event.

Furthermore, in Equation (2.3h), the maximum travel time is assured by checking the difference of the last split's drop-off time and the pick-up time of the first split for a request. Then, Equation (2.3i) guarantees the timing for all subsequent splits, so that no user is picked up before they even arrive at the location. Finally, in Equation (2.3j), the route option variables of a request r are set into relation to the decision variable p_r. This ensures that exactly one route option is selected if and only if $p_r = 1$.

Lemma 2.2. *All variables x_{r_i} for $r \in R$ and $i \in \Omega(r)$ from the liDARPT model can be restricted to binary values without loss of optimality.*

Proof. Assume we have an optimal solution of the model.

For any $r \in R$ we consider the following cases:

1. case $p_r = 0$:

   $\Rightarrow x_{r_i} = 0 \quad \forall i \in \Omega(r)$, because of Equation (2.3c).

2. case $p_r = 1$:
   1. case $\exists x_{r_i} = 1$:

      $\Rightarrow x_{r_j} = 0 \quad \forall j \in \Omega(r), j \neq i$

   2. case $\exists x_{r_i} : 0 < x_{r_i} < 1$: In this case, we could transform the solution to one with equal objective value and binary variables. We simply set all $0 < x_{r_j} < 1$ to 0, except one that should be equal to 1. This would obviously satisfy Equation (2.3j). Similarly, Equation (2.3c) is also still upheld, because the x_a variables are binary themselves. So their sum was already required to be 1 before. The variables occur in no other constraint, so all of them are fulfilled and the objective value is the same.

□

3. Software and Implementation

In this chapter we will take a closer look at our implemented application for solving liDARPT instances optimally. The code is written in Python 3.10 and uses the IBM Cplex Developer Edition for solving the ensuing MILP. It is publicly available on GitHub[1]. The software is implemented in a way to enable simple extension, like observing the liDARPT and our solution approach in the dynamic setting or simply employing different solving algorithms. Thus, some of the data is redundant in the static case and some interfaces could be implemented in a different way.

We will first look at the necessary input data and the format our implementation expects in Section 3.1. In the following Section 3.2, we will go in detail on the different modules and dependencies to gain an overview of the application flow. Afterwards, some of the more interesting and complex algorithms used for preprocessing, especially for generating the events and the corresponding graph are detailed in Section 3.3.

3.1. Input Data

As the goal of this thesis mainly concerns the feasibility of the liDARPT, we do not use actual locations, roads and requests for our instances. Instead stops are referenced as points on a 2D plane and the euclidean distance and a fixed traveling speed is presumed, when moving in a straight line from point to point.

The input to the software is structured as follows. The program directly receives the path to a configuration file, which entails many hyperparameters. Among them are the service time π, unit distance, average speed and time window length for pick-up. Also, the number of extra transfers, used for reducing the amount of considered route options for requests is noted in the file. Most importantly, the request and network files are linked from there.

The network file includes a set of all stops with id and two-dimensional coordinates. Furthermore, lines with their ids are noted, each consist of a list of stop-ids. A line also receives a depot coordinate and a capacity of users. We do not define the capacity per bus, to simplify the MILP model. If we were to define the capacity per bus, then we would need to create a separate event graph with different events for each bus on a line, as a different capacity can lead to new events. Finally, all buses are listed with unique ids and a corresponding line id.

Similarly, in the request file, each request is identified by an id. Additionally, the ids of their pick-up and drop-off stops are documented and the amount of users tied to each request. Finally, the earliest start time of the user is recorded and, for observing the

[1] `https://github.com/barjon0/liDARP`

problem in the dynamic case, the file entails the time a request is first registered by the system.

It should be noted that the request file does not foresee a fixed latest arrival time. Instead, the application finds the shortest path and traveling duration for a request, then calculates the maximum travel time by a function stated in the configuration file. By adding the time window for pick-up, also noted in the configuration file, and the maximum travel time to the earliest start time, we implicitly compute the latest arrival time of a request.

In our tests we used $1.2 \cdot \log_{1.2}(t_r^*)$ as the function for the maximum delay, considering a shortest travel time t_r^*. The maximum travel time $\mathcal{L}_r$ is then derived from the addition of the shortest travel time and the maximum delay. Subsequently, we compute the latest arrival time ℓ_{r^-} by adding the fixed length of the pick-up time window and maximum travel time $\mathcal{L}_r$ to the earliest start time e_{r^+}. The function tolerates roughly double the shortest time as delay for short requests of less than 5 minutes. However, for requests of long to very long duration around 30 to 40 extra minutes are allowed.

3.2. Code Structure

First, we will get a better understanding for the implementation by discussing the modeling classes. They are used to structure the problem and categorize the input data, as well as the output. Also, some classes are important for the intermediate steps in preprocessing. One can gain a basic overview in Figure 3.1.

In the upper right corner of Figure 3.1 we can see the basic modeling classes used for the networks. Every bus travels on a line, while a line contains a list of stops and each stop possesses a coordinate. Each bus also has a unique start and end time to specify a time window in which the bus is on the road. Furthermore, the capacity of a bus is decided by the line it drives on, as *Line* contains that parameter. The line also specifies the depot, from which its buses start its tour and return to.

To the left, the modeling of requests is pictured. In the implementation, we use an abstract class *AbstractRequest* to differentiate between actual *Requests* and the segmented *SplitRequests* that do have a lot of the same characteristics. The abstract class contains the basic information needed for a request, as well as implicitly calculated time windows for pick-up and drop-off. Moreover, a request receives a time at which it was registered. We also state the number of transfers, needed for the shortest path of a request. Finally, every request gets a dictionary for all different route options and their splits, which is calculated by the built in function *splitUpRequest()*. The *SplitRequests* get a unique id and the line they are meant to travel along.

For structuring the output of the program, the *Route* class is used. It specifies a tour for a defined bus, consisting of a list of *RouteStops*. This describes a stop at a certain location, with some arrival and departure time, and a list of requests picked up and dropped off. We can describe plans for all buses with these classes.

As we already discussed, our approach entails building an event-based MILP. For that, we also have to model the events. First an abstract class *Event* specifies the core variables

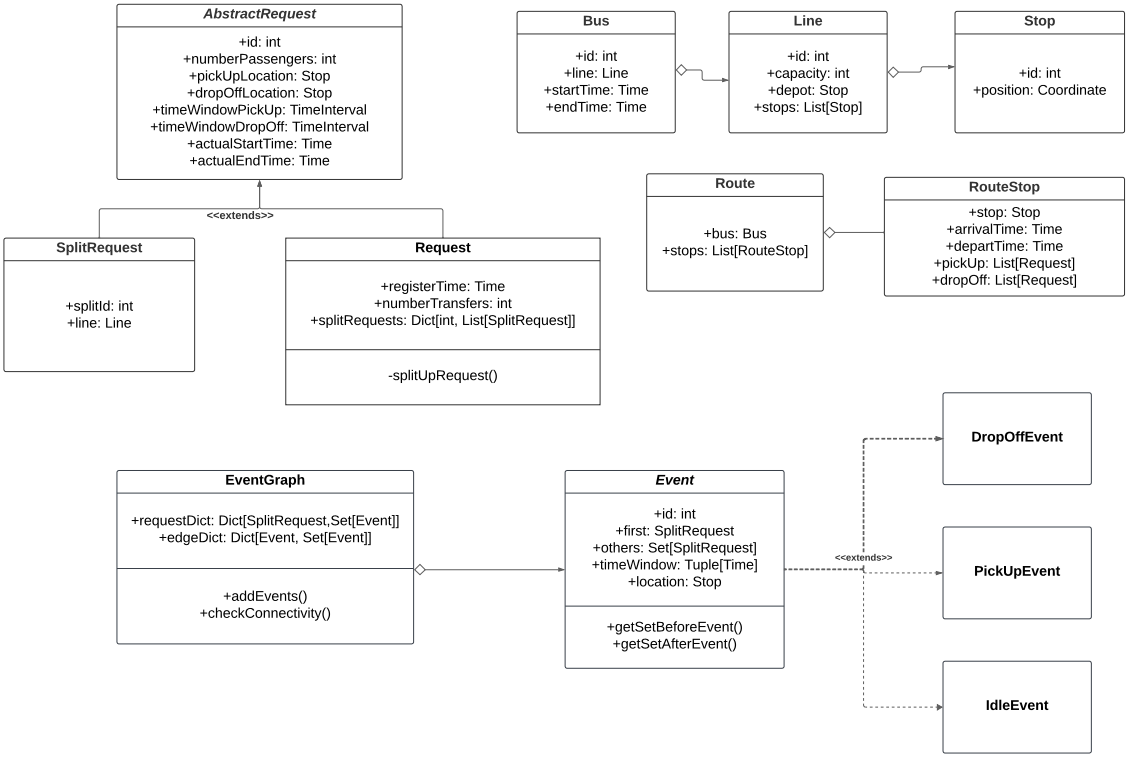


Fig. 3.1.: UML Diagram of modeling classes.

required for such a data type. For an event, we need to know the split in question, that is either picked up or dropped off. Furthermore, all other splits also in the vehicle need to be specified. A time window for an event is inferred within our program to eliminate some events and edges in the graph. This abstract class is extended by the three possible types *PickUpEvent*, *DropOffEvent*, and *IdleEvent*. Their behavior mostly differs in how they implement the class methods *getSetBeforeEvent()* and *getSetAfterEvent()*, which return the set of requests in the vehicle before or after the event. This will become important for building the event graph. The graph structure in *EventGraph* is modeled by two dictionaries. One gets the neighboring events for a given event, the other groups events by their important split. The event graph can be build incremently by its *addEvents()* method, that adds a list of events to the graph and infers the necessary edges. We check that the graph is valid and fulfills some relevant properties with *checkConnectivity()*.

The modules used for handling the workflow and containing most algorithms are pictured in Figure 3.2. First, we can acknowledge the *DataGenerator* component on the left side. Its responsibility is generating random user requests for given bus networks and a specified number of users, which we use for testing. We will detail its inner workings in Chapter 4. As discussed before, the program receives a configuration file, that specifies necessary hyperparameters, as well as bus network and request file.

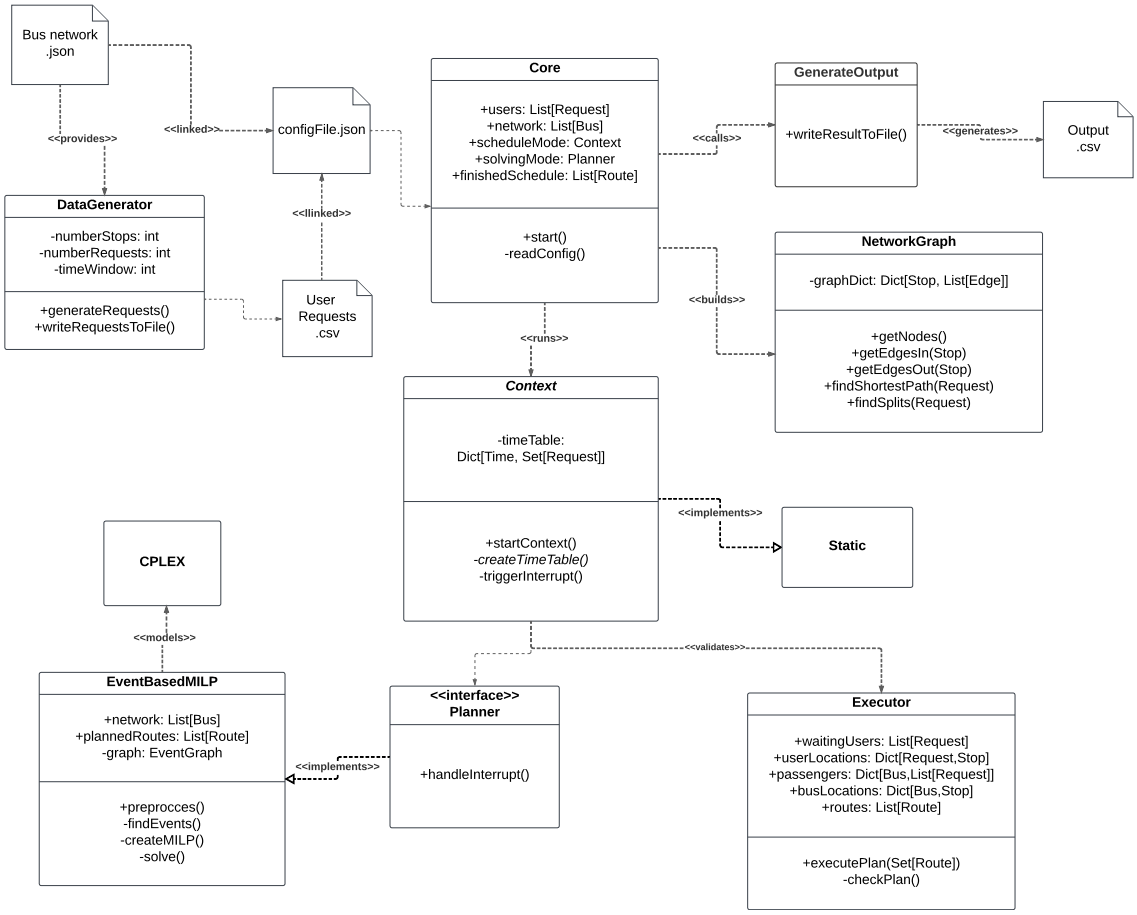


Fig. 3.2.: UML Diagram of functional workflow modules.

The *Core* module reads in the data and delegates control flow. To start, it instantiates a *NetworkGraph* object, which is a graph of the bus network. This is then used to find all feasible route options and splits for requests, see Section 3.3 We store this information with the basic requests. The module *GenerateOutput* is called at the very end, to write the resulting plan and some key figures into an external file. The remaining components of the program are *Context*, *Planner*, and *Executor*. We decided on this implementation, as it enables us to generalize the code for use in an dynamic environment. In the static case the context simply instantiates a class implementing the *Planner* interface and waits for a valid schedule. Then, forwards this result to the *Executor*, which validates the plan by stepwise execution.

If we were to extend the program to the dynamic case, the *Context* module handles control flow, as it hands currently registered requests to the *Planner*, waits for a result or interrupts after a while to adhere to its time table. Then, the *Executor* is not only used to validate a plan, but also to find the current location of buses and request users at

a point in time. This way, the *Planner* is updated on the situation, when new requests are registered and a new plan needs to be generated.

Different planning algorithms are feasible for this problem and could extend our *Planner* interface, like heuristics or other optimization approaches. However, we focus on our event-based MILP method. The corresponding class *EventBasedMILP* receives a list of buses and requests, as well as information on their current locations. With this data, it starts preprocessing. A procedure that entails finding all possible events, that can happen on a line. It then uses an *EventGraph* instance to structure the events, so that the MILP model can be created. We use the CPLEX Python API for the process of creating a model instance and then solving it via a local CPLEX installation. Afterwards, the CPLEX output variables need to be translated into a readable plan. In our case, a number of *Route* objects to specify the paths each bus takes.

3.3. Preprocessing Algorithms

Preprocessing the input is a critical process in our program to create a simpler event-based MILP model. So in this section we will take another look at the most important steps. One of the first things we need to do is finding the shortest route for a request within the network. For one, we need this information to implicitly determine the maximum travel time we allow for this request. Furthermore, as discussed in Section 3.1, the number of transfers for this path is used in determining the allowed number of transfers in any other route option. The Dijkstra-algorithm is a well-known approach of finding such a path. Thus, we employed this method on an instance of our *NetworkGraph*, to find the shortest path for a request.

Subsequently, for finding all possible route options of a request we decided on a depth-first search. The pseudocode in Algorithm 1 illustrates our approach that mirrors the classic recursive depth-first search with backtracking to find all paths in a graph, with some additional stop criteria.

The algorithm receives the current location, elapsed time, and the list of stops explored to get to the current location. It generates a set of all possible paths through the line network from pick-up to the drop-off location of a request as output.

For that purpose, the method can also reference to a *NetworkGraph* object, named *network*. This construct only entails transfer stops as vertices, as all other stops do not allow for us to make any kind of decision, so they can be omitted here. For a request, its pick-up and drop-off locations are added to the graph and its edges are inserted accordingly. Overall, the *network* object enables the algorithm to discover neighboring stops easily, so it can traverse through the network quickly. It also finds the duration of travel between two points.

When the method is first called for a request $r \in R$, it receives the pick-up location, current time of zero and an empty set, because no stops have been explored yet. As this is a recursive method, it is then called multiple times, for every explorable transfer stop, along the way to the destination.

Due to its recursive nature, the method begins with a stop condition Line 1, returning

Algorithm 1: recDFS($currStop, currTime, explStops$)

```
1  if currTime > maxTime ∨ size(explStops) > maxNumberTransfers then
2      return set()
3  explStops.append(currStop)
4  if currStop = target then
5      return set(list(currStop))
6  else
7      optionSet ← set()
8      for successor ∈ (network.getNeighbours(currStop) \ explStops) do
9          duration = network.getDuration(currStop, successor)
10         foundOptions = recDFS(successor, duration + currTime,
               explStops.copy())
11         for option ∈ foundOption do
12             option = currStop + option
13         optionSet.addAll(foundOptions)
14     return optionSet
```

an empty set if the current time exceeds the requests' maximum travel time or the allowed number of transfers is surpassed. After having made sure that none of these conditions are hurt, the current location is added to the set of explored stops. Should the current location also be the desired drop-off location, another stop condition is triggered and a new set, with a list of the current stop is returned in Line 5, because it describes the only valid path to the pick-up location. Otherwise, we prepare a recursive call for every neighbor of the current stop in the network that has not been visited yet. Thus, circular routes are excluded. After the duration for a route segment is computed, we find all route options from the successors' view to arrive at the drop-off location in Line 10. We convert this solution to a solution for the current stop, by simply prepending the current stop to all of the found route options, see Line 12. Then, after the altered options for every neighbor are added together into one set, we can return this result to the calling method.

Because we recurse on all neighbors of the current location, that were not already traversed up to this stop, we are guaranteed to find all paths without circles between pick-up and drop-off location in the network.

One of the central aspects of this thesis is the use of an event-based structure and building the corresponding event graph. This step is very important for our method. We aim to exclude as many unnecessary events and edges as possible, to shrink the size of our MILP model and increase the solvers performance. However, the preprocessing itself needs to be efficient and should not surpass the time required to solve the model.

The overall workflow for generating the event graph in our implementation is depicted in Algorithm 2. The algorithm receives the set of all splits R_Ω as input and creates the necessary events and builds the event graph.

Algorithm 2: makeEventGraph(R_Ω)

```
1  for line ∈ network.lines do
2      events = set()
3      for dir ∈ line.direction do
4          candidateMapLocal = map()
5          candidateMapTime = map()
6          R'_Ω ← r_{i,j} ∈ R_Ω where r is on line and traveling in direction dir
7          candidateMapLocal ← sweepLineLocal(R'_Ω)
8          candidateMapTime ← sweepLineTime(R'_Ω)
9          candidateMap = map()
10         for r_{i,j} ∈ R'_Ω do
11             candidateMap[r_{i,j}].pickUp = candidateMapLocal[r_{i,j}].pickUp ∩
                candidateMapTime[r_{i,j}].pickUp
12             candidateMap[r_{i,j}].dropOff = candidateMapLocal[r_{i,j}].dropOff ∩
                candidateMapTime[r_{i,j}].dropOff)
13         for r_{i,j} ∈ R'_Ω do
14             for permutation ∈ buildPermutations(candiateMap[r_{i,j}].pickUp) do
15                 events.add(PickUpEvent(r_{i,j}, permutation))
16             for permutation ∈ buildPermutations(candiateMap[r_{i,j}].dropOff) do
17                 events.add(DropOffEvent(r_{i,j}, permutation))
18     hashList = list()
19     for event ∈ events do
20         hashList.add(hash(event.setBeforeEvent), event)
21         hashList.add(hash(event.setAfterEvent), event)
22     eventGraph.addNodes(events)
23     eventGraph.addEdges(hashList)
```

The algorithm is called with the set of all split requests R_Ω. Splits that travel along different lines can be processed independently of each other, as they can never be part of the same event or have adjacent events. We loop over all lines in the network. As the first task is to find all possible events, we can also loop over both directions in which a bus can travel on the line. This is due to the fact that our busses never change direction while passengers are inside the vehicle. Thus, there can never be two requests that want to travel in opposite directions in the vehicle at the same time.

For every split request, we want to find all other splits that may ride together in the vehicle. Out of all other splits R'_Ω that travel on the same line and in the same direction, we filter the corresponding candidates in two ways. First, we observe the interval of stops

between the splits' pick-up and drop-off locations and look for possible overlaps. This process is done once for all splits in R'_Ω in the *sweepLineLocal()* method and the results are written into *candidateMapLocal*, see Line 7. This map now contains, for every split, all splits that might be in the vehicle when the pick-up occurs and similarly all candidates for the drop-off. We filter out all splits belonging to the same request.

Subsequently, we build all possible candidates for pick-up and drop-off, based on the split requests' time windows. These are inferred from the requests' overall time windows and minimum durations of every split. So we get a earliest and latest start, as well as end times, for every split. With these two intervals, we call the slightly modified *sweepLineTime()* method and collect its result in *candidateMapTime*.

Now that we have filtered the candidates based on their locality and timing, in Line 11 we combine both results, by computing the intersection for every split and action type into *candidateMap*. From Line 13 onward, the *buildPermutations()* subroutine receives a set of splits as input and generates all possible set permutations, up to a size of c_l. The method helps with further removing events, by generating the permutations incrementally and always checking if it is feasible for the splits to share a vehicle, without any of them missing their time windows. We do not explore any supersets further if this condition is not met. Overall, the method allows us to build all events for pick-up and drop-off of a split, now that we have all possible combinations of candidates for each action of a split.

After all events are created, we need to find all directed edges between events for our graph. Two conditions have to be met for two events to be adjacent. First, the set of users in the vehicle after the predecessing event has to be the same as the set of users before the successor is completed. Second, the events have to match up in regards of timing. To focus on the first condition, instead of checking every pair individually, we use a hash list. For every event, we get the set of users in the vehicle before the event and the set of users after the event. For both sets, we compute its hash and store our event at that hash in the list, we also need to mark if the event was placed there due to the before or after set. We now go through the hash list and, at every index with multiple events, we add edges from events placed there by their after set to events placed by before sets.

This process is conducted in the *addEdges()* method of the event graph. Before adding the edge we check that there are no violations in regards of the timing between the events. This can be achieved by inferring time windows for every event from the time windows of the corresponding splits. If it is impossible to arrive at the successor event in time, after starting at the predecessing event within its time window, then no edge is created.

4. Test Experiments

This chapter concerns itself with evaluating the capabilities of the event-based model and the effectiveness of the liDARPT approach. We achieve this by running our implementation with multiple test instances and comparing the results. The focus of the tests lies on three aspects, specifically.

First, we want to investigate the effects of the single, lexicographically weighted objective function, we proposed in Section 2.2, by comparing it to a second approach. In this alternative the model is solved in two steps with different objective functions. The overall question being if optimizing the number of accepted requests and traveled distance at the same time is too conflicting for the CPLEX Optimizer or leads to faster computation times, see Section 4.2.1.

Furthermore, we examine the feasibility of the liDARPT approach by analyzing some of the key performance indicators (KPIs) in Section 4.2.2. The following Section 4.2.3 is concerned with determining the key factors that influence the complexity and scalability of the event-based MILP.

Finally, in Section 4.2.4 the comparison of the liDARPT to the closely related DARP becomes of interest. We want to see how well our approach works in relation to the DARP, where all vehicles can roam freely. For this undertaking, we translated our test instances to fit the software implementation from [GKP+24] and ran the tests again in a static DARP setting with their basic event-based model.

4.1. Test Data

Before we expand on the results of our tests, we will provide an overview of our instances and the experimental setup. We already discussed the required input format in Section 3.1. The networks consist of a number of two-dimensional points on the plane that form the stops and are chained together to lines. For a more practice-oriented approach, the network instances are modeled after existing long-distance bus lines by using publicly available network design diagrams as templates. These diagrams are not in one-to-one correspondence with actual locations and streets, as they provide a more clear and concise abstraction. They concern three different areas in the rural region around the city of Schweinfurt in Germany. The three areas are Schweinfurt-Schleerieth, Schweinfurt-Gerolzhofen and Karlstadt-Lohr. For each area, we varied the number of stops and lines to create multiple instances from one network topology. Overall, we chose seven network instances, detailed in Table 4.1. The table provides an overview over the number of lines and stops in each instance. The number of transfer stops that are served by multiple lines, allowing for requests to transfer between lines, is also listed.

Network Name	Lines	Stops	Transfer Stops	Transfer Stop Degree Sum	Network Length in km
Markt-Karl	3	15	3	8	109.3
Markt-Karl-Lohr	5	26	5	14	165.6
SW-Geo_2	2	21	2	5	76.5
SW-Geo_full	4	32	4	13	120.1
SW-Schlee_2	2	14	2	6	58.6
SW-Schlee_3	3	21	3	11	92.8
SW-Schlee_full	5	28	5	17	129.9

Tab. 4.1.: Network overview

For the fourth column, the degrees of all transfer stops are summed and noted per network instance. Intuitively, this is another parameter that could affect the complexity. The higher the value the more transfer options are possible, in theory. Finally, we can see the sum of the length of all lines in kilometers of our instances.

By examining many networks of different sizes and varying the number of lines and transfer stops, we not only make sure many different topologies are covered, but it will also allow us to compare the results across network instances and determine key factors that increase complexity.

The *Markt-Karl-Lohr* instance is exemplified in Figure 4.1. It is one of the more complicated topologies, as it consists of 26 stops and five different lines. Visualizations of the other networks can be viewed in the Appendix A from Figure A.1 to Figure A.6.

The number of requests is certainly one of the most important parameters responsible for the complexity of an instance. Thus, for every network, multiple sets of requests were created with varying cardinality. However, the length of the time span that encloses all of the requests' pick-up times is also of great importance. Simply said, finding a solution for serving 20 requests in one hour should be more difficult than doing so in five hours.

While the networks were modeled after real-world examples, the requests were generated randomly as no real-world data was available to us. To account for the length of the time span, we decided on generating requests for a length of three, six and nine hours. The number of requests was varied from 10 to 100 in steps of 10 requests. Thus, for every network, 10 different sets of requests are available for each time span length.

The requests were created by drawing two different stops, uniformly at random, out of all available stops in the network for every request. These were selected as pick-up and drop-off location. By splitting the entire time span into discrete five minute steps, we now sample an earliest pick-up time for a request with equal probability. Finally, the number of passengers connected to a request was chosen by assigning a 90% probability for picking a single passenger, 9% for two passengers and 1% for three passengers.

The following hyperparameters were used across all instances for testing. Every line in the network was assigned two buses. The capacity for passengers c_l of a bus on line l was set to six for all lines. The service time parameter π was fixed to two minutes, while we assumed a maximum delay for pick-up of a request to be 15 minutes. As

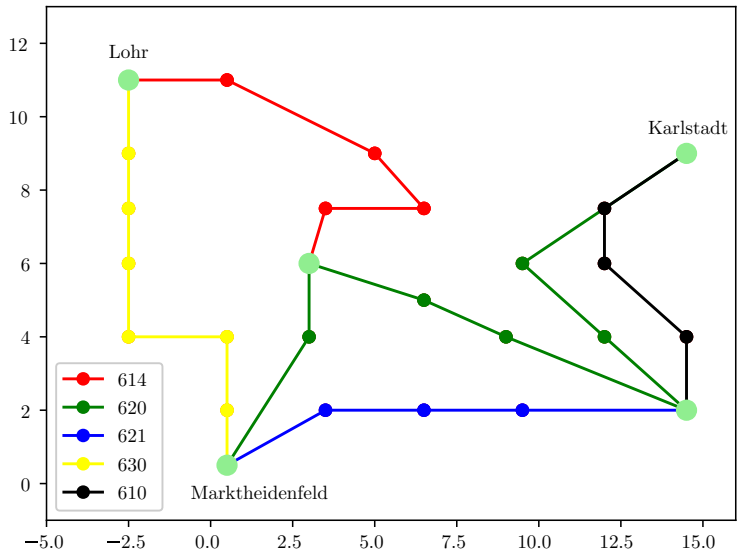


Fig. 4.1.: Visualization of the Markt-Karl-Lohr network instance in the 2D plane. Transfer stops are marked in light green.

stated in Section 3.1, the maximum delay after pick-up of a request was determined by $1.2 \cdot \log_{1.2}(t_r^*)$, where t_r^* is the shortest possible travel time of r in the network. At most one extra transfer is allowed for a request compared to the number of transfers in the shortest route. The traveling speed was assumed to be fixed to 65km/h for all instances, except for the bigger examples *SW-Geo_2* and *SW-Geo_full*, where it was set to 70km/h.

All of the tests were conducted on a 24 core Intel i9-13900F machine with 32GB RAM and operating system nixOS 24.11. The code was run through a Python 3.10 interpreter and the ensuing MILP model was solved by a local installation of the CPLEX Developer Edition 22.1.1. We limited the models solving time to 15 minutes to test under realistic conditions, where longer inference times are often infeasible.

4.2. Results

Before looking at our findings, we first define the important metrics that we will focus on when analyzing and comparing different results. The authors in [LLMS21] put together KPIs for Dial-a-Ride services. Among them is the *system efficiency*, which is computed as follows.

$$\text{systemEfficiency} = \frac{\text{kmBooked}}{\text{kmTraveled}} \tag{4.1}$$

Here, the term *kmBooked* describes the sum of kilometers the accepted requests would need if they were to travel to their destination directly, while *kmTraveled* is the overall kilometers driven by the buses in our system to service these requests. Thus, with a system efficiency of greater than 1 we would be saving kilometers compared to all requests driving their own car. Another important figure is the *vehicle utilization*, it helps us determine whether passengers are effectively grouped together. We define this figure slightly different than in [LLMS21] by inserting the total distance traveled into the denominator instead of the used distance.

$$\text{vehicleUtilization} = \frac{\text{accRequestKM}}{\text{kmTraveled}} \tag{4.2}$$

The term *accRequestKM* refers to the sum of the distance each request travels in the network. Even though the formula seems might seem complicated, its result is rather intuitive. If we attain a utilization of 1.0, it means that on average each bus carried one passenger at all times.

Furthermore, to investigate the user-friendliness of the system, we observe the acceptance rate of requests and average delay. By focusing on all of these figures, we assess the quality of solutions and evaluate instances against each other.

4.2.1. Objective Function

There are two objectives we want to achieve for the solutions in lexicographic order. The main one is to accept as many requests as possible, then the aim is to minimize the travel distance of buses. There are multiple options to implement this, we focus on two intuitive methods. We either combine both objectives into one weighted function as shown in Section 2.2, or we solve the model with multiple objective functions. For this, the model is solved once, where the number of accepted requests is maximized. This number is then fixed by adding a new constraint and the model is solved again, with a new objective function that minimizes travel distance.

Generally, both options have their upsides and downsides. With a single objective only one solve is required. However, large weights can sometimes lead to numerical instability. Furthermore, finding the optimal value with conflicting objectives can increase the difficulty for the solver. The multi-objective approach on the other hand does not encounter this issue, as the solver can focus on one objective at a time. Here, the main problem arises with a maximal computation time, where we have to decide how long the solver should spend on optimizing each of the objectives. As we selected a maximum solve time of 15 minutes for all tests, we decided on a 10 minute limit for the maximization of requests, while the rest is spent on the travel distance.

To identify what approach works better for our MILP model, we tested both options on the provided instances. For a time span length of three hours, the maximum number of requests we tested was 60, while for six hours up to 90 request instances were evaluated and in the nine hour span all request instances were tested. By limiting the number of requests, we filter out all instances with solutions that far exceed a MIP Gap of 20% in the single-objective approach.

The main question is how do the objective values of solutions in the two model versions compare. To explore this, we transformed the final solutions of the multi objective into the weighted objective function that is used for the single-objective approach. When both methods lead to optimal solutions the objective values should be equal. It will be interesting to see how the values compare for bigger instances, where the solver was unable to compute an optimal solution. For a fair comparison, we evaluate the relative difference of values for each instance. The results are depicted in Figure 4.2. As will be the case for most plots, on the x-axis we can see the density of requests. That is the number of requests divided by the length of the time span of each instance. The y-axis shows the relative difference computed as

$$\hat{\Delta}\text{objValue} = \frac{(\text{objVal}_{\text{Mult}} - \text{objVal}_{\text{Single}}) \cdot 100}{\text{objVal}_{\text{Mult}}}.$$

So, a positive value indicates that the single-objective approach has a lower objective value compared to the multi objective for this instance, meaning the solution is better here.

First, for small densities we expected both solutions to be equal. As our weights in the single objective should be high enough that a lexicographic order is induced. This is consistent with our results, because densities of less than eight, were solved to the optimum in both approaches and no difference in objective values were found. However, if we increase the density it would appear that most instances received a better solution in the single objective, with a difference of up to 40% in some cases. Even though there are some instances a little below the breakeven point, meaning that the multi-objective solutions were slightly better here. A big outlier is the instance *SW-Schlee_3* with 90 requests and six hour time span at density 15, with a relative difference of over 80%. This was one of the few cases were the multi-objective approach was unable to find an acceptable number of requests to transport within the 10 minutes, while the single objective got within 20% of the lower bound. Generally, we can see particularly high gaps for the instances of *SW-Schlee_3*, *SW-Schlee_full* and for other networks with many lines.

In the weighted objective function the weights enforce the lexicographic ordering between both objectives. Thus, the relative difference of objective values almost entirely consists of the difference in accepted requests. This is why the difference in accepted requests depicts the same pattern, see Figure 4.3.

However, we can observe the scale of the differences and see that for instances with a density of at least ten, in roughly 67% of examples more requests were accepted in the single-objective approach. Furthermore, for around 27% of instances with this density the difference exceeds 10%.

Next, we can focus on the second objective, which is the travel distance. We aim to assess if the five minutes spent on optimizing distance alone leads to more efficient routes. By simply investigating the overall travel distance, no fair comparison can be made, due to different numbers of accepted requests. Alternatively, the system efficiency normalizes the travel distance with the booked kilometers of accepted requests. The difference in system efficiency is plotted in Figure 4.4.

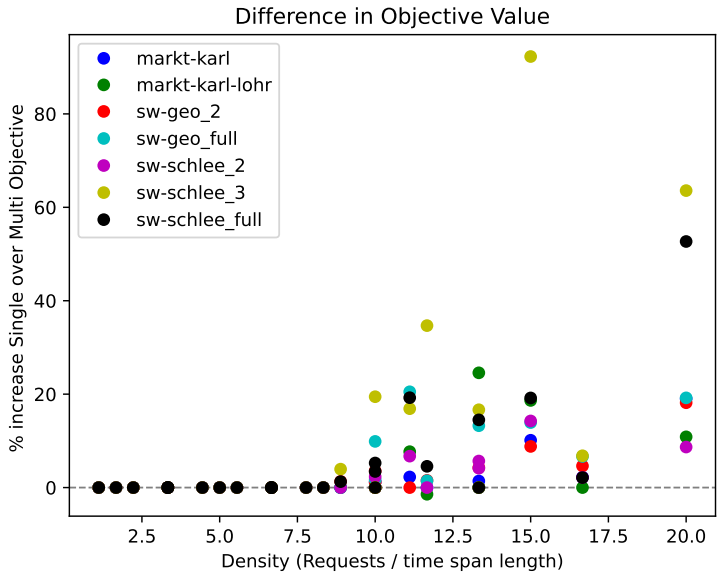


Fig. 4.2.: Relative difference in objective value per instance, when using single objective over multi objective.

Again, positive values correspond to instances where the single-objective approach outperformed its counterpart in this metric. Here, for many instances the system efficiency is better in the multi-objective solutions, as most markers are placed in the sub-zero region. Specifically, investigating instances with a density of at least ten, we see that roughly 21% of the examples are more efficient in the single-objective approach, opposed to 51% of instances performing better with multiple objectives.

If we observe the instances more closely, it appears that almost all the markers in the upper half belong to the bigger instances with more lines and transfer stops. At the same time, the networks that perform relatively well in the multi-objective approach are *SW-Schlee_2* and *SW-Geo_2*, both of which consist of only two lines.

Overall, the results show a trend of more efficient multi-objective solutions. This could could be an indicator, to spend less time on minimizing the distance, as the results here seem competitive to the other approach, and instead increasing the time for the first objective, where the multi objective is lacking in quality.

Finally, we shift our focus to the computation times in both approaches. For small to medium sized instances that can be solved optimally in 15 minutes, we want to find out if one of the options can preferred to the other. In Figure 4.5, the difference in computation times between both options is displayed. By subtracting the computation in the multi-objective approach from the single objective, a marker in the positive region indicates that we solved the instance faster with the latter.

The results show that for most instances the computation times in both options were very close, as for roughly 91% of all instances the difference was less than a minute. In

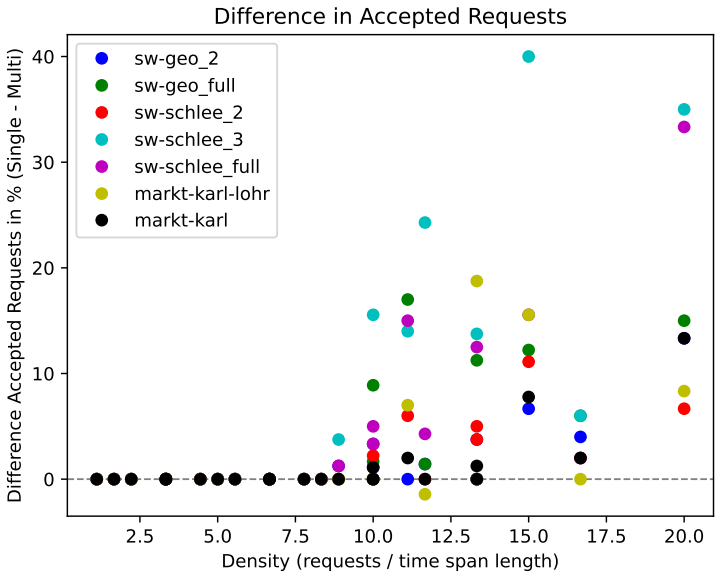


Fig. 4.3.: Relative difference in accepted requests per instance, when using single objective over multi objective.

25% of examples the single-objective approach was faster, while it was at least a second slower in 10% of instances. For the rest, the difference was less than a second. However, there are a few clear outliers in either direction. Especially in favor of the single-objective approach, as we can see a few instances that were solved 400 to 600 seconds faster here. For instances of the same network there appears to be great fluctuations from on to the other. For example, the *SW-Schlee_2* network has instances that are outliers in the positive and negative area. Even though it seems as if more instances were solved faster with the single objective, overall no clear trend can be determined from these results.

In conclusion, while the system efficiency is close and no clear general distinction between computation times can be made, out of both options the single-objective approach clearly outperformed its counterpart in accepted requests. As this is our main objective we will focus on this approach in the following sections in all of our tests.

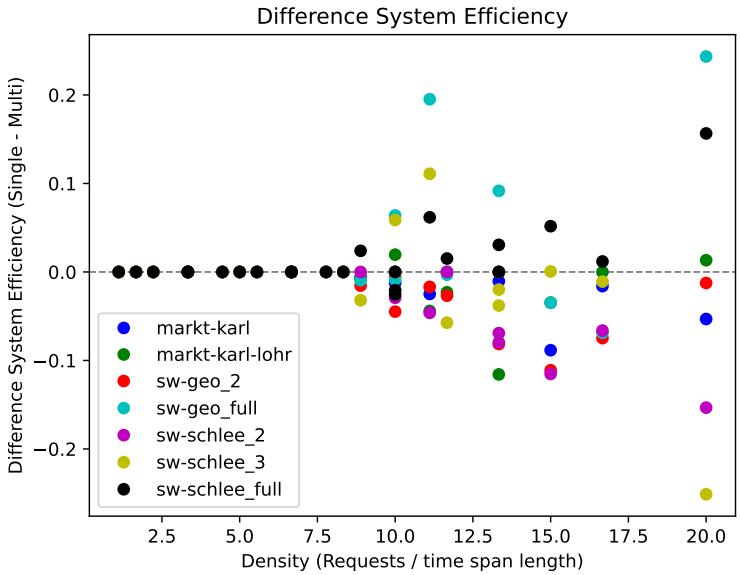


Fig. 4.4.: Difference in system efficiency per instance, between multi objective and single objective.

4.2.2. Solution Quality

In this section we will investigate the quality of solutions that were achieved with a single objective function, by plotting some of the key figures from each instance solution. Again, we plot the results for instances with up to 60 requests for a three hour span, 90 requests over six hours and 100 requests over nine hours, resulting in 175 instances overall. Increasing the number of requests for any of the time spans would lead to cases where the relative MIP gaps far exceed 20% after a 15 minute solve.

This is the first quality indicator we focus on, see Figure 4.6. The relative MIP gap is defined as

$$\frac{(\text{objVal} - \text{BestBound})}{\text{BestBound}} \cdot 100,$$

where *BestBound* refers to the greatest lower bound known to the solver when the program stops. Instead of the density, we split up the instances by their time span into three plots.

As expected, the gap tends to rise when the number of requests increases. Especially in the final step, where the number of requests rises to the maximum, we can see a big leap for most networks with relative MIP gaps of around 20% in four cases. Surprisingly, the MIP gap of the supposedly harder instances, like *SW-Schlee_full*, *SW-Geo_full* is relatively small across all time spans, exceeding a relative MIP gap of 5% only twice. On the contrary, the networks with only two lines performed worse. Out of the nine occasions a MIP gap of 10% was surpassed, four of those instances belonged to the networks *SW-*

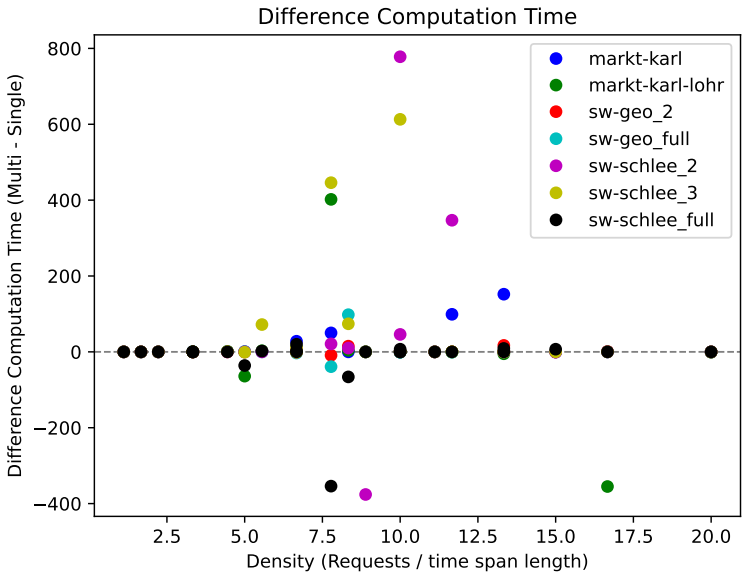


Fig. 4.5.: Difference in Computation Time in seconds, between multi objective and single objective.

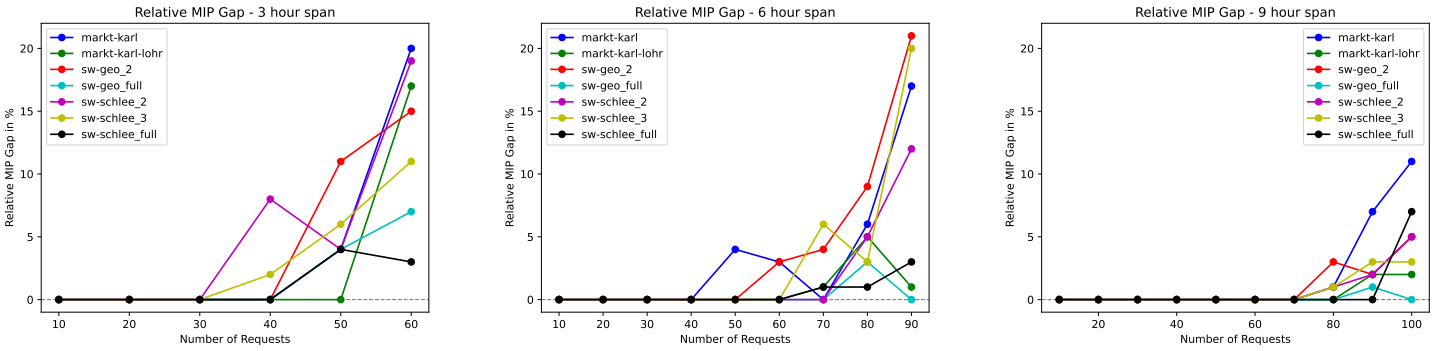


Fig. 4.6.: Relative MIP gap in percentage per instance.

Schlee_2 and *SW-Geo_2*. The other five occurred for networks with three lines. This effect might be explainable by a lack of busses. All networks receive two busses per line. The ratio of available seats to requests is significantly smaller for networks with two lines, which could lead to a more complicated problem.

The number of accepted requests is the main objective of our solution. Thus, the MIP gap highly correlates with this value. The ratio of accepted requests to the available number of requests is presented in Figure 4.7.

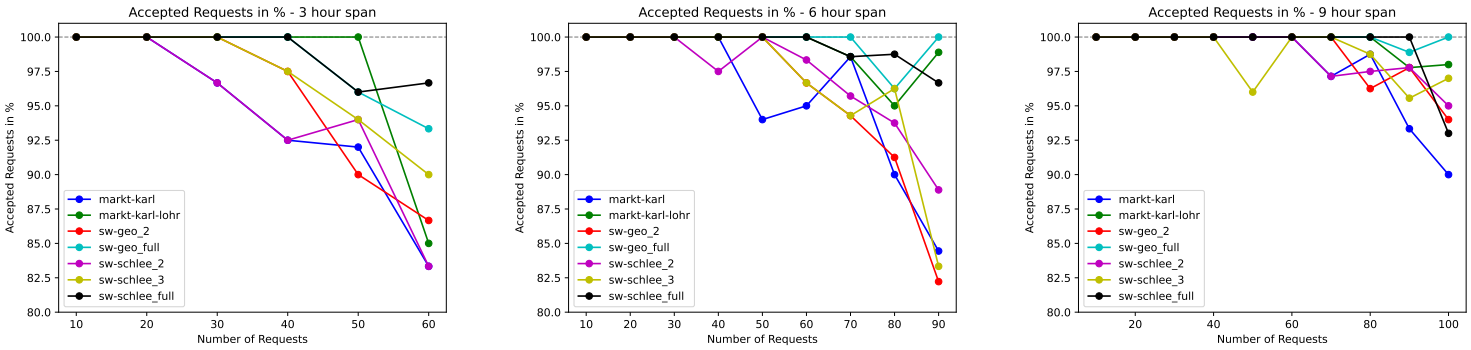


Fig. 4.7.: Accepted Requests in percentage per instance.

Generally, more than 90% of requests were accepted in all but six instances, where up to 18% of requests were denied. For the most part, the curves appear very similar to the ones from Figure 4.6, particularly for the maximum number of requests. However, some small deviations can be observed. Especially for the network *SW-Schlee_2*, for example with a three hour time span and 30 available requests, one request was denied even though the solve was shown to be optimal by the figure above. This also occurs for the six hour span and above all in the nine hour case. Due to the buses being restricted to their lines, it can happen that many requests appear along the same line, which could overload the two buses assigned to this line. Thus, they are unable to serve the requests within their respective time windows. Again, this is the case for networks with less buses, like *SW-Schlee_2*, in particular. Overall, in roughly 22% of all instances, the percentage of denied requests was higher than the relative MIP gap, meaning that the solver was able to improve the lower bound by definitively excluding requests in these cases. The most extreme case occurred for the instance *Markt-Karl* in a three hour span and 40 requests, it was solved optimally, but 7.5% of requests are denied, which corresponds to three requests. Among all optimal solutions, on average 99.7% of accepts are accepted.

Next, to assess the economic viability of the liDARPT approach, we investigate the system efficiency in Figure 4.8.

The breakeven value of 1 was only achieved once across all instances. However, due to the fact that the kilometers booked are computed by the straight line distance in our examples, it becomes harder to compare the two values than in a real-world scenario, where there are not as many shortcuts. The only occurrence of a value greater 1 was for the *SW-Schlee_2* instance with a three hour span and 60 requests. It should be noted,

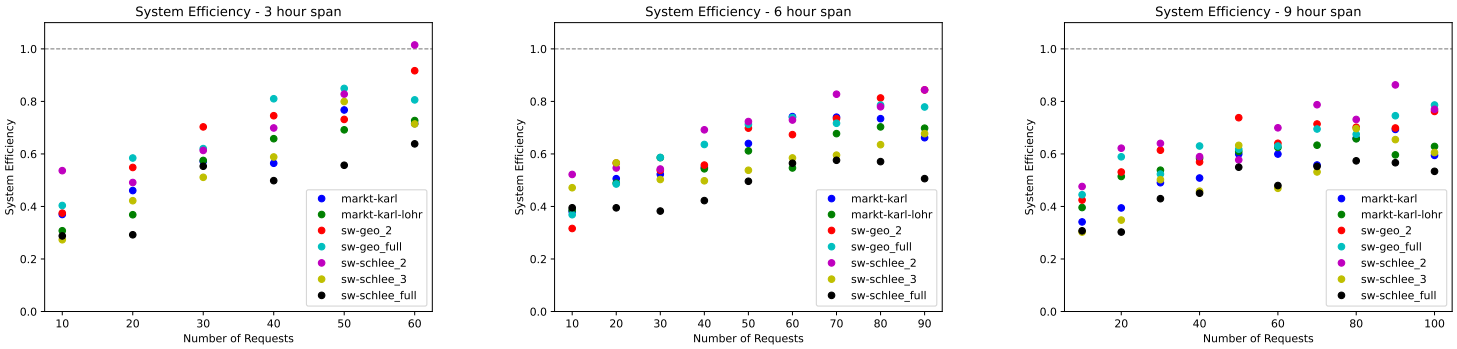


Fig. 4.8.: System Efficiency per instance.

that in this solution a relatively high amount of requests was denied with 10 out of 60 requests, see Figure 4.7. In general, we can observe a trend where the system efficiency increases with the number of available requests. This makes sense intuitively, as more requests allow for more combination possibilities to share a vehicle.

It is evident that networks with more lines show a worse efficiency, compared to ones with less lines. Especially the *SW-Schlee_full* instances produced the lowest efficiency in almost all cases. At the same time instances from *SW-Schlee_2* mostly provided the best results. Most likely, this is due to the stops being split over more lines, thus the requests need more transfers and cannot take shortcuts, like when they start and end on the same line. So, the overall deviation is longer.

To investigate this effect further, we look into the average delays for accepted requests of each instance, see Figure 4.9. The delay refers to the additional time a request takes from pick-up to drop-off compared to the shortest possible travel time in the network.

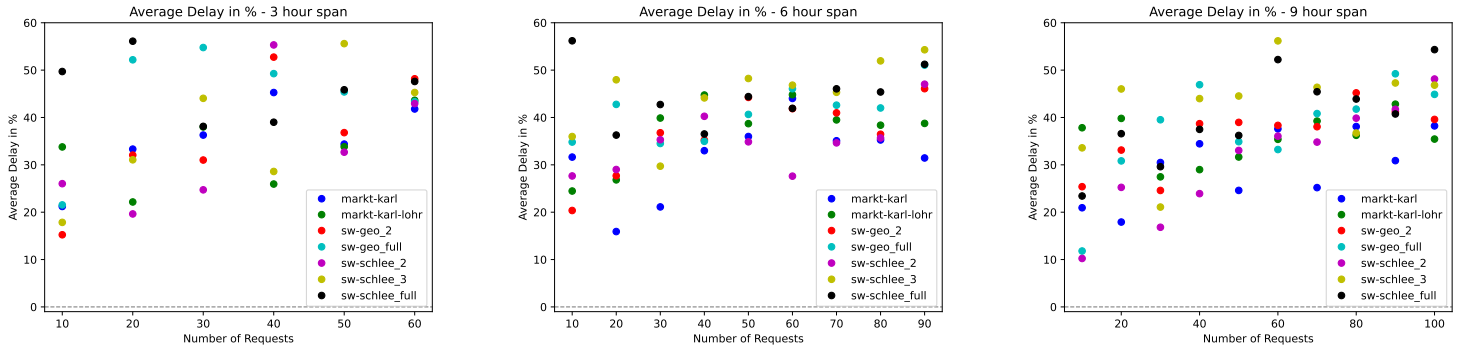


Fig. 4.9.: Average Delay in percentage per instance.

Unlike before, the values here are very spread out. In particular, for instances with a small number of requests, as the average is more susceptible to outliers. Also, the delay seems heavily reliant on the specific set of requests. Overall, the maximum average delay is around 50% to 60%, which occurs for six of the instances in the three hour span.

In general, the average delay increases with the number of available requests. This is the expected behavior, as more requests will need to be grouped together. It is hard to see a trend for any of the networks, specifically. Instances from *Markt-Karl* appear mostly on the lower end, that said in these instances many requests are denied. Similarly, in *SW-Schlee_3* not all requests are serviced, however the average delay is among the highest in many instances. It might have something to do with their different topologies, but we cannot pinpoint a specific reason with certainty.

Next, the vehicle utilization is another measurement we are interested in and it can be viewed in Figure 4.10.

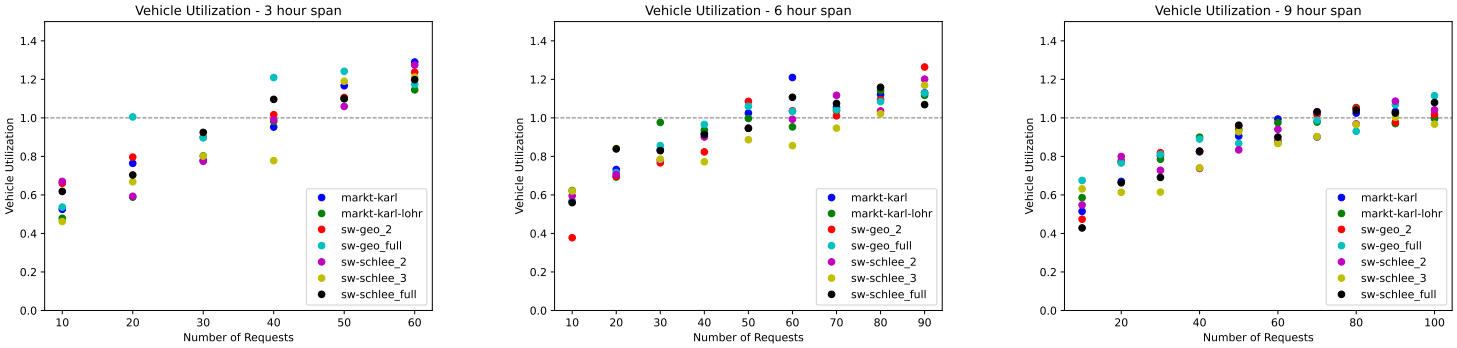


Fig. 4.10.: Vehicle Utilization per instance.

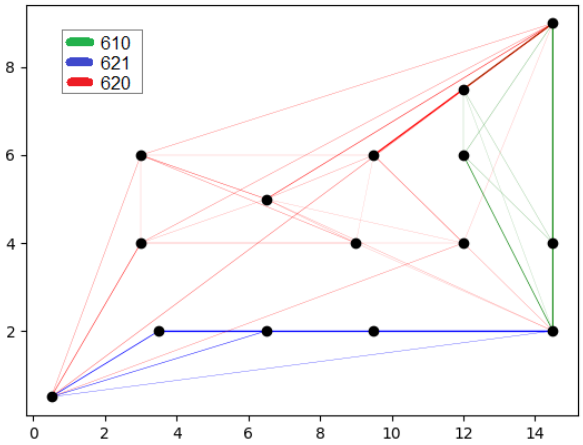
For this figure, we consider a ratio above 1.0 acceptable, as it implies that there is at least one person in the vehicle at all times on average. Most instances with a density below ten requests per hour struggled to achieve this goal, especially with a the three hour span. In the visualization for the nine hour spans, we can even see some examples that cleared the threshold earlier. Overall, the distribution follows a clearly visible curve for all time spans with little variance between the networks. Finally, it is noteworthy that for the maximum number of requests, the vehicle utilization keeps trending upwards, even though quite a few requests were denied in some of these instances. This suggests a good solution quality in those cases for the requests that were accepted.

Before we conclude this section, we will look at a few solution examples that reflect the routes of buses. They are exemplified in Figure 4.11, the frequency of each route is indicated by how bold each line is drawn. For this a *boldnessFactor* is computed for each route by

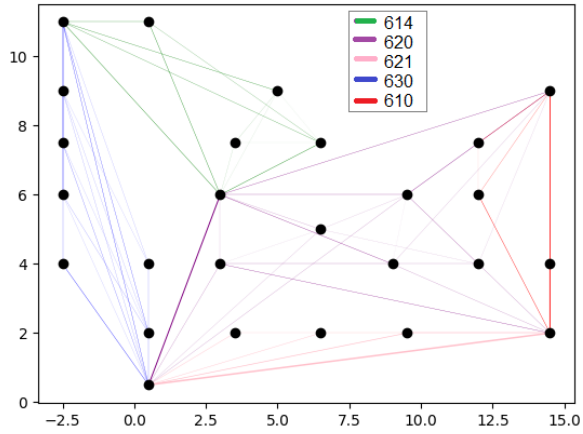
$$\text{boldnessFactor} = \frac{\text{KMRoute} \times \text{frequency}}{\text{KMtraveled}}.$$

From the visualization we can deduce that many shortcuts are used, in particular from and to transfer stops. Only in a few cases the actual network lines are visible from the solutions, for example line 621 in the *Markt-Karl* instance. Though, in most cases it seems as if the buses roam freely within their own areas, similar to the DARP setup, and deliver the requests from and to transfer stops if required.

However, with a larger number of requests or in a real-world scenario with less shortcuts, the line property could be amplified again.



(a) *Markt-Karl* with 6 hour span and 80 requests.



(b) *Markt-Karl-Lohr* with 9 hour span and 90 requests.

Fig. 4.11.: Solution examples with the bus routes between the stops and their relative frequency, depending on how bold the line is.

In summary, the results show that instances with a density of up to ten were solved optimally or with a relative MIP gap of less than 5% in all but one case. Among optimal solutions on average 99.7% of requests were accepted. For the maximum number of requests in each time span length, the solver was not able to close the MIP gap and denied up to 18% of requests in six cases. The liDARPT approach seems feasible, even though the system efficiency was rather low overall, the vehicle utilization shows that it is possible to group many requests together, which could indicate a higher efficiency in a real-world setting.

4.2.3. Complexity

After we have gained an overview over the quality of our solutions, we turn our attention to the complexity of our instances, to find key factors that make it especially hard for the solver. We expect that more lines and transfer stops should increase the complexity. The overall number of stops could be another crucial aspect, we assume that with less stops the overlap between requests increases, leading to more events and event edges. By looking at the MIP gap depicted in Figure 4.6 we can already determine particularly complex instances, that produced a big MIP gap after 15 minutes. To expand on that, the computation times for each instance can be examined in Figure 4.12.

The data shows many of the instances with a smaller density were solved in a very short time span. For up to 20 requests, we were able to find a solution almost instantly. Generally, it is not possible to distinguish the networks from each other to see a clear trend. However, for each time span length we can observe a cut off point, around 40 requests for three hours, 60 requests for six hours and 80 requests for nine hours. After

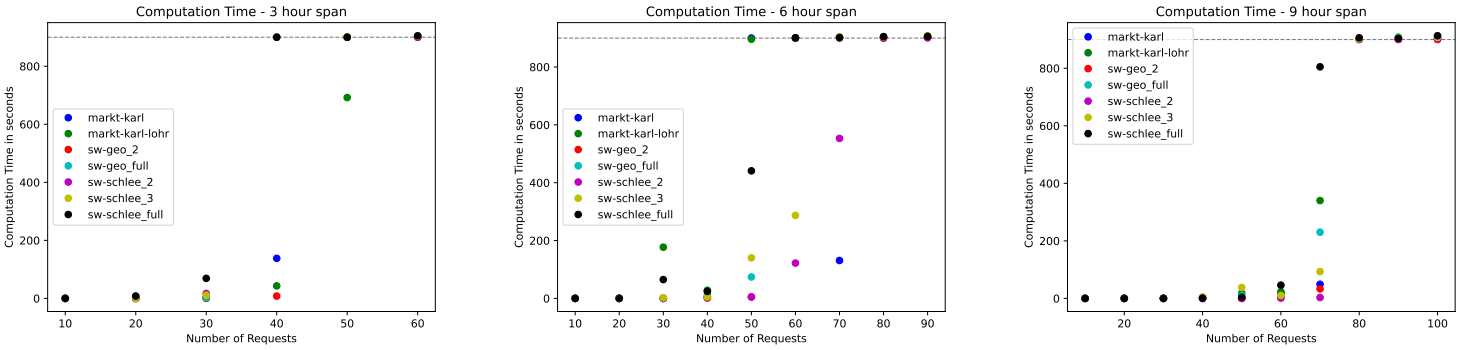


Fig. 4.12.: Computation Times per instance.

that amount almost all instances took the whole 15 minutes to solve. Before that, most examples could be solved optimally in a relatively short duration. This observation also shows that the complexity is not directly proportional to the density of requests. The densities at the cutoff points are 13.33, 10 and 8.89, it decreases with a longer time span, while there are more requests. Thus, the number of requests seems to play a bigger role for the complexity than the density alone. We will expand on this effect further later in this section.

Next, we aim to investigate the following question. Because the model and the number of constraints depend highly on the size of the event graph, we want to know if there is a clear correlation to the computation time. Furthermore, how important is the number of edges compared to the number of nodes. Looking at Figure 4.13 helps us answer these questions.

In theory, the number of edges can grow asymptotically faster than the edges by a factor of $|R|$, see Chapter 2. By plotting both the number of nodes and edges on log-scales, we can observe that they are highly proportional among our instances. This signals that we are able to prune the event graph to an acceptable size by enforcing restrictions like time windows and locality constraints in preprocessing. Furthermore, there seems to exist a transition area around 1,000 nodes and 10,000 edges. Before this, nearly all instances took a short time to solve. For bigger event graphs, the solver was unable to find a solution within 15 minutes. With both of these properties combined, we arrive at the conclusion that we can evaluate the complexity by either one of the graph's components, as they are proportional and show a trend regarding the computation time.

Following this finding, we explore the number of event nodes for every instance in Figure 4.14 to evaluate the complexity of each network, without being restricted by the maximum computation time.

For each time span length, the number of event nodes for each network seems to follow exponential growth in the number of available requests. In particular, when tracing *SW-Geo_full*'s instances in the six hour span this trend becomes more obvious. As expected, the networks with the maximum number of lines result in the highest amounts of events. Even though the number of stops varies between the networks, we cannot declare any

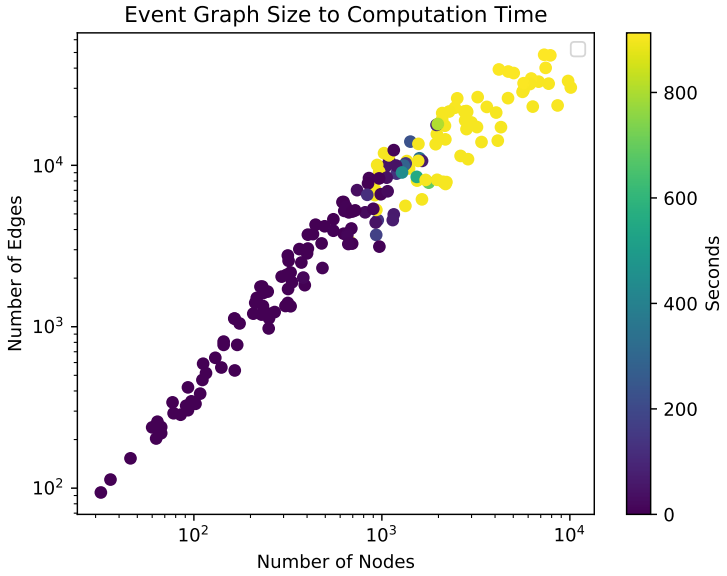


Fig. 4.13.: Size of event graphs in relation to computation time. Number of nodes to number of edges in log-scales.

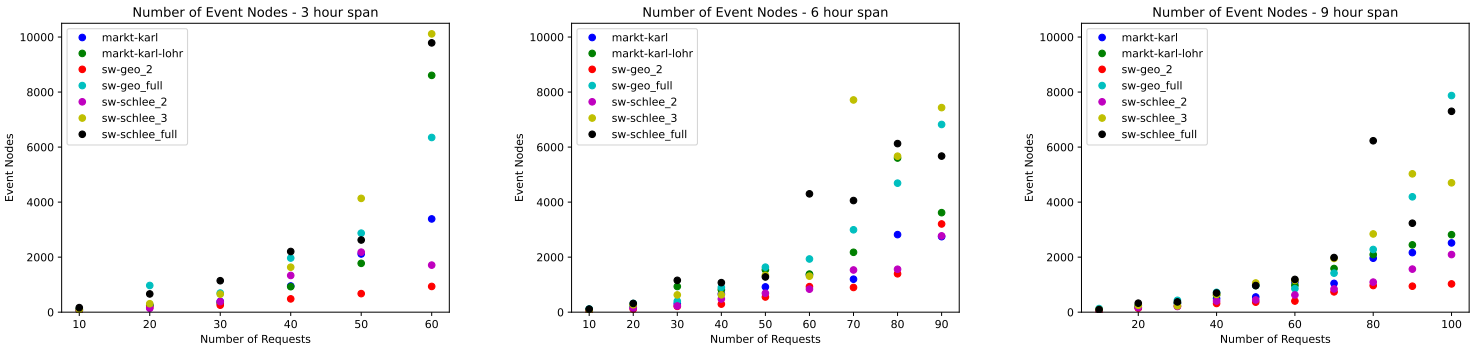


Fig. 4.14.: Number of event nodes in the graph per instance.

one of them to be the most complex in general, as the values vary too much from instance to instance.

Surprisingly, test examples from *SW-Schlee_3* often produce very large event graph sizes. Sometimes even larger than the five line networks, despite consisting of only three lines. Compared to the other network *Markt-Karl* that provides three lines as well, we can observe a very large difference when there are many requests. The latter is comprised of less stops, however we expected less stops would generally lead to more events, due to more overlap of routes. It might be a result of the unique topologies, as examining the related *Markt-Karl-Lohr* and *SW-Schlee_full* networks against each other reveals a similar trend.

The plots would indicate that networks with only two lines are easier to solve, as their instances are consistently among the bottom. Also, we can find evidence to support our assumption about the number of stops, as one of the larger networks *SW-Geo_2* results in the smallest event graphs in our tests. However, by comparing these findings to Figure 4.6 there seems to be important factors missing here. In the figure above, the biggest MIP gaps were found for networks with only two lines. We assume that even though the model is significantly smaller in these instances, it is harder to solve based on the number of buses. The ratio of buses to overall passengers is a lot worse here, which seems to be problematic for the solver when trying to distribute the requests over all vehicles.

4.2.4. DARP Comparison

As we have stated before, the liDARPT is a restriction of the DARP. Our hope is to encounter solutions with less detours and faster computation times. To test these assumptions we used the event-based DARP model from [GKS22]. The model is also mentioned in [GKP$^+$24] along with two newer models. All three have been implemented in C++ and the code is publicly available on gitlab[1]. However, only the standard event-based model enables rejection of requests, which is a necessary criteria for us to compare the two.

After converting the exact instances discussed in Section 4.2.2 to fit the required input format, we ran the tests with identical settings. Including a maximum solve time of 15 minutes, the same number of buses and the same time windows for every request. All other settings described in Section 3.1 stayed identical as well.

Before we start comparing the quality of results, we will first inspect the relative MIP gap in the DARP solutions. We can get an idea of the complexity and solution quality of each instance by looking at the outcomes compared to the liDARPT in Figure 4.15.

First, it should be noted that the solver was not able to find any solutions for two of the instances from network *SW-Schlee_full* within 15 minutes. More specifically, for a three hour span and 60 requests and a six hour span and 80 requests. Furthermore, for the instance *SW-Geo_full* with a six hour span and 90 requests the preprocessing time took over one hour, so we did not complete a solve. From the remaining data points, we

[1] `https://git.uni-wuppertal.de/dgaul/a-tight-formulation-for-the-darp`

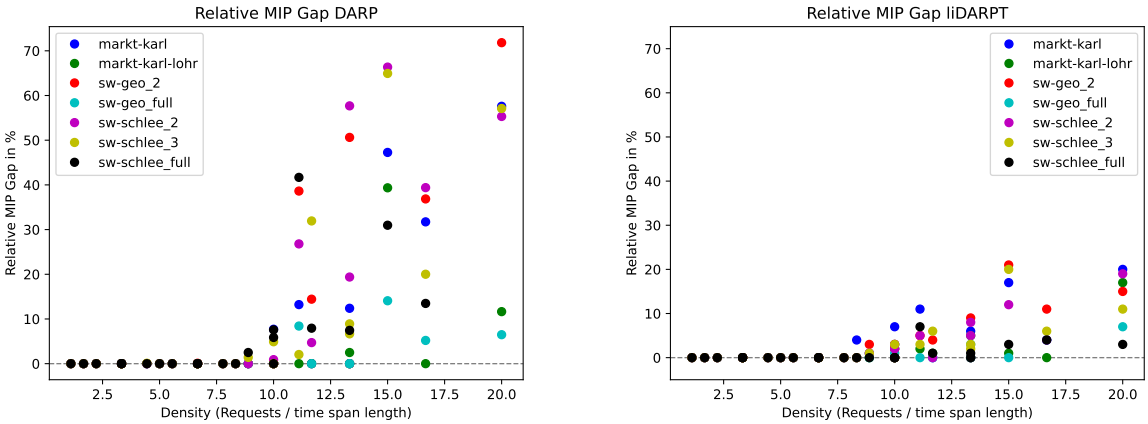


Fig. 4.15.: Relative MIP Gap of DARP and liDARPT solutions by density of requests.

can observe that most instances with a density of less than ten were solved optimally. However, for more dense examples, the relative MIP gap increases rather quickly, with 18 instances exceeding a 20% gap up to 70% in some cases, thus surpassing the encountered gaps in the liDARPT by over 50%.

Similarly to the liDARPT, in the DARP setting the complexity is not directly proportional to the density. There is already a spike up to 40% at density 11.11 that corresponds to a nine hour span and 100 requests. For the time span of six hours the smallest instances that exceed a gap of 40% are at 13.33, while for three hours that first occurs with 50 requests and a density of 16.67. As the time span increases, relatively fewer requests are required to cause large MIP gaps, indicating that the complexity is not proportional to the density.

At last, it seems the solver had the most problems when there were less busses at its disposal, *SW-Schlee_2* and *SW-Geo_2* both received 4 buses in total. These lay among the top of the plots and for the latter no solution was found in two cases. The same trend can be found for networks with more buses. These were consistently placed the lowest.

Overall, we can make many of the same observations as for the liDARPT, however there seems to be a clear difference in complexity of the models, as the relative MIP gaps are significantly higher for the DARP.

Next, we will turn our attention to the number of accepted requests, by evaluating the difference in percentage between both settings per instance, see Figure 4.16. Positive values indicate that instances have more accepted requests in the liDARPT setting and negative the opposite.

On first glance, it appears that in general more requests are accepted in the DARP. More specifically, in roughly 27% of instances this was the case, while the liDARPT was better in about 8% of examples. Even though the MIP gap is worse in most instances, the DARP tends to accept more requests. In particular, looking only at instances that were solved within 20% of optimality in the DARP, of which there are 153, shows that

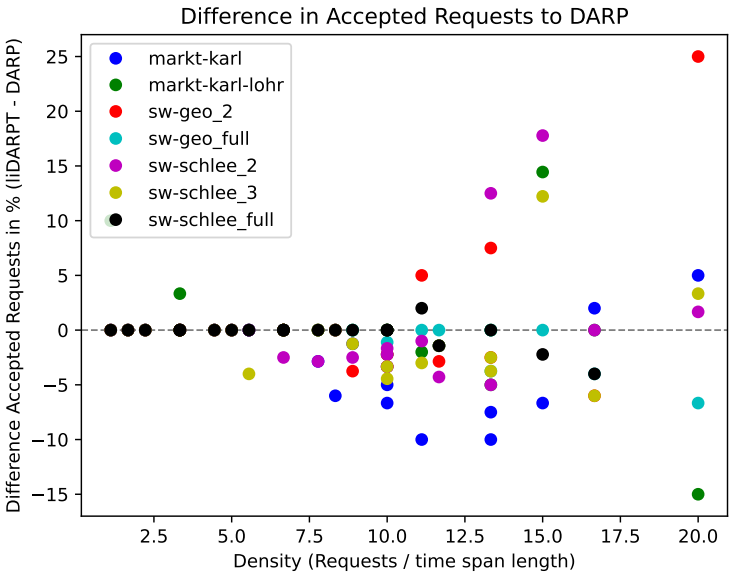


Fig. 4.16.: Difference in accepted Requests in percentage per instance, between DARP and liDARPT solutions.

in only two cases the liDARPT accepted more, which is roughly 1%. In general, the differences mostly range between 1% to 10% with a few outliers up to 25%. Especially at larger densities we can observe some cases, where the liDARPT exceeds the DARP. As we have seen in Figure 4.15 these instances were not solved close to optimal in the latter. Overall, it becomes evident that the liDARPT is more restricted, which leads to more rejected requests.

Following that, we want to figure out if the liDARPT results in more efficient tours. The difference per instance is shown in Figure 4.17. Again, if a marker is placed in the positive region, it means that the liDARPT solution is more efficient.

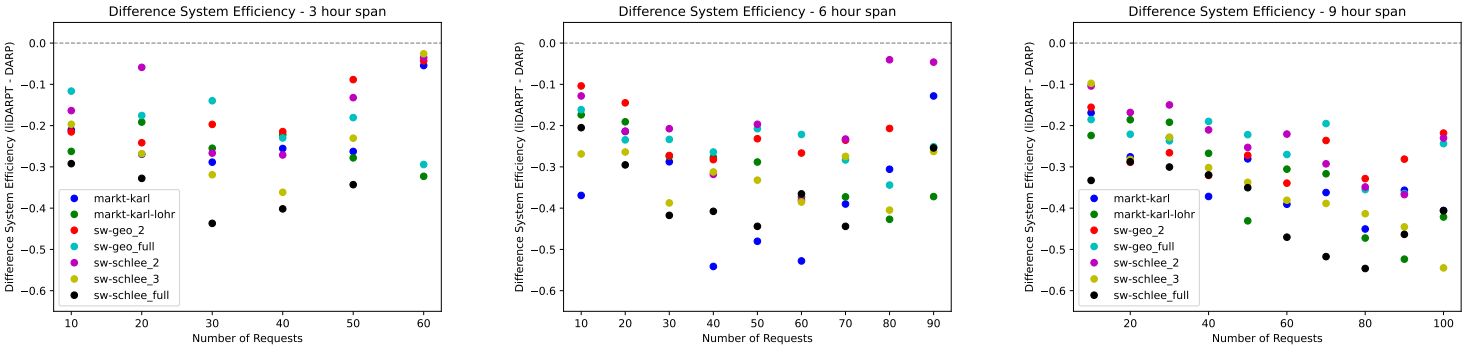


Fig. 4.17.: Difference in System Efficiency per time span, between liDARPT to DARP.

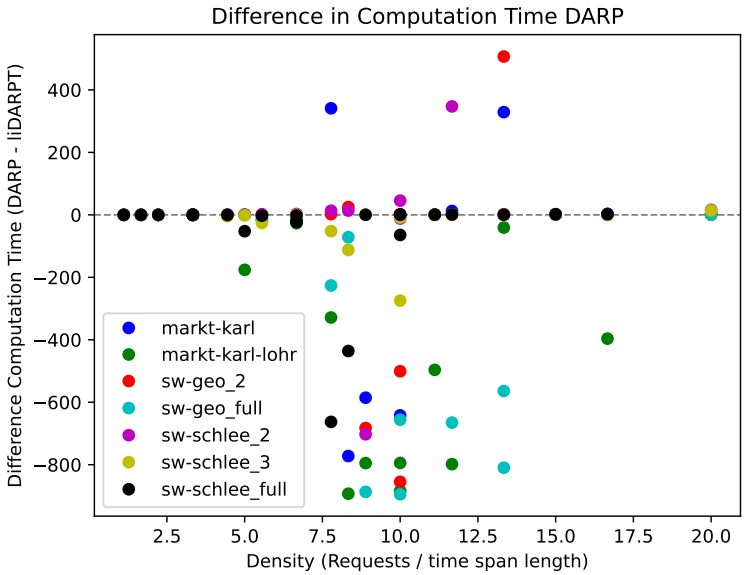


Fig. 4.18.: Difference in computation time of the solver between liDARPT and DARP.

However, we can observer a clear trend as all markers are placed in the negative range, meaning the DARP approach lead to more efficient solutions. In particular, for 43 examples the system efficiency exceeded 1.0 in the DARP setting. In the figure a slight upward curve is visible towards the end, meaning close to the maximum of requests the difference started to shrink down. However, this is most likely due to the big relative MIP gaps occurring for these DARP results. Overall, it appears that having the buses restricted to lines decreases the efficiency for a lower number of requests. Especially, as the buses in the DARP can always travel along a straight line in our simplified setting, while in a real-world scenario driving along actual streets involves more deviations. Furthermore, the effect is heightened by the number of lines in the liDARPT network, as we can see networks like *SW-Schlee_full* place among the lowest in many cases. At the same time, networks with less lines like *SW-Geo_2* and *SW-Schlee_2* appear closer to zero.

After we have compared the quality of solutions against each other, we want to conclude our findings by investigating the difference in computation time needed by the solver, depicted in Figure 4.18. We have already seen from Figure 4.15 that the liDARPT approach scales better with more requests in this regard. Now, our focus shifts to smaller instances. By subtracting the computation time needed for the DARP by the liDARPT solution time, positive values correspond to instances where the liDARPT was able to solve the model faster.

It becomes evident that the DARP approach leads to faster computation times in most cases. This was the case for 33% of all compared instances, while for roughly 24% the liDARPT setting excelled. The gap between the two becomes even more obvious

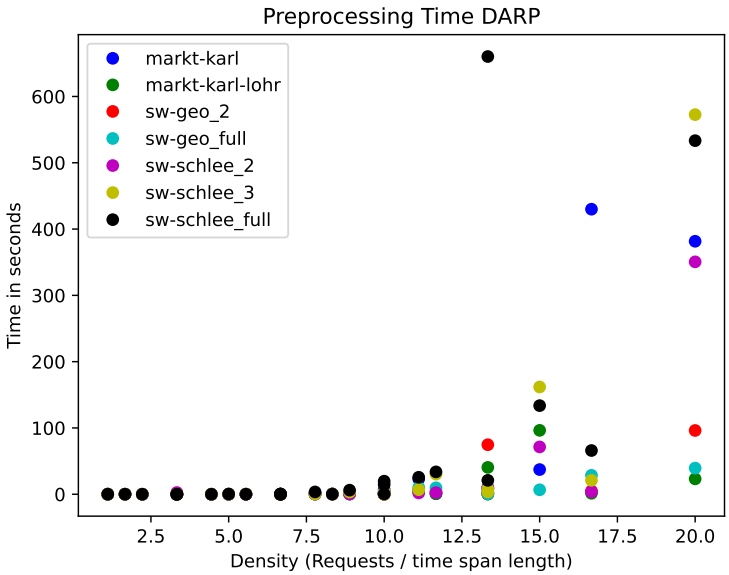


Fig. 4.19.: Preprocessing time of DARP approach.

when examining the average difference when either was better. For instances where the liDARPT was faster, the average difference is 40.7 seconds, while for the opposite the average time saved was 298.4 seconds.

Very small instances, with a density of less than seven, were solved instantaneous in almost all cases in both approaches. Thus, the difference is close or equal zero. Interestingly, for instances with a slightly bigger density the solver generally needed less time in the DARP setting. Including quite a few examples with a difference between 400 to 800 seconds in favor of the DARP. Among these data points are mostly the networks *SW-Geo_full* and *Markt-Karl-Lohr*. Both of which consist of the maximum number of lines being five. So even though the MIP gap was rather low in the liDARPT for these instances, the solver was not able to close it entirely, which was achieved much earlier in the DARP setting, with close to the maximum number of requests, see Figure 4.16.

Apart from this, there are also a few outliers in the other direction. As the density exceeds 15, in both approaches the full 15 minutes were needed for finding an optimal solution, except for one instance of the *Markt-Karl-Lohr* network.

From the results we discussed so far, the DARP excels in most of the aspects. However, there is one factor we did not include so far and that is the time needed for preprocessing. That includes generating the events, followed by finding and pruning edges. The computation time required for this step is depicted in Figure 4.19.

As stated before, one of the instances was left out entirely due to the preprocessing time. By observing the plot we can see a great increase for the most dense instances. As well as some outliers before that, where the preprocessing step took a couple of minutes. Interestingly, the instances that consistently took the longest consider a three hour span,

even though the maximum number of requests is only 60 here. They can be found at density 20 on the far right side of the figure. The density appears to be an important factor for the preprocessing time. Differing to many of our other findings, where it looks like the number of requests is more important than just the ratio.

In the liDARPT, preprocessing was done within seconds for all instances. The longest time being 13 seconds in one case. Here, we can observe one of the limitations of the DARP, as the event graphs become too big to solve very soon.

This becomes particularly evident from the event graph sizes for the DARP compared to the liDARPT, exemplified in Figure 4.20.

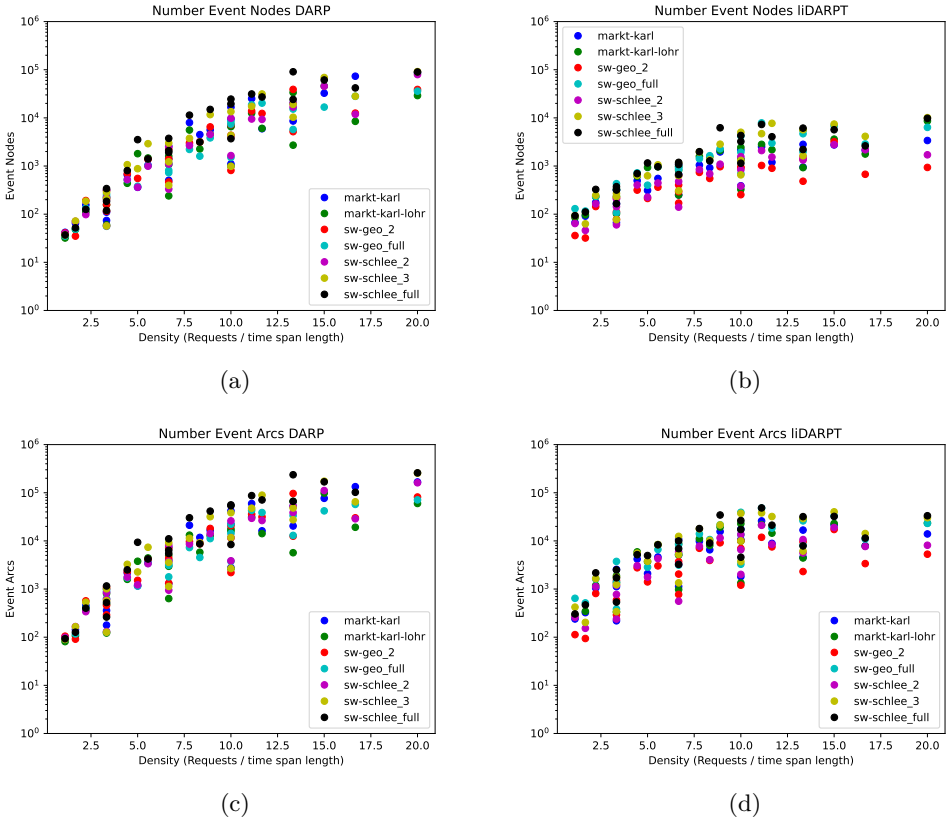


Fig. 4.20.: The number of event nodes in DARP (a) and liDARPT model (b) and number of event arcs in DARP (c) and liDARPT model (d) by density.

On first glance the figures of event nodes and arcs appear very similar for each approach. The relative positioning of the instances to another is almost identical, however the overall amount of arcs seems to be larger by a constant factor. We already discovered this behavior for the liDARPT in Section 4.2.3. Here, we can observe that for the DARP the number of edges is proportional to the number of nodes as well.

For both variants we can identify a curve in each plot. That said, there is a clear difference in the slope between the two. While the graphs rise up to 10,000 nodes in the

DARP, in the liDARPT the largest graphs lie around 1,000 nodes. Again, this suggests a better scalability in the liDARPT as the density of requests increases.

In summary, our results show that the DARP tends to deliver better solutions. On one hand, in terms of accepted requests, with up to 10% more in a few of our tests. On the other hand in overall efficiency, where the difference ranged from 0.1 to 0.55 in favor of the DARP. In particular, the medium sized instances in our tests revealed this trend with faster computation times. However, it should be noted that the DARP solutions decreased in quality with more requests, where the MIP gap increased faster. The liDARPT approach seems to scale better with the number of requests, exemplified by the preprocessing times and event graph sizes.

5. Conclusion

In this thesis, we generalized the liDARP approach from [RSS24] to the liDARPT, a multi-line system that enables users to transfer between buses. Besides formulating a comprehensive problem definition for this problem, we also developed an event-based MILP model that can be used to obtain an exact solution. We then implemented a Python program that can read in various problem instances and perform the necessary preprocessing steps within seconds. After which, it constructs our MILP model to generate a solution that maximizes the number of requests and secondly minimizes the overall travel distance.

The program was used to extensively test the feasibility of the liDARPT approach. Among the tested instances were networks with up to five lines, while varying the number of requests from 10 to 100 and the time span length, so different densities of requests were tested. First, we evaluated two design choices for the objective function against each other. Comparing a single lexicographically weighted function to executing two solves with different objectives. Under an overall time limit of 15 minutes, the former clearly outperformed the multi-objective approach when faced with medium to large amounts of requests. This lead to a difference in relative MIP gaps of up to 80% and for 67% of the larger instances to more accepted requests within our time limit.

Afterwards, we investigated the quality of solutions more thoroughly. The model was solved optimally for instances with a density of up to eight requests per hour. Here, none of the solutions denied more than 7.5% of available requests and on average 99.7% of requests were accepted. Thus, for almost all of the requests, routes were found that comply with the time restrictions, proving the feasibility of the liDARPT approach. However, investigating the system efficiency revealed that the overall distance traveled by the buses exceeds the sum of distances between each pick-up and drop-off pair in all but one case. Meaning that, if all accepted users were to use their own vehicle, it would lead to less CO_2 emissions in these instances. That said, we could observe an upward trend with an increasing number of available requests, reaching a system efficiency of 0.8 for the largest instances.

Investigating the complexity of the model and finding important factors was another one of our goals. We found a clear correspondence of the event graph size to the computation time. Furthermore, the number of event edges is proportional to the number of events in our test instances. This could be an important property, as it may allow for better scalability. While the number of events appears to depend heavily on density of requests, the results suggest that the total number of requests plays a more significant role. However, the event graph size is not the only factor, as we observed that instances with less lines produce relatively small event graphs. However, for larger instances their relative MIP gaps and computation times exceeded networks with more lines in many

cases. This implies that a decrease in the ratio of buses to number of requests causes the complexity to rise.

Finally, we compared the liDARPT setting directly to the related DARP, by employing the C++ implementation from [GKP+24]. Our aim was to see if the liDARPT could prove to be a more efficient alternative. Generally, the results revealed the less restricted DARP was able to service more requests in roughly 27% of instances, with at most a 10% increase. It also achieved more efficient solutions, surpassing a system efficiency of 1.0 in 43 instances, as it was able to use more shortcuts. However, we could also observe the DARP model was struggling with higher request densities, leading to MIP gaps of up to 70% in a limited time setting. Comparably, the liDARPT appears to scale better with bigger instance sizes, which is also evident from the event graph sizes.

In future research, one could explore the liDARPT's event-based MILP model further with bigger instance sizes and longer computation times. Including, more extensive hyperparameter tests, for example varying the number of buses or their capacities in the networks to investigate how these affect the results and complexity. Furthermore, in this work we only considered a static setting, we assume that in a dynamic context the liDARPT approach could close the gap in performance to the DARP setting. As the vehicles would be more equally distributed in the network at all times, it should be easier to adjust its strategy to newly incoming requests.

A. Appendix

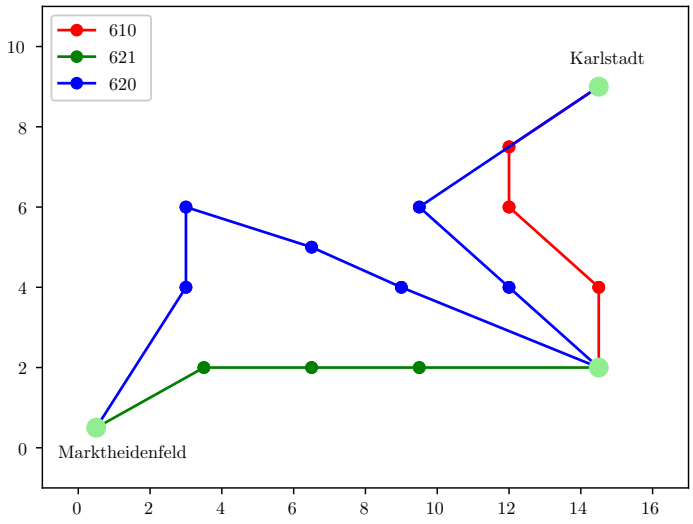


Fig. A.1.: Visualization of the network *Markt-Karl*.

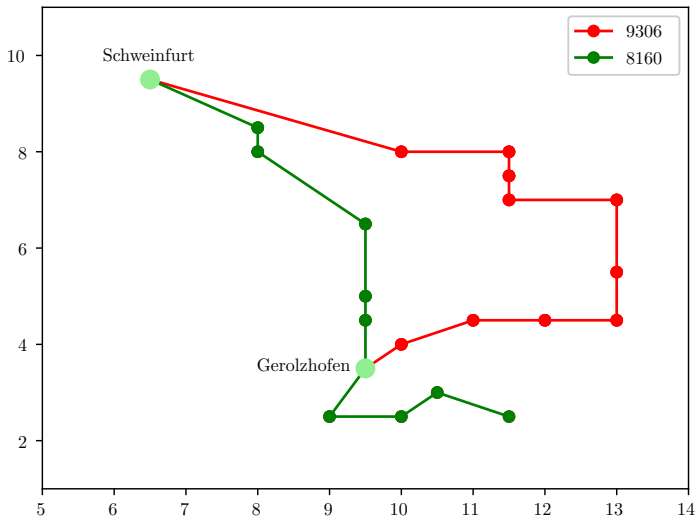


Fig. A.2.: Visualization of the network *SW-Geo_2*.

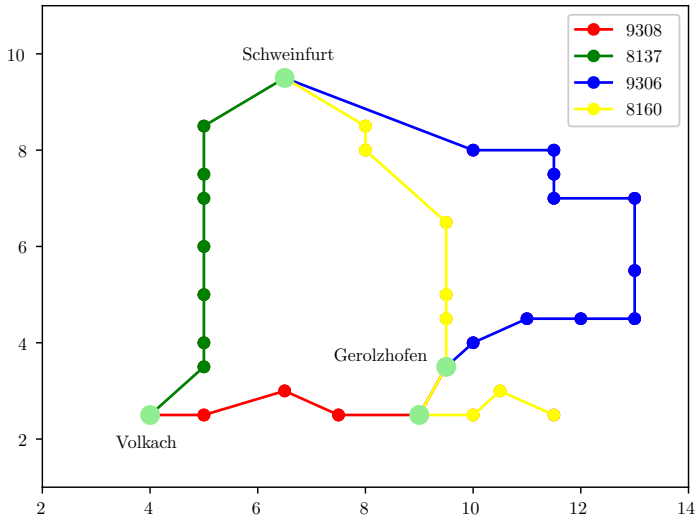


Fig. A.3.: Visualization of the network *SW-Geo_full*.

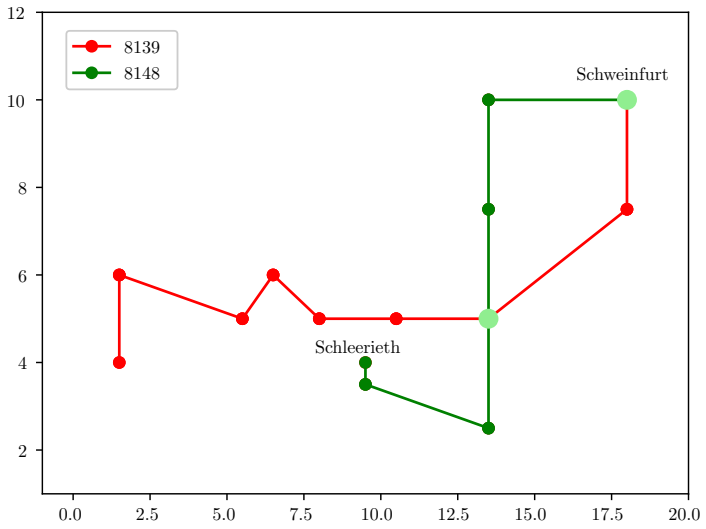


Fig. A.4.: Visualization of the network *SW-Schlee_2*.

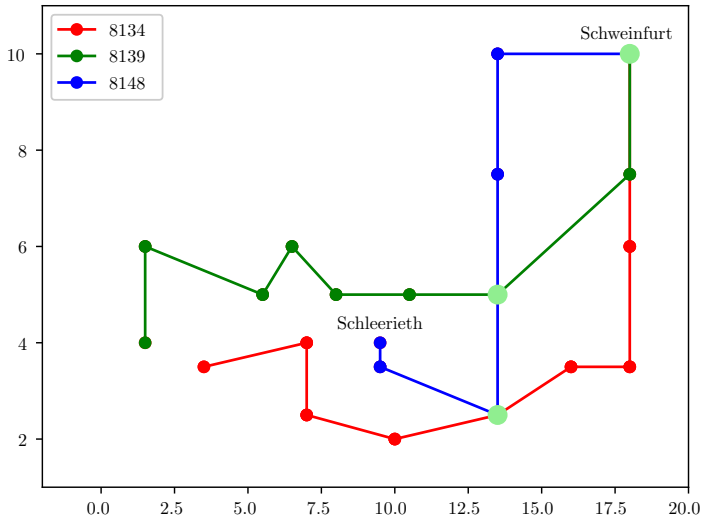


Fig. A.5.: Visualization of the network *SW-Schlee_3*.

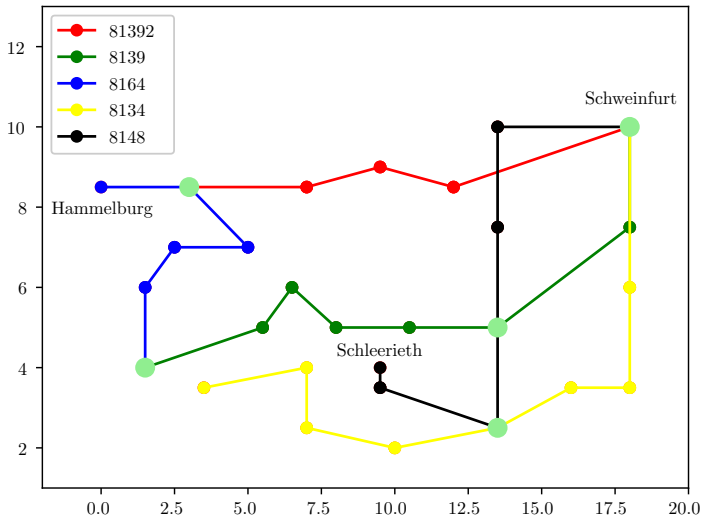


Fig. A.6.: Visualization of the network *SW-Schlee_full*.

Bibliography

[CL07] Jean François Cordeau and Gilbert Laporte: The dial-a-ride problem: models and algorithms. *Annals of operations research*, 153:29–46, 2007, 10.1007/s10479-007-0170-8.

[DDS92] Martin Desrochers, Jacques Desrosiers, and Marius Solomon: A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, 40(2):342–354, 1992, 10.1287/opre.40.2.342.

[DQ13] Samuel Deleplanque and Alain Quilliot: Dial-a-ride problem with time windows, transshipments, and dynamic transfer points. *IFAC Proceedings Volumes*, 46(9):1256–1261, 2013, 10.3182/20130619-3-RU-3018.00435, ISSN 1474-6670. 7th IFAC Conference on Manufacturing Modelling, Management, and Control.

[Flo20] Hannah Florian: Hellblaue „london cabs“ als ergänzung zum wuppertaler Öpnv, 2020. `https://www.wz.de/nrw/wuppertal/on-demand-projekt-wsw-cabs-als-ergaenzung-zum-wuppertaler-oepnv_aid-53941827`, Accessed on the February 24th, 2025.

[fS24] Bundesamt für Statistik: Pkw-dichte 2024 leicht gestiegen, 2024. `https://www.destatis.de/DE/Presse/Pressemitteilungen/2024/10/PD24_N051_46.html`, Accessed on the January 26th, 2025.

[GKP+24] Daniela Gaul, Kathrin Klamroth, Christian Pfeiffer, Michael Stiglmayr, and Arne Schulz: A tight formulation for the dial-a-ride problem. *European Journal of Operational Research*, 2024, 10.1016/j.ejor.2024.09.028.

[GKS22] Daniela Gaul, Kathrin Klamroth, and Michael Stiglmayr: Event-based milp models for ridepooling applications. *European Journal of Operational Research*, 301(3):1048–1063, 2022, 10.1016/j.ejor.2021.11.053.

[GN23] Konstantinos Gkiotsalitis and A Nikolopoulou: The multi-vehicle dial-a-ride problem with interchange and perceived passenger travel times. *Transportation research part C: emerging technologies*, 156:104353, 2023, 10.1016/j.trc.2023.104353.

[HSK+18] Sin C. Ho, W.Y. Szeto, Yong Hong Kuo, Janny M.Y. Leung, Matthew Petering, and Terence W.H. Tou: A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological*, 111:395–421, 2018, 10.1016/j.trb.2018.02.001, ISSN 0191-2615.

[LLMS21] Christian Liebchen, Martin Lehnert, Christian Mehlert, and Martin Schiefelbusch: *Betriebliche Effizienzgrößen für Ridepooling-Systeme*, pages 135–150. Springer Fachmedien Wiesbaden, Wiesbaden, 2021, ISBN 978-3-658-32266-3, 10.1007/978-3-658-32266-3$_7$.

[MLP14] Renaud Masson, Fabien Lehuédé, and Olivier Péton: The dial-a-ride problem with transfers. *Computers Operations Research*, 41:12–23, 2014, 10.1016/j.cor.2013.07.020, ISSN 0305-0548.

[RCL07] Stefan Ropke, Jean François Cordeau, and Gilbert Laporte: Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks*, 49(4):258–272, 2007, 10.1002/net.20177.

[RF21] Yannik Rist and Michael A. Forbes: A new formulation for the dial-a-ride problem. *Transportation Science*, 55(5):1113–1135, 2021, 10.1287/trsc.2021.1044.

[RSS24] Kendra Reiter, Marie Schmidt, and Michael Stiglmayr: The line-based dial-a-ride problem. In *24th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, 2024, 10.4230/OASIcs.ATMOS.2024.14.

[Sch24] Josef Schäfer: Erfolgreiches ruftaxi: Callheinz schwärmt 130 mal am tag aus und soll ab august im ganzen kreis fahren, 2024. `https://www.mainpost.de/regional/schweinfurt/erfolgreiches-ruftaxi-callheinz-schwaermt-130-mal-am-tag-aus-und-soll-ab-august-im-ganzen-kreis-fahren-art-11363579`, Accessed on the February 24th, 2025.

[TLZO01] Kay Chen Tan, Loo Hay Lee, QL Zhu, and Ke Ou: Heuristic methods for vehicle routing problem with time windows. *Artificial intelligence in Engineering*, 15(3):281–295, 2001, 10.1016/S0954-1810(01)00005-X.

[TRA24] Climate TRACE: Greenhouse gas emissions statistics, 2024. `https://climatetrace.org/inventory?year_from=2023&year_to=2023&gas=co2e100`, Accessed on the January 26th, 2025.

[Umw24] Umweltbundesamt: Fahrleistungen, verkehrsleistungen und modal split, 2024. `https://www.umweltbundesamt.de/daten/verkehr/fahrleistungen-verkehrsaufwand-modal-split#anmerkung`, Accessed on the January 26th, 2025.

In this thesis, chatbots like ChatGPT were used occasionally to help with translating words into english, finding synonyms and reformulating sentences to improve readability. All of the work, analysis and conclusions were entirely made on my own.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur selbständigen Leistungserbringung

Titel der Masterarbeit:

Line-based Dial-A-Ride Problem with Transfers

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):

Prof. Dr. Marie Schmidt, Lehrstuhl für Informatik I

Eingereicht durch (Vorname, Nachname, Matrikel):

Jonas Barth, 2505270

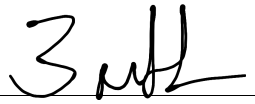
Ich versichere, dass ich die vorstehende schriftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene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

Mönchstockheim, 15.06.2025 

Ort, Datum, Unterschrift