Transit Planning Utilizing Ride Sharing Techniques

Master's Thesis of

Maximilian Walz

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	PD Dr. Torsten Ueckerdt
Advisors:	Adrian Feilhauer
	Michael Zündorf

1 January 2025 – 1 July 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 1 July 2025

[illegible]

. . . .

(Maximilian Walz)

Abstract

Problems dealing with serving passengers' travel demands using shared vehicles occur in both the transit planning as well as the ride sharing domain. Nevertheless, to the best of our knowledge, there is no work transferring techniques between them. Bridging this gap, we pursue to solve a transit planning problem by adapting a technique based around calculating possible detours of passengers to find common routes that can be travelled in shared vehicles. In transit planning, the drivers sharing their vehicle correspond to buses and have no agenda on their own, implying there is no constraint in their number or itineraries. Our goal is to settle the passenger demand within their respective time constraints, while using as few buses as possible and minimizing the overall passenger transfers. To that end we use the possible passenger detours to determine demand for buses between pairs of stations, that is preferably shared by many passengers. We then construct an instance of Minimum Cost Flow from it, which yields the required set of bus itineraries. We evaluate our approach by comparing it to a naïve baseline solver, which we outperform. However, there still is a lot of potential for improvement in the number of required buses. The number of transfers along with waiting times and delays, on the other hand, stay notably low. We attribute this to the approach originating in ride sharing, where transfers, delays and waiting times are generally less acceptable than in public transit.

Zusammenfassung

Die Problemstellung, Fahrgastnachfragen mit geteilten Fahrzeugen zu bedienen, stellt sich sowohl in der Verkehrsnetzplanung als auch im Bereich Ride-Sharing. Trotz dieser inhaltlichen Nähe gibt es, soweit wir wissen, bislang keine Ansätze, die Methoden zwischen beiden Bereichen übertragen. Diese Lücke greifen wir auf und schlagen einen neuen Ansatz zur Lösung eines Problems der Verkehrsnetzplanung vor, bei dem wir eine Methode aus dem Ride-Sharing adaptieren. Durch die Berechnung möglicher Umwege von Fahrgästen identifizieren wir gemeinsame Routen, die sich für den Einsatz geteilter Fahrzeuge eignen. In unserem Szenario übernehmen Busse die Rolle der geteilten Fahrzeuge, wobei sie keine festen Routen oder Kapazitätsbeschränkungen haben. Ziel ist es, die Fahrgastnachfrage unter Berücksichtigung der jeweiligen Zeitfenster zu bedienen und dabei möglichst wenige Busse zu verwenden, sowie Umstiege zu minimieren. Dazu nutzen wir die ermittelten Umwege, um stark nachgefragte Streckenabschnitte zwischen Stationen zu identifizieren, die idealerweise von mehreren Fahrgästen gemeinsam genutzt werden. Diese Nachfrage modellieren wir anschließend als ein Minimum Cost Flow-Problem, aus dessen Lösung sich die gesuchten Busrouten ergeben. Unsere Methode zeigt im Vergleich zu einer einfachen Referenz-Lösung bessere Ergebnisse. Allerdings besteht noch Verbesserungspotenzial hinsichtlich der benötigten Anzahl an Bussen. Die Anzahl der Umstiege sowie Wartezeiten und Verspätungen bleiben dagegen auffallend gering. Diesen Effekt schreiben wir der Herkunft unseres Ansatzes aus dem Ride-Sharing zu, wo Umstiege, Wartzeiten und Verspätungen generell weniger Akzeptable als im klassischen öffentlichen Verkehr sind.

Contents

1. Introduction

In this work, we adapt an approach from ride sharing, for solving a variation of the ride sharing problem, to the domain of transit planning. There, we use it as part of our multi-step procedure to solve a time-based version of the Transit Network Design problem.

1.1. Motivation

The domains of transit planning and ride sharing have a lot of conceptual similarities. At their core, both involve coordinating the travel plans of participants using shared transportation vehicles. These participants seek to travel from their individual origins to their individual destinations, commonly also respecting particular travel-time windows. In ride sharing systems, the people are usually partitioned into drivers and riders, where the drivers are the ones providing the vehicles, while the riders are matched to drivers based on compatible routes and schedules. Public transit, in contrast, traditionally operates on predefined routes and timetables. Moreover, public transport relies on vehicles such as buses, trams or trains that are not tied to the travel destination of any single passenger. Despite these operational differences, we argue that public transit can be interpreted as a ride-sharing system, where vehicles serve collective demand patterns. This perspective opens the possibility of adapting techniques originating in ride sharing literature to transit planning.

Specifically, we focus on adapting a method that computes feasible detours of participants to identify ride-sharing potential. To us, having the information of all possible detours seems a very powerful tool to ascertain common routes, which is why we choose this approach. While the original algorithm was developed under real-time constraints, our transit planning context does not impose such strict timing requirements. This relaxation allows us to explore its computational properties more broadly, even though we cannot exploit the constraints introduced by drivers having their personal travel agenda, which is incorporated in the original ride sharing scenario. By creating and investigating this adaptation, we aim to explore the extent to which such methods can be applied to a transit planning scenario and contribute to more demand-responsive transit systems.

1.2. Related Work

The field of transit planning ranges from designing route networks, determining vehicle frequencies to the planning of concrete timetables and the assignments of individual vehicles and drivers [CW86]. Initially, these sub-problems where considered individually and often solved in sequence, where each stage offers a broad variety of constraints and optimization criteria themselves. Mandl [Man80], for instance, proposes approaches for assigning passengers to routes and vehicles to routes separately, while optimizing for passenger transportation costs. They also suggest a method to improve routes on existing transit networks. Salzborn [Sal72] deals with determining bus departure rates to minimize passenger waiting times. In later research, it is common to incorporate the time-component directly in the route designing process, like Zhou et al. [ZYWY21], who propose an optimization of line configuration and frequency setting in regard to a combination of passenger and operator costs.

With ride sharing, on the other hand, the departures of drivers are inherently part of the problem, since drivers always serve the demands of specific riders. Key differences in works concerning ride sharing are whether riders and drivers have fixed or flexible roles, rides back are guaranteed, multiple hops or transportation modes are allowed and a driver will take multiple riders simultaneously [TMY20]. In some works, the exact problem configuration is motivated by concrete real world applications such as to-work scenarios [CMSQ19]. The integration of additional, sometimes autonomous, external vehicles to back up drivers is also subject of research [ANJ19 | CMSQ19]. Peer-to-peer ride sharing is arguably the most common area of ride sharing covered in literature and is centred around the idea of some real-time system, where riders and drivers can register their requests and rides on demand [Nam+18 | MNYJ17].

Despite the similarities between transit planning and ride sharing, we found few works connecting these domains. The ones mentioning ride sharing alongside transit planning, for the most part discuss them as supplementary or sometimes even competing modes of transport in an urban environment [WVJ22 | MOCB21 | Cho+20].

1.3. Outline

In Chapter 2, we cover graph theory basics and notation used throughout this work as well as give an introduction to transit planning and ride sharing.

Chapter 3 presents our detour based-approach to solve the Transit Network Design and Timetabling problem, by first covering the computation of the possible passenger detours in the form of detour-DAGs, which are then collapsed to find the demand of bus rides between station pairs for each passenger. We continue, describing how they are further processed to compute the actual bus rides between station pairs and finally present our bus ride flow network construction, which concludes the chapter. In Chapter 4, we introduce a simple naïve solver that acts as a baseline. We then evaluate our detour-DAG computation along with our detour-solver in Chapter 5, by comparing it with the baseline solver and discussing the results of our experiments on a realistic graph instance with authentic passenger travel demand.

2. Preliminaries

In this chapter we first introduce some general definitions and concepts of graph theory that will be used throughout this work as well as establish definitions specific to our topic. We continue by defining the ride sharing and transit planning problems and conclude the chapter by discussing their similarities.

2.1. Graph Theory

A *graph* $G = (V, E)$ is a tuple of *vertices* V and directed *edges* $E = \{(u, v) \mid u, v \in V\}$. An edge (u, v) may be abbreviated as uv and referred to as an *outgoing edge* of u and an *incoming edge* of v. Moreover, we call v part of the *outgoing neighbourhood* of u and may also refer to it as a *neighbour* of u. Note that, unless stated otherwise, the graphs in this work are directed, without parallel edges and loopless. Moreover, we use $V(G)$ and $E(G)$ to refer to the vertex set and edge set of a graph G respectively. A *subgraph* is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. A *path* P is a graph whose vertices can be seen as a sequence $(v_1, \dots, v_n)$ with $n = |V(P)|$, with edges $v_i v_{i+1}$ for every two consecutive vertices v_i, v_{i+1} and no additional edges. A path *in* a graph G, refers to a subgraph of G, which is a path. We call a path *simple*, if its vertices are pairwise disjoint. A *cycle* is a path with an additional edge $v_n v_1$. Note that paths and cycles are also directed, unless stated otherwise. A *directed acyclic graph* or *DAG*, is a graph which does not contain a cycle as a subgraph. A *topological ordering* of a graph is an ordering of its vertices such that for every edge uv, the vertex u appears before v in the ordering. If such an ordering exists for a graph, it cannot contain cycles and hence is a DAG. We call a vertex of a graph with no incoming edges a *source*. A graph T with only one source r, called the *root*, where for each vertex v there is exactly one path from r to v in T, is called a *directed tree* or just *tree*.

2.2. Flow Networks

A *flow network* is a tuple $F = (G, s, t, c)$ consisting of a graph $G = (V, E)$ with two distinguished vertices $s, t \in V$. We call s the source and t the sink of F. Additionally, F has an capacity function $c : E \longrightarrow \mathbb{N}$. A *flow* on F is a function $f : E \longrightarrow \mathbb{N}$, where $f(e) \leq c(e)$ for all $e \in E$. For a vertex $v \in V$, the *in-flow* is the sum of all incoming flow $f^+(v) = \sum_{uv \in E} f(uv)$ and the *out-flow* is the sum of the outgoing flow $f^-(v) = \sum_{vu \in E} f(vu)$. We call f a *feasible flow*, if it satisfies the *flow conservation* constraint $f^+(v) = f^-(v)$ for all $v \in V \setminus \{s, t\}$. The *value* of a flow $|f| = f^+(t) - f^-(t)$ is the net flow at the sink vertex. A *saturated path* is a path P in F with at least one edge $e \in E(P)$ satisfying $f(e) = c(e)$.

A *weighted flow network* $F = (G, s, t, c, a)$ is a flow network with an additional cost function $a : E \longrightarrow \mathbb{Z}$. The *cost* of a flow f is defined as $a(f) = \sum_{e \in E} f(e) \cdot a(e)$. Given a weighted flow network F and a target flow value k, finding a flow of value k with minimum cost is called the Minimum Cost Flow problem.

2.3. Ride Sharing

There is a plethora of slightly differing ways the ride sharing problem is stated in literature. While most of the core principles are consistent, our upcoming definition adheres most closely to the one used by Masoud and Jayakrishnan [MJ17]. Let $\mathcal{R}$ be a set of *riders* and $\mathcal{D}$ be a set of *drivers* that together make up the set of *participants* $\mathcal{P} = \mathcal{R} \cup \mathcal{D}$. There is also a graph on a set S of *stations* as vertices and a *travel time function* $t : S \times S \longrightarrow \mathbb{N}$ indicating the time it takes to travel from one station to another. This graph is often given as a subgraph of a street network graph. Each participant has a *demand* made up of its *departure station* $s_d \in S$, *arrival station* $s_a \in S$, *earliest departure time* $\tau_{d_{\min}}$ and *latest arrival time* $\tau_{a_{\max}}$. Additionally, there are additional constraints specific to riders and drivers, like the maximum number of times a rider is willing to transfer or the capacity of a driver's vehicle. The goal of the ride sharing problem is to find for each rider an itinerary of rides with drivers, which takes the rider from their departure station to their arrival station in respect to the time- and additional constraints of involved participants, while minimizing a linear combination of their time spent in vehicles, time spent waiting and number of transfers.

2.4. Transit Planning

The definition for the transit planning problem used in this work mostly resembles the Transit Network Design problem (TNDP) combined with a time component akin to the Transit Network Timetabling problem (TNTP), both of which are stated in a review by Guihaire and Hao [GH08]. In our version of the problem we assume the topology of a set S of stations given as a graph $G = (S, E)$ with $E \subseteq S \times S$ and an associated *travel time function* $t_G : S \times S \longrightarrow \mathbb{N}^+$. We will refer to such a graph as a *station graph*. Note that in the context of a station graph, the length of a path in the station graph refers to the sum of the travel times of the edges contained in the path. Additionally, there is a set of passengers $\mathcal{P}$, each passenger $p \in \mathcal{P}$ having a *demand* $\delta(p) = (s_d, s_a, \tau_{a_{\min}}, \tau_{a_{\max}})$ composed of

$$\begin{gathered}\text{departure station } s_d \in S,\\ \text{arrival station } s_a \in S,\\ \text{earliest departure time } \tau_{d_{\min}} \in \mathbb{N} \text{ and}\\ \text{latest arrival time } \tau_{a_{\max}} \in \mathbb{N}.\end{gathered}$$

With these, we define the Transit Network Design and Timetabling problem (TNDTP). The goal is to find a set of itineraries. Each itinerary specifies which stations a vehicle should visit at what time. The solution must satisfy all passenger demands and at the same time, minimize both the number of vehicles needed and the number of overall passenger transfers. If not stated otherwise, we will refer to the vehicles as buses.

Although the typical density of stations in ride sharing might differ from the ones in transit planning scenarios, we can also express their topology as a station graph. The demands of passengers and participants are also alike, but while the TNDTP also seeks to leverage the sharing potential of vehicles along similar routes, there is no distinction of passengers, as with riders and drivers. The buses could be thought of as the equivalent of drivers but they are not part of the passengers, which especially implicates that they have no predetermined departure and arrival constraints in terms of time and station. Additionally, capacities of buses are generally bigger than the ones of the private vehicles provided by drivers.

3. The Detour Approach

In this chapter we introduce our detour-based approach, where the first step is to compute possible detour paths of passengers and represent them as time-expanded graphs. This is inspired by a technique of Masoud and Jayakrishnan [MJ17]. In Section 3.1 we cover their approach in more detail and explain how we adapt it to our setting. Once the possible detours for all passengers are known, we heuristically evaluate which route each passengers should take to maximize the potential of shared bus demands. This is covered in Section 3.2. Afterwards, in Section 3.3, we discuss how the bus demands of all passengers given by the previous step are combined to bus rides. Lastly, in Section 3.4, we determine which bus rides will be served by which bus in which order.

3.1. Detour-DAGs

The goal of this step is to end up with a time-expanded graph for each passenger, that represents all possible detours the respective passenger can take, while still arriving at their destination in time. In this sense, a time expanded graph is based on a station graph G with station set S along with a set $T \subseteq \mathbb{N}$ of points in time and refers to a directed graph $\widehat{G} = (V, E)$ with $V \subseteq S \times T$, where the vertices are tuples of a station and a point in time. Two vertices (s_1, τ_1) and (s_2, τ_2) have a directed edge from (s_1, τ_1) to (s_2, τ_2), if and only if station s_2 can be reached from station s_1 by departing at point in time τ_1 and arriving at point in time τ_2, according to the specified travel time function t_G of the station graph. So formally,

$$E = \{((s_1, \tau_1), (s_2, \tau_2)) \in V \mid t_G(s_1, s_2) = \tau_2 - \tau_1\}.$$

Since we assume to only ever travel forward in time and the travel time of an edge cannot be zero, the time-expanded graph has no cycles, making it a DAG.

3.1.1. The Ellipsoid Spatiotemporal Accessibility Method

Masoud and Jayakrishnan [MJ17] compute similar time-expanded graphs for riders in a ride-sharing scenario. They use a method they call *Ellipsoid Spatiotemporal Accessibility Method (ESTAM)*. As we already discussed in Section 2.4, the domain of transit planning shares a lot of concepts with the ride sharing scenario, which is why we can adopt their approach of finding detours for participants with some small adjustments for the passengers of the TNDTP.

They start of with a graph of stations which represent the locations where drivers can start and end their trips as well as stop to pick up riders. This graph can be interpreted as a station graph G. In their first step, for each rider r, they compute a reduced graph G_r from G that only contains stations that are spatio-temporally reachable for r. Whether a station s is spatio-temporally reachable is determined by calculating the euclidean distance between s and the departure station of r and comparing it to an overestimation $\hat{d}$ of how far r can travel, while still arriving at their destination in time. This can be pictured as reducing G to the stations inside an ellipse of transverse diameter $\hat{d}$ with its focal points being the riders departure and arrival station. In the second step of the ESTAM, they further reduce the graphs G_r by removing the stations that can not be covered by any driver. Since we do not have any departure or arrival station constraints for our equivalent of drivers, the buses, there is nothing we

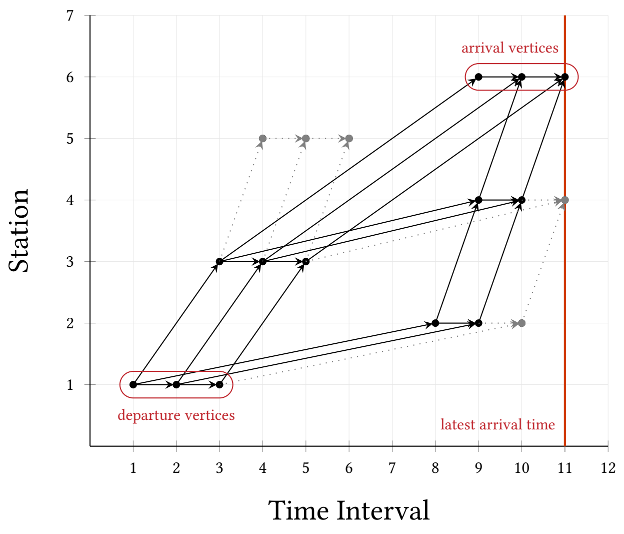


Figure 3.1.: Example of a time-expanded graph created by the ESTAM. The dotted edges and greyed-out vertices are not part of the final graph, since they do not lie on a path from a departure vertex to an arrival vertex.

adapt from this step. The third step is where the time-expansion happens. At this point it is important to note how their definition of time-expanded graphs differs from ours. Together with stations, Masoud and Jayakrishnan use time intervals of uniform length for the vertices and also allow "waiting" edges, which are edges between vertices where the station is the same but the time interval is incremented. In our definition, the intervals are not explicitly given. Instead, a vertex contains a point in time together with the station, which we use to model that the passenger can reach the station by this point in time. We use this modified definition to keep the time expanded graph smaller, while still keeping all of the information. To explain how we can still derive the intervals with our definition, we first need to continue describing the ESTAM.

From each reduced graph G_r they create a time-expanded graph by first finding all paths in G_r from the departure station to the arrival station of r, which have a travel time within the travel time constraints of r. Then, they use this information to determine in which time intervals r can be at which station and create their time-expanded graph. Since there may be multiple vertices that contain the departure or arrival station, these vertices are kept in departure vertex and arrival vertex sets respectively. Figure 3.1 shows an example for a graph created by the ESTAM, where these two sets are marked. There also may be vertices that do not lie on a path from a departure vertex to an arrival vertex. Those can be deleted, since the rider would not be able to reach their destination by going there. The incoming edges of these vertices are represented as dotted lines in the example seen in Figure 3.1.

3.1.2. Detour-DAG Definition

Like mentioned before, we use time-expanded graphs to express that a passenger can reach a station by a given point in time. Implicitly, the station can also be reached at any later point in time by just waiting at the station. So there is no need for waiting edges with this approach. Stations can still appear in multiple vertices of the time-expanded graph when there are multiple paths in the station graph to reach a station. Note, that while there can be multiple vertices in the time-expanded graph that contain the arrival station, there will only ever be one vertex containing the departure station. For a passenger $p \in \mathcal{P}$ with a given demand $\delta = (s_d, s_a, \tau_{d_{\min}}, \tau_{a_{\max}})$, we call such a time-expanded graph based on a station graph G with stations S and points in time T a *detour-DAG* $D(\delta) = (V, E)$ with

$$\begin{aligned} V &= \{(s,\tau) \in S \times T \mid \tau_{d_{\min}} + t_G(s_d, s) + t_G(s, s_a) \leq \tau_{a_{\max}}\} \text{ and} \\ E &= \{((s_1,\tau_1)(s_2,\tau_2)) \in V \times V \mid (s_1,s_2) \in E(G) \text{ and } \tau_2 = \tau_1 + t_G(s_1,s_2)\}. \end{aligned}$$

The vertex that contains the departure station of the demand is sometimes referred to as its *origin* and the vertices containing the arrival station are kept in its *arrival vertex set*

$$V_a = \{(s,\tau) \in V \mid s = s_a\}.$$

Similar to the final time-expanded graphs of the ESTAM, a detour-DAG only contains vertices that lie on a path from its origin to any of its arrival vertices, so that only actual detours, where the arrival station can be reached in time, are represented. We call s_1 a *possible predecessor station* of s_2 for p, if and only if there is an edge $(s_1,\tau_1)(s_2,\tau_2) \in E(D(\delta))$. See Figure 3.2 for an example of a detour-DAG on the same station graph and with the same demand as the example for the time-expanded graph in Figure 3.1 created with the ESTAM. When comparing these figures we can observe that their graph essentially consists of multiple copies of our detour-DAG which are connected by waiting edges. Since we do not have these waiting edges, our graph can be smaller. However, the information of how long the passenger could wait at a station is not lost. From the detour-DAG of the passenger and the time constraints of its demand, it is possible to derive for each station the time intervals in which the passenger can be at the station, while still able to reach their arrival station in time. To that end, we call $T_I^p(s_1,s_2) = [\tau'_{a_{\min}}, \tau'_{a_{\max}}]$ the *implied arrival time intervals* of p for s_2 coming from s_1, with

$$\begin{aligned} \tau'_{a_{\min}} &= \min(\tau_a \mid (s_1,\tau_d)(s_2,\tau_a) \in E(D(\delta))) \text{ and} \\ \tau'_{a_{\max}} &= \tau^{\delta}_{a_{\max}} - t_G(s_2, s^{\delta}_a). \end{aligned}$$

The start points of the intervals are already given by the vertices of the detour-DAG. The arrival vertices also already have an end point given by the time constraints of the demand. These end points can now be propagated to the predecessor vertices by subtracting the travel time associated with the respective edge. This can be repeated until the origin is reached. A visualization of the procedure is shown in Figure 3.3.

3.1.3. Detour-DAG Computation

To compute a detour-DAG for a given passenger demand, we proceed in two steps: First, we create a time expanded graph by exploring the station graph from the departure station of the demand. Afterwards, we reduce this graph to a detour-DAG.

The exploration is essentially a modified depth-first search. A station is explored by exploring all its unvisited neighbours recursively. However, if the exploration of a station is finished, it is un-marked again. This ensures that it can be explored coming from a different predecessor again. Figure 3.4 shows some of the exploration states in an exemplary execution of Algorithm 3.1, with the street graph G on the left and the associated time-expanded graph $\widehat{G}$ on the right. The currently marked vertices are indicated by a bold outline and the order of exploration is marked by arrows. For simplicity, the street graph is drawn as an undirected graph with the travel time given as edge labels. For this example, the earliest departure of the passenger is at time 1 and the latest arrival is at time 11. In Figure 3.4a the exploration starts with s_1 and continues to s_2. Due to the travel time, the passenger could reach s_1 by time 6 and s_2 by time 10, following this route. As the route is explored, the arrival information is

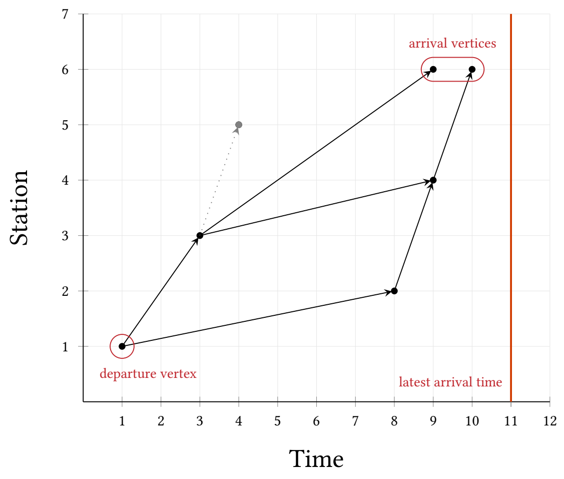


Figure 3.2.: Example of a detour-DAG. The dotted edge and greyed-out vertex is not part of the final graph, since they do not lie on a path from the departure vertex to an arrival vertex.

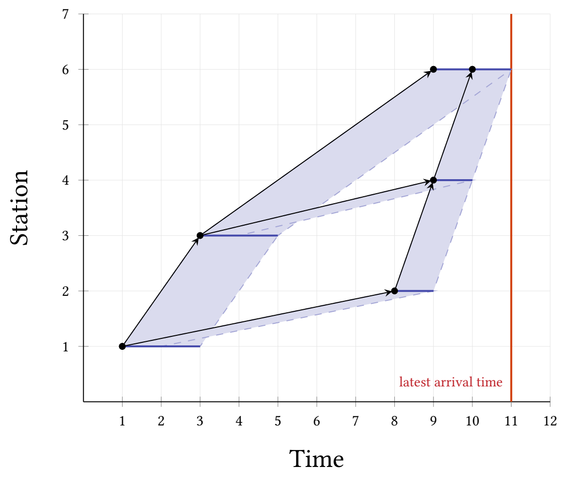
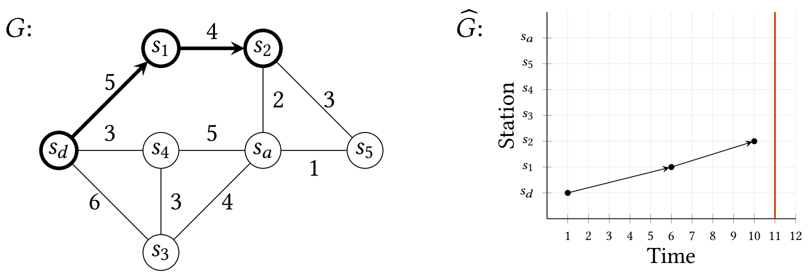


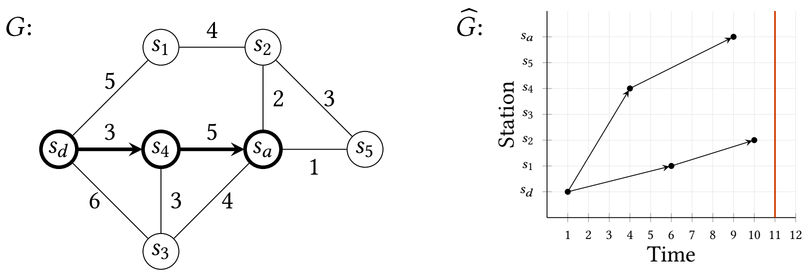
Figure 3.3.: Visualization of the back-propagation of visit interval end points through a detour-DAG.

Algorithm 3.1: ExploreDetours

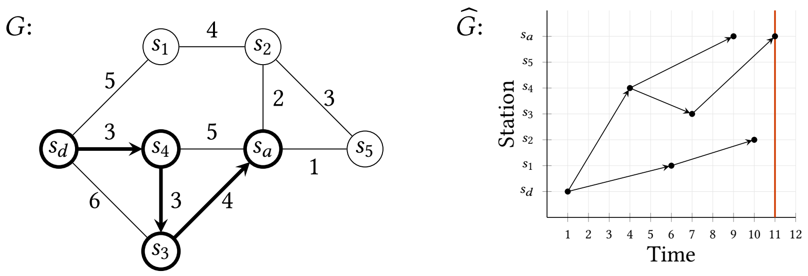
```
Input: Station graph $G$, demand $\delta = (s_d, s_a, \tau_{d_{\min}}, \tau_{a_{\max}})$
Output: Time-expanded graph $\widehat{G}$, arrival vertex set $V_a$
marked($s$) ⟵ false, for all $s \in V(G)$
marked($s_d$) ⟵ true
add vertex $(s_d, \tau_{d_{\min}})$ to $\widehat{G}$
exploreVertex($(s_d, \tau_{d_{\min}})$)
Procedure exploreVertex($(s, \tau)$)
    forall neighbours $s'$ of $s$ do
        $\tau' \longleftarrow \tau + t_G(s, s')$
        if not marked($s'$) and $\tau' \leq \tau_{a_{\max}}$ then
            add vertex $(s', \tau')$ to $\widehat{G}$, if not already contained
            add edge $((s, \tau), (s', \tau'))$ to $\widehat{G}$
            if $s' = s_a$ then
                add vertex $(s', \tau')$ to $V_a$
            else
                marked($s'$) ⟵ true
                exploreVertex($(s', \tau')$)
                marked($s'$) ⟵ false
```


G:
s_1 s_2 s_d s_4 s_a s_5 s_3
4 5 2 3 3 5 1 6 3 4
$\widehat{G}$:
Station
s_a s_5 s_4 s_3 s_2 s_1 s_d
Time
1 2 3 4 5 6 7 8 9 10 11 12

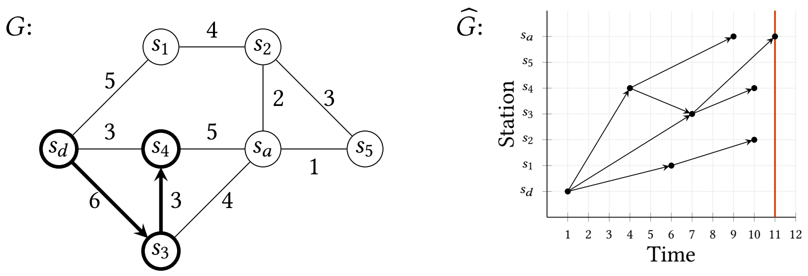
(a) The exploration branch terminates at s_2, since any subsequent arrival would be too late.


G:
s_1 s_2 s_d s_4 s_a s_5 s_3
4 5 2 3 3 5 1 6 3 4
$\widehat{G}$:
Station
s_a s_5 s_4 s_3 s_2 s_1 s_d
Time
1 2 3 4 5 6 7 8 9 10 11 12

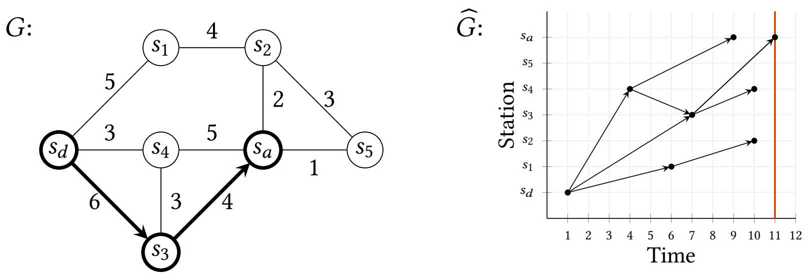
(b) The exploration branch reaches s_a, thus terminating there.


G:
s_1 s_2 s_d s_4 s_a s_5 s_3
4 5 2 3 3 5 1 6 3 4
$\widehat{G}$:
Station
s_a s_5 s_4 s_3 s_2 s_1 s_d
Time
1 2 3 4 5 6 7 8 9 10 11 12

(c) Another path to s_a was explored. The arrival time is later but still not after $\tau_{a_{\max}}$.


G:
s_1 s_2 s_d s_4 s_a s_5 s_3
4 5 2 3 3 5 1 6 3 4
$\widehat{G}$:
Station
s_a s_5 s_4 s_3 s_2 s_1 s_d
Time
1 2 3 4 5 6 7 8 9 10 11 12

(d) Another path to s_3 arriving at time 6 was discovered. The exploration branch terminates at s_4.


G:
s_1 s_2 s_d s_4 s_a s_5 s_3
4 5 2 3 3 5 1 6 3 4
$\widehat{G}$:
Station
s_a s_5 s_4 s_3 s_2 s_1 s_d
Time
1 2 3 4 5 6 7 8 9 10 11 12

(e) Everything explored here is already known, so nothing new is added.

Figure 3.4.: Visualization for an exemplary execution of Algorithm 3.1 for passenger demand $(s_d, s_a, 1, 11)$.

added to the time-expanded graph. Since reaching any of the neighbours of s_2 would take the passenger there after their latest arrival, this branch of the exploration terminates at s_2. As a consequence, s_2 is un-marked. The exploration of s_1 is also finished for now, as it has no other un-marked neighbours, so it is un-marked as well. This leads us to the state seen in Figure 3.4b, where s_4 was chosen as the next station to explore, followed by s_a. Since this is the arrival station of the passenger, this branch of the exploration can be terminated as well. Station s_3 is explored next, through which s_a can be reached again. Figure 3.4c shows this state. By this route, the passenger would arrive at time 11 at their arrival station, which is also still in time. Hence, there are now two vertices in the time-expanded graph that represent station s_a at different points in time. The exploration branch shown in Figure 3.4d finds a new route to s_3 arriving at the same time as before. Note, that this leads to a new edge in the time-expanded graph. Afterwards, as seen in Figure 3.4e, the remaining neighbour of s_3 is explored. In this case, this does not lead to new information, so the time-expanded graph does not change. The exploration is formally described by Algorithm 3.1.

To show the correctness of this procedure we first introduce terminology that makes it easier to characterize the results of the algorithm. For the purpose of this section, we assume $\widehat{G}$ to be the graph and V_a the vertex set computed by Algorithm 3.1 from street graph G and demand $\delta = (s_d, s_a, \tau_{d_{\min}}, \tau_{a_{\max}})$. Furthermore, we call a simple directed path with a length less than or equal to $\tau_{a_{\max}} - \tau_{d_{\min}}$ starting in s_d, which does not contain s_a except as the last vertex an *open δ-detour path*. If the last vertex of an open detour path is s_a, we call it a *proper δ-detour path*, else we refer to it as an *improper δ-detour path*. Like mentioned before and as is also apparent in lines 14 to 16 of Algorithm 3.1, stations are marked as they are explored and un-marked once their exploration is finished, to allow them to be explored coming from different predecessors later. This leads us to Observation 3.1.

Observation 3.1: *A vertex $s \in V(G)$ is marked if and only if,* `exploreVertex` *for a vertex (s, τ) is currently executed.*

As it is possible to reach the same station through multiple routes at the same point in time, like in the example seen before, the graph added to the time-expanded graph by the exploration of a station is not necessarily a tree, like with a regular depth-first search. Nevertheless, there is some structure to it. Since an edge always points forward in time and we always explore from a single station, we end up exploring a DAG with a single source in each recursive exploration step, which is shown in Lemma 3.2.

Lemma 3.2: *The graph containing a vertex $(s, \tau) \in V(\widehat{G})$ and all vertices and edges added to $\widehat{G}$ by an execution of* `exploreVertex` *for (s, τ) is a DAG with unique source (s, τ).*

Proof. In an execution of `exploreVertex`, vertices and edges can only be added to $\widehat{G}$ in line 9, line 10 and by the recursive execution of `exploreVertex` in line 15. Since with every recursive call another vertex of G is marked and recursive calls are only made for unmarked vertices of G, there will eventually be no additional recursive call. In the following, we will use this as the base case for an inductive argument.

Note, that for all edges $((s_1, \tau_1), (s_2, \tau_2))$ added to $\widehat{G}$, $\tau_1 < \tau_2$ applies. This yields a topological ordering for all vertices of $\widehat{G}$ and hence, $\widehat{G}$ is a DAG in any moment of execution of Algorithm 3.1. So it remains to show that (s, τ) is its unique source. In the base case of the induction, there is no further recursive call since all neighbours are already marked. In this case there is also nothing added to $\widehat{G}$, hence the statement holds. Now, we assume that the statement holds for the recursive calls made in an execution of `exploreVertex` for a vertex (s, τ). Since we add the edge $((s, \tau), (s', \tau'))$ for all vertices (s', τ') for which the recursive call is made in line 10, the vertex (s, τ) becomes the distinct source for the graph added to $\widehat{G}$ in this execution of `exploreVertex`. ∎

Hence, we can explore all possible paths starting from the departure station. However, if the exploration reaches the destination station or a station that can not be reached within the current time constraints, we do not explore this station further. Therefore, the time-expanded graph created by

Algorithm 3.1 contains exactly all open δ-detour paths, which we show in Theorem 3.3. To simplify its proof we introduce the concept of a *path expansion*, which is the representation of the time-expanded version of a path from the station graph in the time-expanded graph. So for a path P, given as the sequence $(s_1, \dots, s_n)$, $n \in \mathbb{N}$, in a station graph G with travel time function t_G, we call $\mathcal{E}(P) = (V', E')$ its path expansion with

$$V' = \{(s_i, \tau_i) \mid i \leq n, \tau_i = \tau_{d_{\min}} + \sum_{j=2}^{i} t_G(s_{j-1}, s_j)\} \text{ and}$$
$$E' = \{((s_i, \tau_i), (s_j, \tau_j)) \mid s_i s_j \in E(G)\}.$$

Further, for $i \leq n$, let $\mathcal{E}_P(s_i) = (s_i, \tau_i)$, such that $(s_i, \tau_i) \in V'$ is the vertex in the path expansion of P containing s_i.

Theorem 3.3: *$\widehat{G}$ contains exactly the expansion of every open δ-detour path in G.*

Proof. First, we will show that $\widehat{G}$ contains the expansion of every open δ-detour path in G. Assume there is an open detour path P in G, whose expansion $\mathcal{E}(P)$ is not contained in $\widehat{G}$. Without loss of generality, let P be the shortest such path. Note that this ensures that the path expansion of every open detour path, which is a sub-path of P, is contained in $\widehat{G}$. Let s' be the last and s be the second last vertex of P and let P_s be the path obtained by deleting s' from P. Since $\mathcal{E}(P_s)$ is contained in $\widehat{G}$, we know that $\mathcal{E}_{P_s}(s)$ was added to $\widehat{G}$ in line 9 of Algorithm 3.1. From P being an open detour path follows $s \neq s_a$ and hence, $\mathcal{E}_{P_s}(s)$ is explored subsequently, resulting in the loop in line 6 being executed for s', as it is a neighbour of s in G. Since P cannot have a length greater than $\tau_{a_{\max}} - \tau_{d_{\min}}$, the condition $\tau' \leq \tau_{a_{\max}}$ will hold at that point. Additionally, s' must be un-marked or else, with Observation 3.1 and Lemma 3.2, there would be a vertex (s', τ) somewhere in $\mathcal{E}(P_s)$, which contradicts P being simple. Therefore, s' is un-marked, $\mathcal{E}_P(s')$ is added to $\widehat{G}$ along with the edge from $\mathcal{E}_{P_s}(s) = \mathcal{E}_P(s)$ to $\mathcal{E}_P(s')$ and thus $\mathcal{E}(P)$ is contained in $\widehat{G}$.

Now, we will show that for any vertex of $\widehat{G}$ there is an open δ-detour path in G, whose path expansion contains that vertex. Let (s, τ) be a vertex of $\widehat{G}$. It follows from Lemma 3.2, that (s, τ) is contained in a DAG with $(s_d, \tau_{d_{\min}})$ as the unique source. Thus, there is a directed simple path $\widehat{P}$ from $(s_d, \tau_{d_{\min}})$ to (s, τ) in $\widehat{G}$. Note that line 8 ensures $\tau \leq \tau_{a_{\max}}$. Additionally, s_a can only be part of a vertex of $\widehat{P}$ if $s = s_a$, so as the last vertex, since the recursive call does not happen for s_a and thus, a vertex containing s_a will never have outgoing edges. For every edge $(s_i, \tau_i)(s_j, \tau_j)$ in $\widehat{P}$ there must be a corresponding edge $s_i s_j$ in G, since only such edges are added in line 10. Therefore, there must be a path P in G starting from s_d with $\mathcal{E}(P) = \widehat{P}$. Since the constraints concerning $\tau_{a_{\max}}$ and s_a also translate from $\widehat{P}$ to P, it is an open δ-detour path. ■

The remaining step is to reduce $\widehat{G}$ to the detour-DAG D, by removing all improper δ-detour paths which leaves us with only the proper δ-detour paths being represented in D. Figure 3.5 visualizes the reduction for the result of the example from Figure 3.4. This is done by executing a depth first search in the graph obtained by flipping all edges in $\widehat{G}$. The search then starts from the arrival vertices in V_a. All vertices that were not discovered by the search are deleted, which discards all vertices that do not lie on a path from the origin to an arrival station.

Note, that there is an improvement to Algorithm 3.1, that prevents the exploration of some improper detour paths in the first place. It is sufficient to only explore a station s', if it is still possible to reach the arrival station in time, after arriving there. Changing the time-constraint in line 8 to $\tau' + t_G(s', s_a) \leq \tau_{a_{\max}}$ will ensure this. However, this does not necessarily prevent the exploration of all improper detour paths, as all paths to reach the arrival station from s' in time could go through a station that is marked in the current exploration state.

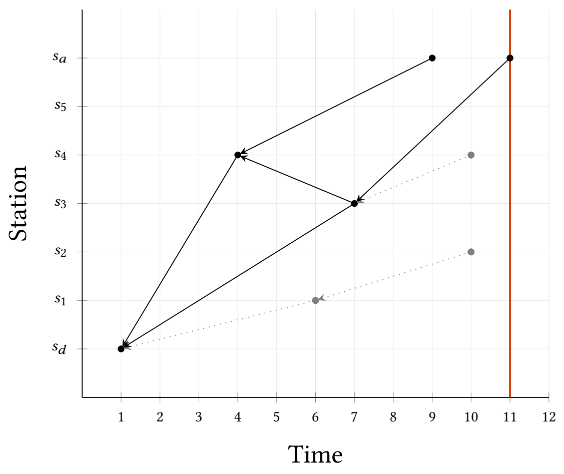


Figure 3.5.: Reduction of the time-expanded graph obtained in Figure 3.4. The search starts at the vertices of station s_a and follows the edges flipped edges to the origin. The vertices not reached by that are indicated in grey and are not part of the final detour-DAG.

There is another detail about our actual implementation of the exploration that we want to mention here. Different than described in the pseudocode, each time we add a vertex to the time-expanded graph in line 9, a new instance of this vertex is added to the actual data structure. As a consequence, every execution of `exploreVertex` adds a directed tree to the time-expanded graph instead of a DAG. As a tree is also a DAG with only one source, this does not influence the correctness. This adding of duplicate vertices has the advantage that we can store the final detour-DAG as a list of arrival vertex copies and a list of predecessors, where each vertex has exactly one predecessor. This makes it efficient to traverse the detour-DAG backwards, as we will do it in the following step in order to use the implied arrival time intervals.

The time-complexity of the time-expanded graph computation is linear in the output size, since we stop the exploration of a vertex as soon as the time-constraints are not met any more. In theory, the number of simple paths in the station graph starting from the departure node and meeting the time-constraints could be exponential in the length of the shortest path. However, realistic station graph instances are likely to be thin graphs and realistic time-constraints of a passenger demand do not allow for an exploration of the whole graph. How the detour-DAG computation behaves with realistic instances is discussed in Chapter 5. The reduction of the time-expanded graph can be done in linear time in respect to the size of the resulting detour-DAG.

3.2. Bus Demands

The goal of this step is to use the information about all the possible detours given by the detour-DAGs to compute the demand for single bus rides between station pairs. So essentially, the detour-DAG we computed in the previous step from the demand $\delta = (s_d^\delta, s_a^\delta, \tau_{d_{\min}}^\delta, \tau_{a_{\max}}^\delta)$ of a passenger p is now used to compute a set of bus ride demands for p. A single bus ride demand is a tuple $(s_d, s_a, \tau_{a_{\min}}, \tau_{a_{\max}})$, with

$$\begin{gathered}\text{departure station } s_d \in S,\\ \text{arrival station } s_a \in S,\\ \text{earliest arrival time } \tau_{d_{\min}} \in T \text{ and}\\ \text{latest arrival time } \tau_{a_{\max}} \in T.\end{gathered}$$

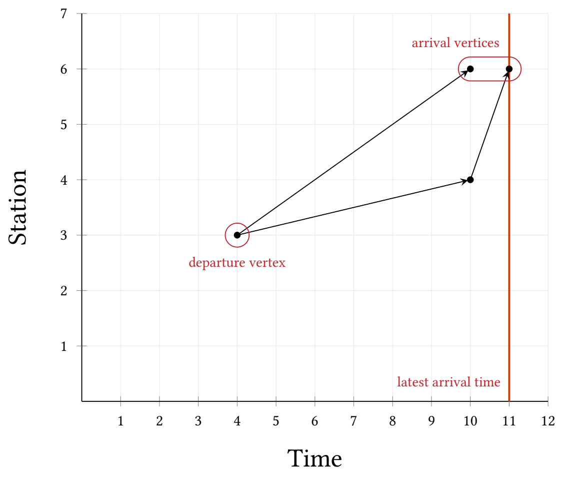
The earliest and latest arrival time indicate the *arrival time interval* $[\hat{\tau}_{a_{\min}}, \tau_{a_{\max}}]$ of p at the arrival station. This definition is similar to the regular passenger demand. Note, however, that each passenger can have multiple associated bus ride demands, which together form their route from s_d^δ to s_a^δ. The set of all bus ride demands of p is denoted by $\Delta_B(P)$ and we refer to the union of all bus ride demands as $\Delta_B(\mathcal{P})$.

In order to compute the demand, we need to collapse all of the possible routes represented by a detour-DAG to a single route, which the corresponding passenger will actually take. We do this greedily, by starting at the destination of a passenger and determining which predecessor station the passenger must come from, to maximise the number of other passengers that could potentially also take that ride. The actual routes taken by the other passengers are not considered by this approach, but only the sharing potential inferred from the detour-DAGs. This way, the order of passengers, for which we compute the demand, does not matter and it could be computed in parallel. Additionally note, that once the decisions for a predecessor is made, it is not changed again. We continue by determining the next predecessor and repeating the process until the origin is reached. Refer to Figure 3.2 for a simple example and to Algorithm 3.2 for a detailed description of the procedure. The backwards traversal and the way the earliest arrival is updated in line 16, ensures that the earliest arrival between two subsequent stations for a passengers are consistent with the time it takes to travel between these stations, which we formalise in Observation 3.4.

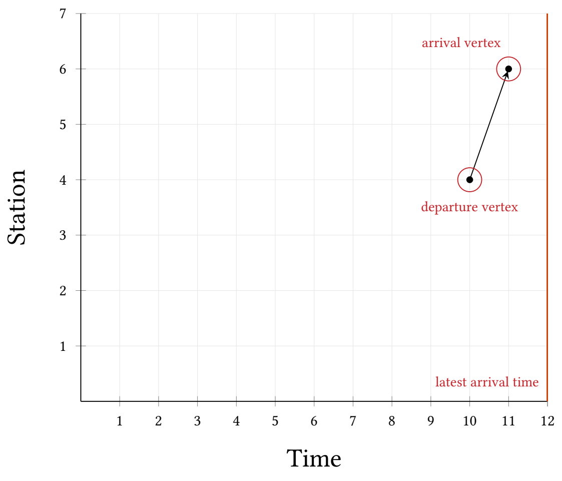
Observation 3.4: *For every two demands $(s_1, s_2, \tau_{a_{min}}, \tau_{a_{max}}), (s_2, s_3, \tau'_{a_{min}}, \tau'_{a_{max}}) \in \Delta_B(p)$, computed by Algorithm 3.2 for an arbitrary passenger $p \in \mathcal{P}$, the inequality $\tau_{a_{min}} \leq \tau'_{a_{min}} + t_G(s_2, s_3)$ holds.*

There are a few things we still need to address. The first of which is how the predecessor, maximizing the sharing potential, is determined in each iteration. We do this by first finding the *maximal passenger interval* for each predecessor. Consider a passenger p and their arrival time interval $[\tau_{a_{\min}}, \tau_{a_{\max}}]$ at s_2, coming from s_1. The respective maximal passenger interval is the interval $[\hat{\tau}_{a_{\min}}, \hat{\tau}_{a_{\max}}] \subseteq [\tau_{a_{\min}}, \tau_{a_{\max}}]$, which maximises the number of passengers with an arrival time interval at s_2, coming from s_1, containing it. To find it, we do a sweep over the start- and end-points, of all arrival intervals of the station pair (s_1, s_2), sorted by time. When processing an interval start- or end-point, we keep track of the currently best result and how it is influenced by this interval point. After the maximal passenger interval is found for every predecessor, the once covering the most passengers is chosen.

The second thing is, that with each iteration, we potentially further restrict the arrival time interval. If the interval start point changes, we get a situation like shown in Figure 3.8, where a passenger arrives at a station later than previously assumed. The problem here is, that we propagate the interval changes in the algorithm only backwards in time. To solve this, we keep track of the time by which we would move the interval start point and clip all intervals only at the end.



(a)



(b)

Figure 3.6.: Two additional examples of detour-DAGs

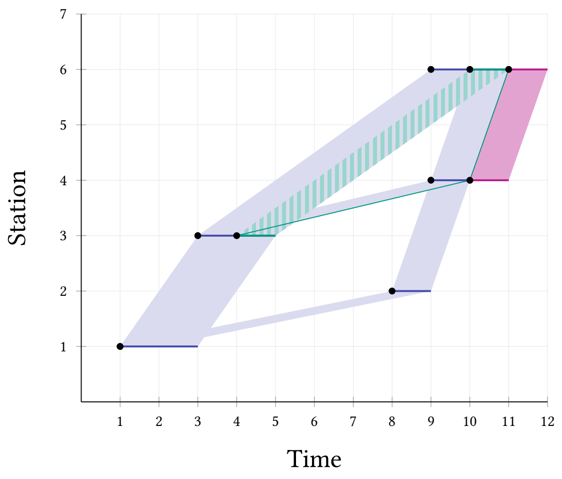


Figure 3.7.: Visualization of the overlaid implied intervals of the detour-DAGs from Figure 3.2 in blue, Figure 3.6a in green and Figure 3.6b in purple. Let us attribute the blue detour-DAG to a passenger p_1, the green one to p_2 and the purple one to p_3. Consider for example, that we want to determine for p_1 how they reach Station 6. Note that the detour-DAG is traversed backwards in time, so we follow the directed edges in reverse. Passenger p_1 could come from either Station 3 or Station 4. For both possible predecessor stations we determine the time interval where p_1 could arrive at Station 6 coming from the respective station, where the most other passengers could also travel to Station 6 from that station. So for Station 3 as the predecessor station, that is $[10, 11]$, containing p_2 alongside p_1. For Station 4, we get $[11, 11]$, containing p_2, p_1 as well as p_3. Note, that it is not determined whether p_2 travels from Station 3 or Station 4 to Station 6, nor is any other passenger committed to a specific route by this step. It is just about the number of passengers, potentially sharing the ride between stations. In this case, the sharing potential with Station 4 is higher than with Station 3 and thus, we would create a bus ride demand for p_1 from Station 4 to Station 6 arriving in $[11, 11]$. This procedure is then repeated to determine how p_1 reaches Station 4 and so on.

Algorithm 3.2: ComputeDemandIntervals

```
Input: Set $\mathcal{D}$ of detour-DAGs
Output: Bus ride demand $\Delta_B(p)$ sorted by $\tau_{a_\mathrm{min}}$ for passengers $p \in \mathcal{P}$
forall detour-DAGs $D$ of $\mathcal{D}$ do
    $\tau_{a_\mathrm{min}} \leftarrow$ earliest arrival in $D$
    $\tau_{a_\mathrm{max}} \leftarrow$ latest arrival in $D$
    $s \leftarrow$ arrival station of $D$
    $p \leftarrow$ passenger of $D$
    $t_\mathrm{clip} \leftarrow 0$
    while $s$ is not arrival station in $D$ do
        $[\hat{\tau}_{a_\mathrm{min}}, \hat{\tau}_{a_\mathrm{max}}] \leftarrow \perp$
        $\hat{s}' \leftarrow \perp$
        forall predecessors stations $s'$ of $s$ in $D$ do
            $t \leftarrow \mathcal{D}$.maxPassengerInterval($s, s', \tau_{a_\mathrm{min}} + t_\mathrm{clip}, \tau_{a_\mathrm{max}}, p$)
            if $t$ covers more passengers than $[\hat{\tau}_{a_\mathrm{min}}, \hat{\tau}_{a_\mathrm{max}}]$ then
                $[\hat{\tau}_{a_\mathrm{min}}, \hat{\tau}_{a_\mathrm{max}}] \leftarrow t$
                $\hat{s}' \leftarrow s'$
        $t_\mathrm{clip} \leftarrow \hat{\tau}_{a_\mathrm{min}} - \tau_{a_\mathrm{min}}$
        $\tau_{a_\mathrm{min}} \leftarrow \tau_{a_\mathrm{min}} - t_G(\hat{s}', s)$
        $\tau_{a_\mathrm{max}} \leftarrow \hat{\tau}_{a_\mathrm{max}} - t_G(\hat{s}', s)$
        add $(s', s, \tau_{a_\mathrm{min}}, \tau_{a_\mathrm{max}})$ to $\Delta_B(p)$
        $s \leftarrow \hat{s}'$
    forall $(s_d, s_a, \tau'_{a_\mathrm{min}}, \tau'_{a_\mathrm{max}})$ in $\Delta_B(p)$ do
        $\tau'_{a_\mathrm{min}} \leftarrow \tau'_{a_\mathrm{min}} + t_\mathrm{clip}$
```

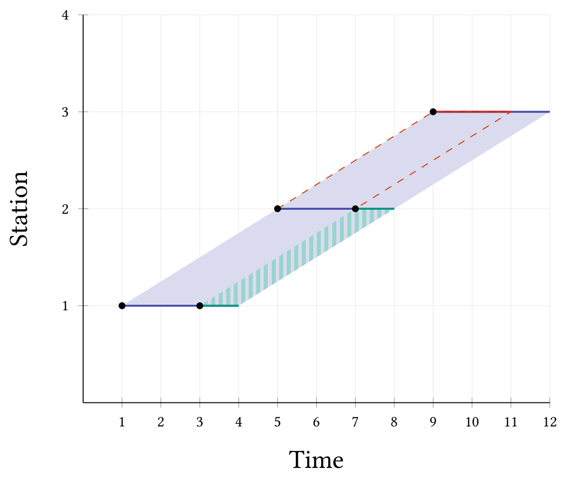


Figure 3.8.: Visualization of implied intervals of two passengers. The interval containing the most other passengers for p_1, marked in blue, at Station 2 is $[7, 8]$, since it might be able to share the ride with the green passenger. That moves the earliest arrival of p_1 at Station 2 from 5 to 7. But when p_1 arrives at time 7, they can not take any bus before that, hence the earliest arrival at Station 3 changes as well and the according interval must be clipped accordingly, which is marked in red.

Furthermore, we want to cover a few implementation details. To know which are the possible predecessor stations of a station for a passenger, we can build a predecessor station map from a detour-DAG in advance, by traversing it backwards from the arrival vertices. When we traverse an edge $(s_1, \tau_1)(s_2, \tau_2)$, we know that s_1 is a possible predecessor of s_2 and can add that information to the predecessor stations map. To efficiently sweep over the arrival intervals when finding the maximal passenger intervals, we can also build a map from the detour-DAGs in advance. It will map each pair of stations to a list of arrival time start and end-points sorted by time. We will refer to this map as the *overlaid detour-DAG*. If a passenger has multiple arrival intervals at a station, it suffices to insert the earliest start-point and the latest end-point, since a passenger can always just wait at the station.

Lastly, we discuss the computation time. In the overlaid detour-DAG, every station pair will map to a maximum of $|\mathcal{P}|$ interval start- and end-points each. Since every detour-DAG edge contributes to exactly one interval, the overlaid detour-DAG can be built in $O(|E(\mathcal{D})| \cdot \log |\mathcal{P}|)$ time by traversing the detour-DAGs backwards whilst inserting the boundaries of the implied intervals in the respective set. To find a maximal passenger interval, we have to sweep over $|\mathcal{P}|$ elements, in the worst case. Since the maximal passenger interval may have to be computed for every detour-DAG edge, the worst-case time-complexity is $O(|E(\mathcal{D})| \cdot |\mathcal{P}|)$. In reality however, instances probably result in overlaid detour-DAGs where the size of the sorted sets is a lot smaller than $|\mathcal{P}|$. Assuming the size of the largest sorted set associated with a station pair has size $\hat{T}$, the overlaid detour-DAG can be built in $O(|E(\mathcal{D})| \cdot \log |\hat{T}|)$ and the overall time complexity is $O(|E(\mathcal{D})| \cdot |\hat{T}|)$.

3.3. Bus Rides

The bus ride demand obtained from the previous step indicates for each passenger, between what station pairs they need a bus ride arriving in which time interval at the arrival station. So for a given station pair, some of the associated arrival time intervals may overlap, indicating that the corresponding passengers could share a ride between this station pair. So in this step, we want to compute the actual bus rides from the passengers' bus ride demand, while minimizing the total number of rides needed by leveraging overlapping demands. A *bus ride* is a tuple $(s_d^b, s_a^b, \tau_d^b, \tau_a^b, P^b)$ with

$$\begin{aligned} \text{departure station } s_d^b &\in S, \\ \text{arrival station } s_a^b &\in S, \\ \text{departure time } \tau_d^b &\in T, \\ \text{arrival time } \tau_a^b &\in T \text{ and} \\ \text{passenger set } P^b &\subseteq \mathcal{P}. \end{aligned}$$

The general idea is shown in Figure 3.9 using an example. We sweep over the start- and end-points of all of the bus ride demand arrival intervals and pack the corresponding passengers in a bus ride between the corresponding stations until an interval of one of the contained passengers ends, meaning that the bus needs to arrive by that time. Then, this ride would be finished and a new one is created for the concerned station pair. So in the example, we can pack p_1, p_2 and p_3 in the same bus ride, since they all have a demand from station s_1 to station s_2 with an overlapping arrival time frame and the latest arrival of p_3 dictates the actual arrival time of that bus ride.

However, there are cases where this simple procedure is not sufficient. Consider Figure 3.10. There, it leads to a problem that the arrival time of b_1 is some time later than the earliest arrival of the passengers contained in the bus ride, since that implies that these passengers also arrive the same amount of time after their earliest arrival at their next station. If this is not taken into account, passengers could be

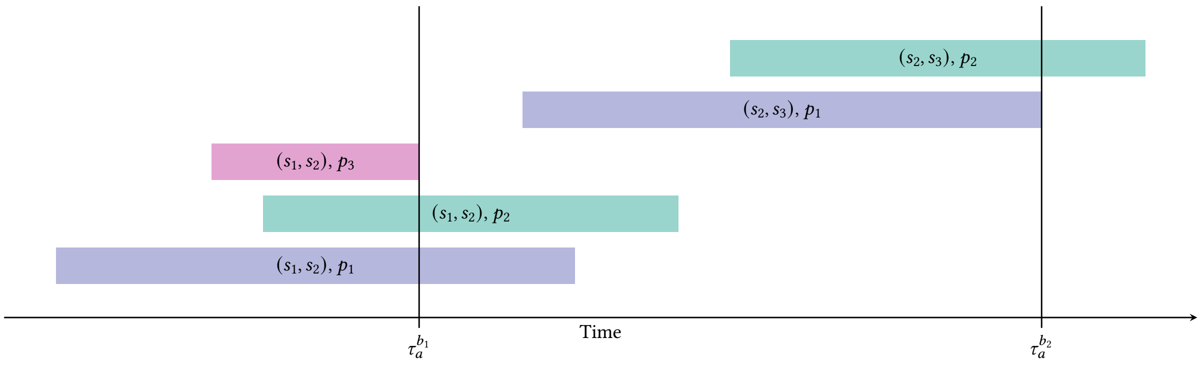


Figure 3.9.: Example for the assignment of passengers to bus rides using their bus ride demand intervals. Bus ride b_1 will arrive at $\tau_a^{b_1}$ and contain passengers p_1, p_2 and p_3, while b_2 will arrive at $\tau_a^{b_2}$ containing p_1 and p_2.

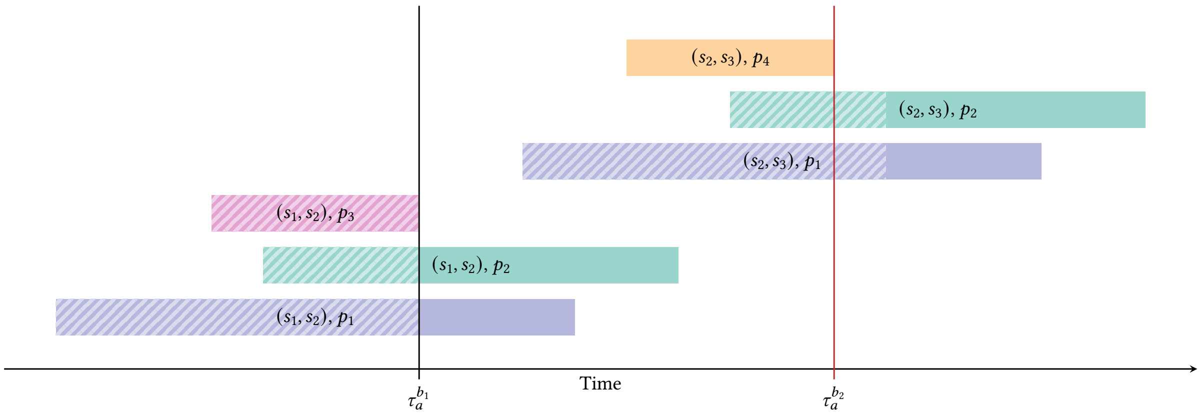


Figure 3.10.: Example for the assignment of passengers to bus rides using their bus ride demand intervals. As soon as the arrival time of bus ride b_1 is determined, the remaining intervals of the containing passengers need to be shortened, which is indicated by the striped area.

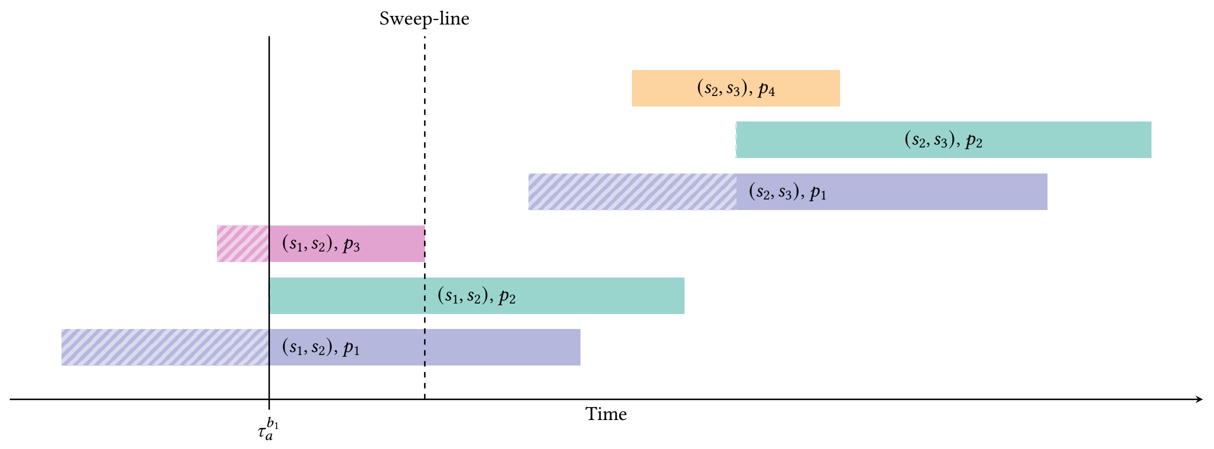


Figure 3.11.: Example for the assignment of passengers to bus rides using their bus ride demand intervals with the improved strategy of choosing the bus ride arrival time.

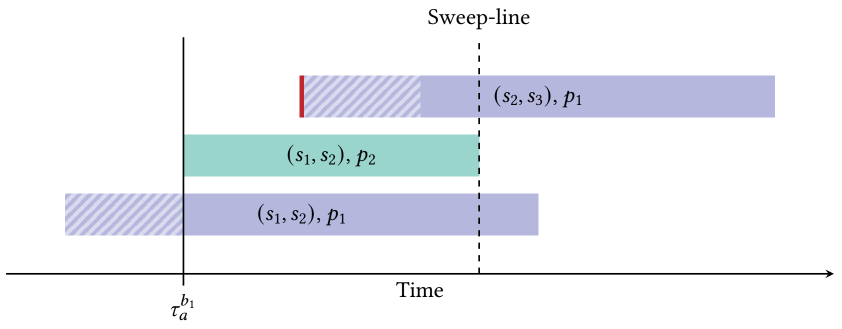


Figure 3.12.: Example for the assignment of passengers to bus rides using their bus ride demand intervals. When the arrival time of bus ride b_1 is determined, the start-point of the interval for station pair (s_2, s_3) of p_1, marked in red, was already passed. However, the position of the start-point will now be moved due to the shortening, which is indicated by the striped area.

assigned to a bus ride that departs at a station before they arrive there. This is remedied by shortening the concerned intervals accordingly, which is indicated by the striped area in Figure 3.10. As a consequence, the interval that caused the finishing of the bus ride will be shortened to a single point in time, as seen in Figure 3.10 on the interval of p_3. We can improve this without loosing any passengers. To that end, we also keep track of the latest interval start-point concerning a bus ride. Then, like shown in Figure 3.11, when the bus ride is finished by an interval end-point, the latest start-point becomes the bus ride arrival time.

The clipping introduces a new problem, that needs to be handled. Consider the case shown in Figure 3.12, where the sweep line already passed the start-point of the interval for (s_2, s_3) of p_1, marked in red, before the arrival time of b_1 is determined. Thus, p_1 is also assigned to the active bus ride of (s_2, s_3), even tough the interval will be shortened, which could move the start-point beyond the actual arrival time of the active bus ride of (s_2, s_3). This problem is mitigated by restricting the number of unfinished bus rides, that a passenger can be assigned to, to one. So when the sweep-line encounters an interval start-point associated with a passenger that is already in a bus ride, which has its final arrival time not set yet, it is ignored for the moment. The finishing of the active bus ride will eventually trigger the shortening of this interval, which is when the start-point will be considered again by the sweep-line.

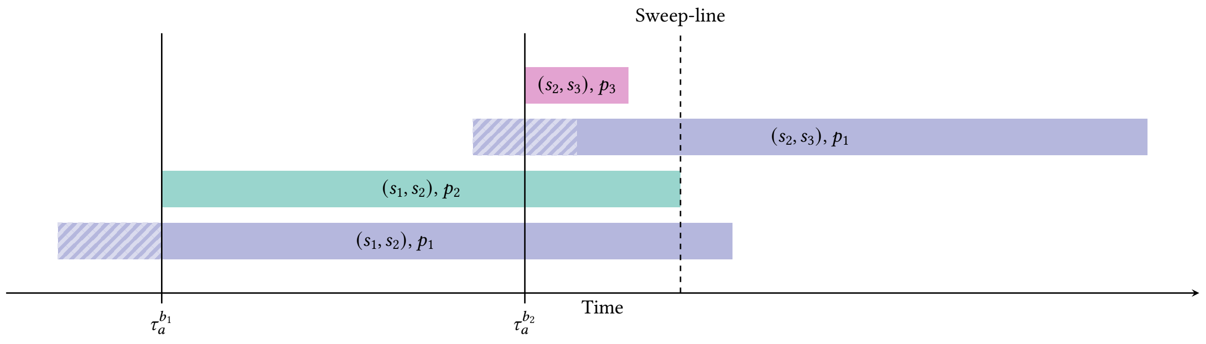


Figure 3.13.: Example for the assignment of passengers to bus rides using their bus ride demand intervals. The start-point of the interval of p_1 for (s_2, s_3) is ignored by the sweep-line, since p_1 is already assigned to b_1 at that time, which is not finished then. Its finishing leads to the mentioned start-point being processed retrospectively. Although p_1 could share the bus ride b_2 with p_3, it cannot be assigned to it in retrospect, since b_2 is already finished when the retrospective processing happens.

If this shortening is not enough to move the before ignored start-point beyond the sweep-line, like with the case shown in Figure 3.12, the sweep-line has to process them retrospectively. Note that by this time, the corresponding end-point has not been processed. Therefore, the associated passenger of every interval that is processed retrospectively like this, can simply be assigned to the active bus ride of the respective station-pair. The one case, where a sharing opportunity can actually be missed by the retrospective processing, can be seen in Figure 3.13. It might be possible to remedy this. However, it would probably involve changing an already finished bus ride, which in turn causes new special cases to consider. For now, dealing with this does not seem worth the computational effort to us.

Algorithm 3.3 shows how the whole procedure of computing bus rides from bus ride demands can be realized. It uses two priority queues. One for the start-points and one for the end-points of demand intervals. These will enable us to efficiently retrieve the next intervals to process. Moreover, there are data structures to hold the active bus ride for each station pair and the time of the currently relevant interval start-point for each passenger. The latter is used to determine how much the remaining intervals need to be shortened upon an bus ride finishing. The queues use the respective time of an interval's start- and end-points as the key and the station pair along with the passenger as the value. The element with the minimal key denotes the next start- and end-event respectively. The minimal keys of the two queues are compared in each iteration, to determine which event will be processed next. On a start-event, we assign the passenger to the currently active bus ride of the associated station pair. We also update the arrival time of the bus ride, so that it is always set to the time of the latest start-event. The handling of an end-event indicates that the active bus ride of the associated station pair needs to arrive at that time. For each passenger contained in that ride, the first of its demands is removed, as that is now served. We also delete the corresponding end-events that are still in the queue. Additionally, we update the passengers' total clipping time and insert the events for their next demand, with the interval clipped by increasing the key of the start-point accordingly. This ensures that all passenger demands are served, which we show in Lemma 3.5. Finally, a new, empty, bus ride is initialized for the concerned station-pair.

That a passenger is not assigned to multiple bus rides simultaneously, is guaranteed by the process of inserting the events of passenger demands only when all previous demands of that passenger are processed. Together with the clipping we use this in Lemma 3.6 to show that every transfer implied by the bus rides is doable in the sense, that the first bus arrives before the connecting bus leaves. We call two bus rides with this property *time-consistent*. Formally, bus rides $b_1 = (s_1, s_2, \tau_d^{b_1}, \tau_a^{b_1}, P^{b_1})$, $b_2 = (s_2, s_3, \tau_d^{b_2}, \tau_a^{b_2}, P^{b_2})$ are time-consistent, if the inequality $\tau_a^{b_1} \le \tau_d^{b_2}$ holds.

Algorithm 3.3: ComputeBusRides

```
Input: Bus ride demand $\Delta_B(p)$ sorted by $\tau_{a_{\min}}$ for passengers $p \in \mathcal{P}$, travel time function $t_G$
Output: Set $B$ of bus rides

Initialize priority queues $Q_{a_{\min}}$, $Q_{a_{\max}}$
activeBusRide($(s_1, s_2)$) $\leftarrow (s_1, s_2, \perp, \perp, \emptyset)$ for all $(s_1, s_2) \in S \times S$
forall $p$ in $\mathcal{P}$ do
    currentStartPoint($p$) $\leftarrow \perp$
    $t_{\mathtt{clip}}(p) \leftarrow 0$
    $(s_d, s_a, \tau_{a_{\min}}, \tau_{a_{\max}}) \leftarrow \Delta_B(p)$.front()
    $Q_{a_{\min}}$.insert($((s_d, s_a), p)$, $\tau_{a_{\min}}$)
    $Q_{a_{\max}}$.insert($((s_d, s_a), p)$, $\tau_{a_{\max}}$)
while $Q_{a_{\max}}$ not empty do
    $\tau_{\text{start}} \leftarrow Q_{a_{\min}}$.peekMinKey()
    $\tau_{\text{end}} \leftarrow Q_{a_{\max}}$.peekMinKey()
    if $\tau_{\text{start}} \leq \tau_{\text{end}}$ then
        handleStartEvent()
    else
        handleEndEvent()

Procedure handleStartEvent()
    $((s_1, s_2), p), \tau \leftarrow Q_{a_{\min}}$.deleteMin()
    $b \leftarrow$ activeBusRide($(s_1, s_2)$)
    $\tau_d^b \leftarrow \tau - t_G(s_1, s_2)$
    $\tau_a^b \leftarrow \tau$
    add $p$ to $P^b$
    currentStartPoint($p$) $\leftarrow \tau$

Procedure handleEndEvent()
    $((s_1, s_2), p), \tau \leftarrow Q_{a_{\max}}$.deleteMin()
    $b \leftarrow$ activeBusRide($(s_1, s_2)$)
    forall $p'$ in $P^b$ do
        $\Delta_B(p')$.popFront()
        $Q_{a_{\max}}$.delete($((s_1, s_2), p')$)
        $t_{\mathtt{clip}}(p') \leftarrow t_{\mathtt{clip}}(p') + \tau_a^b -$ currentStartPoint($p'$)
        $(s_d, s_a, \tau_{a_{\min}}, \tau_{a_{\max}}) \leftarrow \Delta_B(p')$.front()
        $Q_{a_{\min}}$.insert($((s_d, s_a), p')$, $\tau_{a_{\min}} + t_{\mathtt{clip}}(p')$)
        $Q_{a_{\max}}$.insert($((s_d, s_a), p')$, $\tau_{a_{\max}}$)
    add $b$ to $B$
    activeBusRide($(s_1, s_2)$) $\leftarrow (s_1, s_2, \perp, \perp, \emptyset)$
```

In summary, assuming all bus rides are served, it follows from Lemmas 3.5 and 3.6 that every passenger has an itinerary to fulfil their demand, which we formalise in Theorem 3.7. Additionally, the way we insert the events results in the queues' never having more than $|\mathcal{P}|$ elements. Thus, `deleteMin` can be done in $O(\log(|\mathcal{P}|)$ for these queues. The loop in line 26 is executed for each passenger in the concerned bus ride. Since a start-event must have been processed for each of them, the number of times the loop is executed equals the number of start-events, which is equal to the total number of bus ride demands. Therefore, Algorithm 3.3 takes $O(|\Delta_B(\mathcal{P})| \cdot \log |\mathcal{P}|)$ time.

Lemma 3.5: *Let B be set of bus rides computed by Algorithm 3.3 from bus ride demands $\Delta_B(p)$ of passengers $p \in \mathcal{P}$. For every bus ride demand $(s_d, s_a, \tau_{a_{min}}, \tau_{a_{max}}) \in \Delta_B(p)$ there is a bus ride $(s_d^b, s_a^b, \tau_d^b, \tau_a^b, P^b) \in B$ with $s_d^b = s_d$, $s_a^b = s_a$, $\tau_a^b \in [\tau_{a_{min}}, \tau_{a_{max}}]$ and $p \in P^b$.*

Proof. When the start-point of a demand $(s_d, s_a, \tau_{d_{\min}}, \tau_{a_{\max}})$ is processed, the corresponding passenger is assigned to the active bus ride b of the station pair associated with the demand, so $s_d^b = s_d$ and $s_a^b = s_a$. Since the arrival time of the bus is set to the start-point being processed in line 20, $\tau_a^b \geq \tau_{a_{\min}}$ holds. At the latest when the corresponding end-point is processed, bus ride b will be finished by the assignment of a new bus ride as the active bus ride of the respective station pair in line 34. Hence, $\tau_a^b \leq \tau_{a_{\max}}$ holds as well.

It remains to show that for every demand a start- and end-event will be processed eventually. The start- and end-point of the first bus ride demand of each passenger is inserted in the respective queue in lines 7 and 8 of Algorithm 3.3. Note, that start-events are handled before end-events. For each inserted start-point, the corresponding passenger is eventually assigned to a bus ride. Additionally, every bus ride containing a passenger is eventually finished by an end-event of one of the contained passengers. This leads to the events of the next demand of all contained passengers being inserted in the respective queues. ∎

Lemma 3.6: *Every two bus rides $b_1 = (s_1, s_2, \tau_d^{b_1}, \tau_a^{b_1}, P^{b_1})$, $b_2 = (s_2, s_3, \tau_d^{b_2}, \tau_a^{b_2}, P^{b_2})$ with $|P^{b_1} \cap P^{b_2}| > 0$ computed by Algorithm 3.3, are time-consistent.*

Proof. Let $b_1 = (s_1, s_2, \tau_d^{b_1}, \tau_a^{b_1}, P^{b_1})$, $b_2 = (s_2, s_3, \tau_d^{b_2}, \tau_a^{b_2}, P^{b_2})$ with $|P^{b_1} \cap P^{b_2}| > 0$. Since there is a passenger $p \in P^{b_1} \cap P^{b_2}$, there were bus ride demands $\delta_B = (s_1, s_2, \tau_{a_{\min}}, \tau_{a_{\max}})$, $\delta'_B = (s_2, s_3, \tau'_{a_{\min}}, \tau'_{a_{\max}}) \in \Delta_B(p)$. Let $t_{\text{clip}}(p)$ denote the clipping time of p. From Observation 3.4 follows that $\tau_{a_{\min}} < \tau'_{a_{\min}}$. Thus, the start-event of δ_B will be processed before that of δ'_B and $t_{\text{clip}}(p) \geq \tau_a^{b_1} - \tau_{a_{\min}}$. Since the start-point of δ'_B will be moved by $t_{\text{clip}}(p)$, this implies for the arrival time of b_2, that $\tau_a^{b_2} \geq \tau'_{a_{\min}} + \tau_a^{b_1} - \tau_{a_{\min}}$ holds and hence, also $\tau_d^{b_2} \geq \tau'_{a_{\min}} + \tau_a^{b_1} - \tau_{a_{\min}} - t_G(s_2, s_3)$. By Observation 3.4, the statement holds. ∎

Theorem 3.7: *The bus rides computed by Algorithm 3.3 from bus demands $\Delta_B(\mathcal{P})$ offer each passenger $p \in \mathcal{P}$ time-consistent bus rides, that settle their bus ride demand $\Delta_B(p)$.*

Proof. Follows from Lemmas 3.5 and 3.6. ∎

3.4. Bus Ride Flows

We now have the set B of bus rides that need to be served in order to get all passengers to their destination in time. The next step is to determine how many buses are needed and which bus rides are performed by each of them. To that end we construct a weighted flow network on which we then solve the Minimum Cost Flow problem in order to obtain a set of bus itineraries for a given maximum number β of buses. By repeatedly solving the problem for increasing values for β, we also get the minimal number of required buses to serve all bus rides, while minimizing the number of passenger transfers. Section 3.4.1 covers the details of this flow network construction and in Section 3.4.2 we prove its correctness.

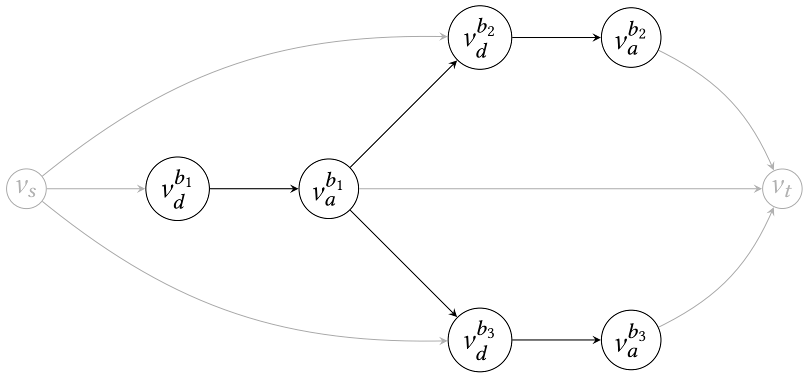


Figure 3.14.: Example for the graph of a bus ride flow network on bus rides $\{b_1, b_2, b_3\}$, where b_2 and b_3 are reachable from b_1.

3.4.1. Construction

A simple example is shown in Figure 3.14. The idea of this construction is to have bus rides modelled by edges and buses represented by flow units, flowing through the bus ride edges they serve. So for each bus ride $b \in B$ we create the *departure vertex* $v_d^b = (s_d^b, \tau_d^b)$ and the *arrival vertex* $v_a^b = (s_a^b, \tau_a^b)$, associated with the time and station of the departure and arrival of the bus ride respectively. In between, we insert a directed edge from departure vertex to arrival vertex, representing the bus ride itself. Now, we want to model the choices a bus has after serving a bus ride, of which bus ride to serve next. To that end, we introduce the concept of reachability of bus rides. Consider the occurring departure times $T_d(B) = \{\tau_d^b \mid b \in B\}$ of B. For a given maximum waiting time w and point in time τ we define the *waiting-time neighbourhood*

$$U_w(\tau) = \{\tau' \in T_d(B) \mid \tau \leq \tau' \leq \tau + w\}.$$

With this, we call bus ride b_2 *reachable* from a bus ride b_1, if and only if $o^{b_2} = d^{b_1}$ and $\tau_d^{b_2} \in U_w(\tau_a^{b_1})$.

In order to connect the bus rides in our flow network, we insert an edge from the arrival vertex of a bus ride to the departure vertices of all reachable bus rides. The *source* v_s and *sink* v_t of the flow network are additional vertices which could be interpreted as a start- and end point for buses respectively. The source is therefore connected with the departure vertices of all bus rides and the arrival vertices of all bus rides are in turn connected with the sink. With this the graph $G_F(B) = (V, E)$ of the bus ride flow network is complete and consists of vertex set $V = \{v_d^b, v_a^b \mid b \in B\} \cup \{v_s, v_t\}$ and edge set $E = E_s \cup E_t \cup E_R \cup E_M$ where

$$\begin{aligned}
\text{the } \textit{source edges } E_s &= \{(v_s, v_d^b) \mid b \in B\},\\
\text{the } \textit{sink edges } E_t &= \{(v_a^b, v_t) \mid b \in B\},\\
\text{the } \textit{ride edges } E_R &= \{(v_d^b, v_a^b) \mid b \in B\},\\
\text{the } \textit{reachability edges } E_M &= \{(v_a^{b_1}, v_d^{b_2}) \mid b_1, b_2 \in B,\ b_2 \text{ reachable from } b_1\}.
\end{aligned}$$

To complete the flow network construction, we now define the capacity function and the cost function. As mentioned before, a flow unit in the flow network should represent a bus. Since a single bus ride should only be served by one bus, we set the capacity of the ride edges to one. The source edges connect to the departure vertices of bus rides. These always have exactly one outgoing ride edge and thus,

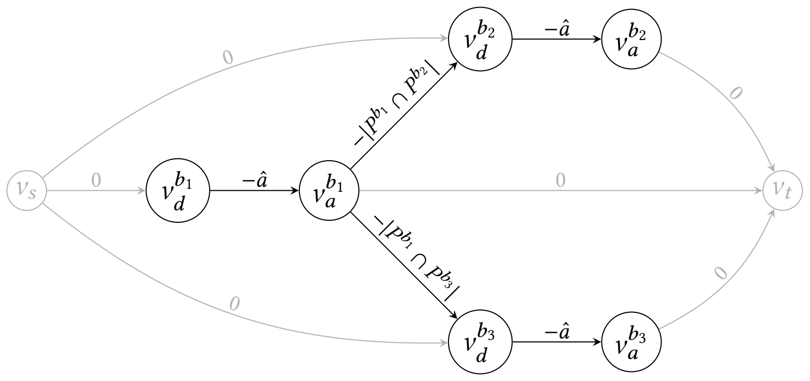


Figure 3.15.: Example for a bus ride flow network $F(B)$ with $B = \{b_1, b_2, b_3\}$ where b_2 and b_3 are reachable from b_1. The labels on the edges represent their cost.

there is a maximal out-flow of one on the departure vertices. Hence, a capacity of one does not restrict the possible flows. A symmetrical argument can be made for the sink edges and the in-flow of arrival vertices. Since the reachability edges connect arrival vertices with departure vertices, there will also be a maximal flow of one on these edges. So in summary, we obtain $c : E \to \mathbb{N}$ with $c(e) = 1$ for all $e \in E$ as the capacity function.

For the cost function, we want it to ensure served rides are maximized first and the number of transfers are minimized as a secondary criterion. To that end, we assign a negative cost to some of the edges, which will act as a reward for sending flow units through them. To address the secondary criterion, we reward reachability edges that connect bus rides with a lot of overlap in their containing passengers. All passengers contained in both bus rides can consequentially stay in the bus and do not have to transfer to another bus at the end of the first ride, which results in more reward for fewer overall transfers.

However, the main criterion is still that as many bus rides are served as possible. Therefore, we need the reward of serving a bus ride surpass any accumulation of reward from minimizing transfers. This is done by setting the reward of bus ride edges higher than an overestimate $\hat{a} \in \mathbb{N}$ of the possible reward from transfer minimization accumulated on any path from source to sink. So $\hat{a} > \sum_{b \in B} |P^b|$. All of the other edges, which are the ones connected to source or sink, are assigned a reward of zero. This gives us $a : E \to \mathbb{Z}$ with

$$a(e) = \begin{cases} -\hat{a}, & e \in E_R \\ -|P^{b_1} \cap P^{b_2}|, & e = (v_d^{b_1}, v_a^{b_2}) \in E_M \\ 0 & otherwise \end{cases}$$

as the *cost function.* Now, the definition of the *bus ride flow network* of B is complete and states as $F(B) = (G_F(B), v_s, v_t, c, a)$. Figure 3.15 shows the example from before with the associated edge costs.

A bus ride flow network for bus rides B has $2|B| + 2$ vertices. Additionally, there are $|B|$ source edges, $|B|$ sink edges and $|B|$ ride edges. The number of reachability edges depends on the reachability of rides and thus also on the maximum waiting time. In the worst case, there are $O(|B|^2)$ reachability edges. This gives a worst-case time-complexity of $O(|B^2|)$ for the construction of $F(B)$. However, in realistic instances we expect the bus rides to be spread out in space as well as time, along with a small maximum waiting time in relation to the overall time horizon. With the way the bus rides are computed, they can be stored by their departure station and sorted by departure time in order to efficiently determine the reachable rides for every bus ride. Thus, we expect the construction in realistic scenarios to perform considerably better than the worst case.

3.4.2. Correctness

In the following, we assume a flow network $F(B)$ constructed as specified in Section 3.4.1 from a set B of bus rides. In the course of this section we want to show that solving the Minimum Cost Flow problem on $F(B)$ leaves us with the intended bus itineraries, maximizing served rides and minimizing transfers. We start by transferring the concept of itineraries to our flow network. Since a flow unit is meant to represent a bus, the path taken by a flow unit will represent the itinerary of a single bus. Note, that the flows considered in this work are always integer-valued. A *bus ride itinerary* will denote the assignment of all of the bus rides to buses, meaning it is equivalent to a set of paths I from the source to the sink of $F(B)$, that share no vertices except the source and the sink itself. We call a bus ride itinerary *satisfying* if and only if it contains all bus ride edges of B. Further, we establish the set of bus ride pairs between which a transfer happens in I as

$$\mathcal{T}(I) = \{(b_1, b_2) \mid (v_a^{b_1}, v_d^{b_2}) \in \{e \in E(R) \mid R \in I\}\},$$

to define the *number of transfers*

$$t(I) = \left(\sum_{(b_1,b_2)\in\mathcal{T}(I)} (|P^{b_1} \setminus P^{b_2}| \right) - |\mathcal{P}|.$$

We then show in Lemma 3.8 that a feasible flow on our flow network yields us such a satisfying bus ride itinerary I and additionally, the value of the flow equals the size $|I|$ of the itinerary.

Lemma 3.8: *A feasible flow f is equivalent to a bus ride itinerary of size $|f|$.*

Proof. We first show that we can construct a bus ride itinerary from a feasible flow. Let f be a feasible flow on $F(B)$. Since f sends $|f|$ flow units from source v_s to sink v_t and all edges in $F(B)$ have unit capacity, f is equivalent to $|f|$ $v_s v_t$-paths. Suppose there are saturated $v_s v_t$-paths P_1, P_2, that share a vertex $v \in V(F(B)) \setminus \{v_s, v_t\}$.

Case 1: v is a departure vertex of a bus ride $b \in B$: Then v has only one outgoing edge, the one to v_a^b. Because of flow conservation, v can thus also only have one incoming edge with flow.

Case 2: v is an arrival vertex of a bus ride b: Then v has only one incoming edge, the one to v_d^b. Because of flow conservation, v can thus also only have one outgoing edge with flow.

Therefore, in both cases, such two paths P_1, P_2 cannot exist.

Now we show that we can construct a feasible flow from a bus ride itinerary. Let I be a bus ride itinerary of size k. We construct a flow f by assigning every edge of every path in I a flow value of 1. Since there are k paths, the value of f is also k. Furthermore, the paths in I are vertex disjoint, hence flow conservation holds for f. ∎

With Lemma 3.9, we show that by the construction of our cost function, a min-cost-flow on the flow network maximizes the number of covered bus rides, as intended. Further, we show in Lemma 3.10, that the cost function ensures the second criterion of minimizing the transfers for a fixed number of served rides as well. Using these two lemmata, we conclude the section with Theorem 3.11, showing that solving Minimum Cost Flow on our flow network construction yields an itinerary, which maximizes served bus rides and minimizes the number of transfers for a fixed number of buses. Successively solving Minimum Cost Flow with an increasing number of buses thus also finds the minimum number of buses needed to serve all bus rides. Note, that there are implementations to solve Minimum Cost Flow by repeatedly finding augmenting paths [Ber+92]. Since all edges have a capacity of one, each augmenting path corresponds to another bus itinerary. This method can be used to enumerate the solutions for increasing number of buses. We use a custom implementation to model the flow network and find the augmenting paths instead of a black-box solver, since in this way, we can leverage the unit capacities and retrieve the solutions for increasing number of buses like mentioned before. There certainly is some improvement potential for our Min Cost Flow implementation and there may even be algorithms that make more use of the specific structure of our flow network construction, yet we did not find any in our research.

Lemma 3.9: *A min-cost-flow maximizes the number of bus ride edges contained in the equivalent bus ride itinerary.*

Proof. Let f be a min-cost flow on a bus ride flow network F and I_f the equivalent bus ride itinerary, containing k bus ride edges.
Suppose there is a flow $\hat{f}$ with equivalent bus ride itinerary $I_{\hat{f}}$ containing $\geq k+1$ bus ride edges. Then, f is missing the reward $\hat{a}$ of a bus ride edge which, by the construction of the cost function, cannot be mitigated by the reward of reachability edges. Therefore, $a(\hat{f}) \leq (k+1) \cdot \hat{a} \leq a(f) \leq k \cdot \hat{a}$, which contradicts that f is a min-cost flow. ■

Lemma 3.10: *Let f, f' be flows with their respective equivalent bus ride itineraries $I_f, I_{f'}$ containing k bus ride edges. For their number of transfers, it holds that $t(I_f) < t(I_{f'}) \iff a(f) < a(f')$.*

Proof. Let $E_M(f) = \{e \in E_M \mid f(e) > 0\}$ be the set of reachability edges of a flow f. Further, let f, f' be two flows with their respective equivalent bus ride itineraries $I_f, I_{f'}$ containing k bus ride edges. Then,

$$\begin{array}{rccc}
 & t(I_f) & < & t(I_{f'}) \\
\iff & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_f)}(|P^{b_1} \setminus P^{b_2}|) - |\mathcal{P}| & < & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_{f'})}(|P^{b_1} \setminus P^{b_2}|) - |\mathcal{P}| \\
\iff & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_f)}(|P^{b_1} \setminus P^{b_2}|) & < & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_{f'})}(|P^{b_1} \setminus P^{b_2}|) \\
\iff & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_f)}(|P^{b_1} \cap \overline{P^{b_2}}|) & < & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_{f'})}(|P^{b_1} \cap \overline{P^{b_2}}|) \\
\iff & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_f)}(-|P^{b_1} \cap P^{b_2}|) & < & \sum\limits_{(b_1,b_2)\in\mathcal{T}(I_{f'})}(-|P^{b_1} \cap P^{b_2}|) \\
\iff & \sum\limits_{(v_d^{b_1},v_a^{b_2})\in E_M(f)}(-|P^{b_1} \cap P^{b_2}|) & < & \sum\limits_{(v_d^{b_1},v_a^{b_2})\in E_M(f')}(-|P^{b_1} \cap P^{b_2}|) \\
\iff & \sum\limits_{e\in E_M(f)}(a(e)) & < & \sum\limits_{e\in E_M(f')}(a(e)) \\
\iff & \sum\limits_{e\in E_M(f)}(a(e)) - k\hat{a} & < & \sum\limits_{e\in E_M(f')}(a(e)) - k\hat{a} \\
\iff & a(f) & < & a(f').
\end{array}$$

■

Theorem 3.11: *For a given set B of bus rides and a number β of buses, solving* Minimum Cost Flow *on $F(B)$ with a flow demand of β yields the itinerary with the maximal number of served bus rides and the minimal number of transfers for this number of served bus rides.*

Proof. Let f be a min cost flow on $F(B)$ with a flow demand of β and I_f the equivalent bus ride itinerary. By Lemma 3.9, f maximizes the number of bus rides contained in I_f, so the number of served bus rides. Let k be the number of served bus rides. Then, Lemma 3.10 implies, that I_f has the minimal amount of transfers for k served bus rides. ∎

4. The Baseline Approach

In order to have a reference point, to assess our solver, we implemented a naïve baseline solver described in Algorithm 4.1. For a passenger demand, we first compute the shortest path from the passenger's departure station to their arrival station in the station graph. From the length of the shortest path, we can determine the time by which they arrive earlier at their arrival station, than their latest arrival time. This will be their waiting time contingent. Additionally, we keep track of how much waiting time they already used, which we call their delay. The shortest path is then processed edge by edge starting from the departure station. For each edge, we check whether waiting for the next bus ride happening between the respective stations is inside the waiting time contingent of the passenger. If it is, they can take this ride and the waiting time contingent and delay of the passenger are updated accordingly. Else, a new bus ride is created for the concerned station pair, departing as soon as the passenger arrives at the departure station of the ride. We process the passenger demands in descending order of their departure times in order to increase the potential to use already existing bus rides. The resulting set of bus rides can then be compared with the set of bus rides obtained from the detour approach by constructing a bus ride flow from each of the sets and solving Minimum Cost Flow with successively increasing numbers of buses.

Algorithm 4.1: BaselineSolver

```
Input: Station graph $G$, set $\Delta$ of passenger demands sorted by descending departure time
Output: Set $B$ of bus rides

forall demand $(s_d, s_a, \tau_d, \tau_a)$ of $p$ in $\Delta$ do
    dist, succ $\leftarrow$ shortestPath($G, s_d, s_a$)
    $t_{\text{wait}} \leftarrow \tau_a - (\tau_d + \text{dist}[s_a])$
    $t_{\text{delay}} \leftarrow 0$
    $v \leftarrow s_d$
    while $v$ is not $s_a$ do
        $\tau_{\text{now}} \leftarrow \tau_d + \text{dist}[v] + t_{\text{delay}}$
        $u \leftarrow \text{succ}[v]$
        $b \leftarrow \text{getUpcomingRide}(v, u)$
        if $\tau_d^b < \tau_{\text{now}} + t_{\text{wait}}$ then
            // Bus ride $b$ can be taken
            add $p$ to $P^b$
            $t_{\text{delay}} \leftarrow t_{\text{delay}} + (\tau_d^b - \tau_{\text{now}})$
            $t_{\text{wait}} \leftarrow t_{\text{wait}} - (\tau_d^b - \tau_{\text{now}})$
        else
            // Spawn new bus ride
            $\tau_a^{b'} \leftarrow \tau_d + \text{dist}[u] + t_{\text{delay}}$
            add $(v, u, \tau_{\text{now}}, \tau_a^{b'}, \{p\})$ to $B$
        $v \leftarrow u$
```

5. Evaluation

This chapter presents a comparative evaluation of our proposed solver, which is based on the detour-approach, against a naïve baseline. The evaluation focuses on both computational performance and solution quality. We analyse how different parameters such as the maximal allowed travel time and bus waiting time affect the computation time and solution quality. To this end, we run both solvers on a station graph based on a real-world road network with realistic passenger demands.

5.1. Experimental Setup

All algorithms were implemented in C++23, compiled with gcc version 11.4.0 using cmake version 3.22.1 in its release configuration and executed on a 2023US-TR4 Supermicro A+ Server with 256GB RAM and two 16-Core AMD EPYC Zen1 7281 CPUs with 2666MHz.

We created the instances from a dataset consisting of a street graph and a set of passenger demands. The street graph represents roughly the Stuttgart Metropolitan Region in Germany [SHP11]. The street graph consists of 134 663 vertices and 307 759 edges. From this, we generate station graphs of varied sizes by overlaying a $n \times n$ uniform grid for some integer n and choosing the geographically closest vertex as the station for this grid cell. If there happen to be no vertices in a grid cell, the cell will not have a station. Two stations are adjacent in the station graph, if there is no other station on the shortest path between them in the street graph.

The passenger demands are given as a departure vertex and an arrival vertex in the street graph as well as an earliest departure time. The set of demands used in this chapter has 280 364 entries and represents an hour of rush-hour car traffic on a Tuesday evening. The data was created with mobiTopp using calibration data of a household travel survey conducted in 2009/2010 [MKV13 | MV15 | Stu11]. As the departure and arrival station, we assign the closest station, in terms of travel time, to the departure and arrival vertex respectively. For some of these demands, their departure and arrival vertices could be assigned to the same station in the station graph. Such demands will be discarded and not considered by the solvers. For each demand, we also need a latest arrival time which is not given by this dataset. Hence, we determine the minimal travel time from departure station to arrival station in the station graph and multiply it by a factor α to obtain a maximal travel time. So the detours in the station graph can take a factor of α longer than the shortest path. Table 5.1 shows the grid sizes we use along with the size and density of the resulting station graphs as well as the number of considered passenger demands.

Table 5.1.: Size and density of the station graph created with different grid sizes n along with the size of the considered passenger demand set $\delta(\mathcal{P})$ after discarding the ones where the assigned departure and arrival station are identical.

Name	n	#Vertices	#Edges	Density	$\lvert\delta(\mathcal{P})\rvert$
grid40	40	374	42 026	0.301	188 263
grid50	50	570	80 602	0.249	200 397

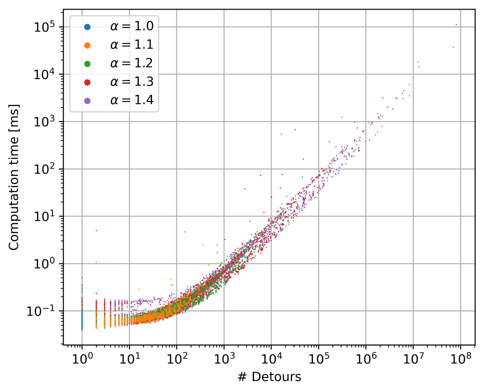


Figure 5.1.: Computation time of the individual detour-DAGs by the number of their containing detours for station graph *grid40* and differing travel time multipliers α.

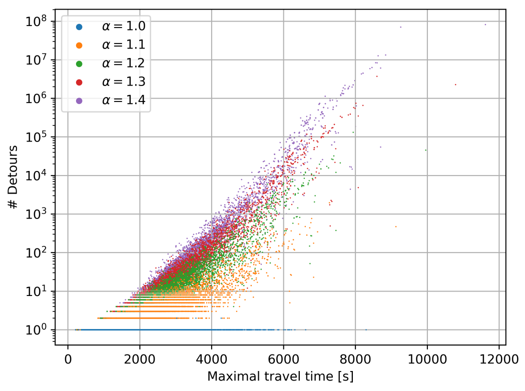


Figure 5.2.: Number of containing detours of individual detour-DAGs by the maximal allowed travel time for station graph *grid40* and differing travel time multipliers α.

5.2. Detour-DAG Computation

In this section, we evaluate the detour-DAG computation. The full results with varying travel time factors and on station graphs of different grid sizes can be seen in Table A.1. Figure 5.1 shows the relation between the computation time of the individual detour-DAGs and the number of proper detour paths, confirming that, for large number of detours, it is linear. We see in Figure 5.2, that the number of detours increases exponentially with the maximal travel time.

Looking at the distribution of maximal travel times shown in Figure 5.3 in relation to the contribution to the total computation time of detour-DAGs by their maximal travel time seen in Figure 5.4, suggests that for $\alpha = 1.4$ the computation time is dominated by few demands with large travel times. For $\alpha = 1.3$ this effect is still noticeable, while it is not apparent with lower values for α. Refer to Figure A.2 for a version of Figure 5.4, where the contributions with lower values for α can be observed more clearly. This is in line with the total number of explored edges increasing exponentially with increasing values for α, as shown in Figure 5.5, where we can also observe that the total number of edges that are part of the final detour-DAGs follow this trend as well. Depending on the application, it may thus be feasible to only consider demands below a certain distance threshold, confining for example to intra-city transport.

Furthermore, Figures 5.5 and 5.6 indicate that the grid size also has an influence on the computation time. However, we do not have enough data to reasonably draw any conclusions on that, which is partly due to the computation times reaching values beyond our time constraints. The computations for $\alpha = 1.4$ for station graph *grid50* were terminated after surpassing 5 hours.

Overall it becomes clear to us that this way of computing detour-DAGs does not scale well and is therefore only feasible for station graphs up to a certain size. In the case of the station graphs created with our grid-based approach, a grid size of 40 still allows for detour-DAG computations up to $\alpha = 1.4$ in a reasonable amount of time. We will therefore focus on the instance *grid40*.

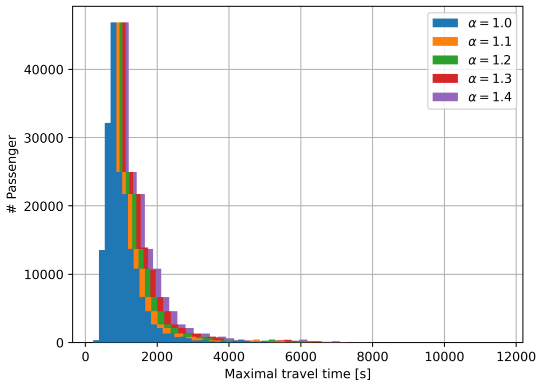


Figure 5.3.: Histogram of maximal travel times on station graph *grid40* for varying travel time multipliers α.

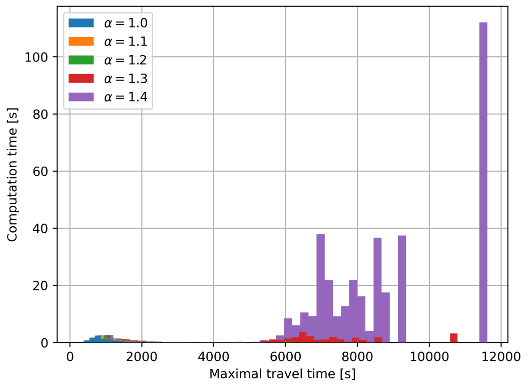


Figure 5.4.: Histogram showing the cumulative computation time associated with detour-DAGs falling into the corresponding travel time bin, computed on station graph *grid40* with varying travel time multipliers α.

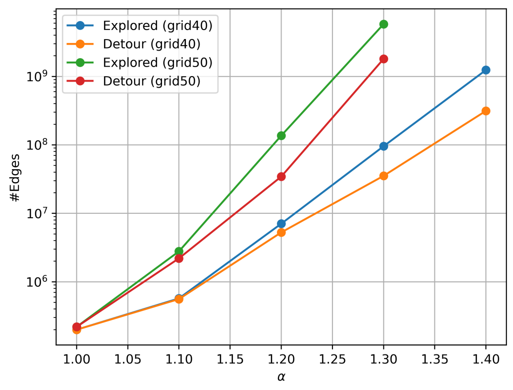


Figure 5.5.: Number of explored edges in comparison to the number of edges part of the final detour-DAG for varying travel time multipliers α and different station graphs.

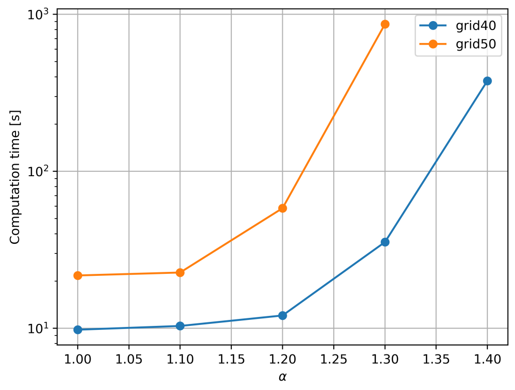


Figure 5.6.: Total detour-DAG computation time for varying maximal travel time multipliers α on different station graphs.

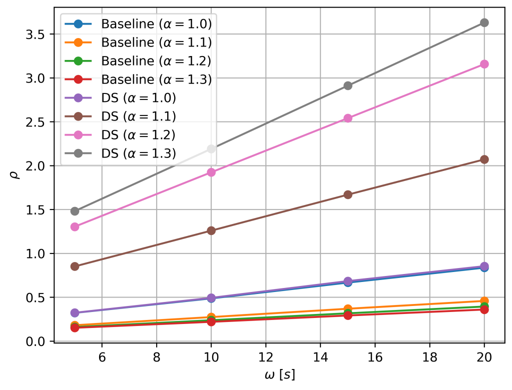


Figure 5.7.: Reachability score ρ of the solver runs on station graph *grid40* for different maximal bus waiting times ω. Note, that for $\alpha = 1.0$ the detour-solver data obscures the baseline data.

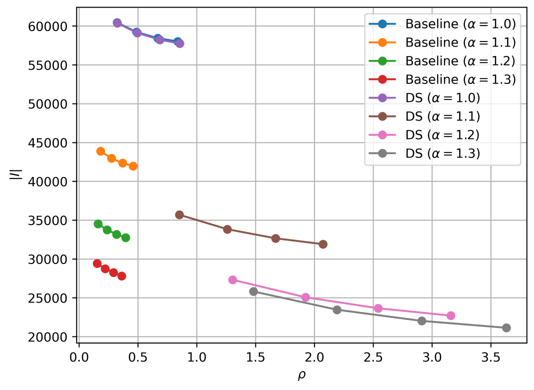


Figure 5.8.: Number of buses computed to settle the passenger demand of the solver runs on station graph *grid40* by their reachability score ρ. Note, that for $\alpha = 1.0$ the detour-solver data obscures the baseline data.

5.3. Detour-Solver

In this section we evaluate the detour-based solver (DS) by comparing it with the baseline solver. To compare the computed bus rides, we also calculate a reachability score ρ for each bus ride flow network, which indicates the average number of bus rides that are reachable from a bus ride. Refer to Table A.2 for a detailed insight of all the results.

As expected, ρ increases with the maximal bus waiting time ω, which can be seen in Figure 5.7. A high value for ρ indicates that, on average, there are more choices for a bus to continue its travel. Interestingly, with the detour-solver, ρ increases as α increases, whereas with the baseline, ρ decreases. There is however no direct correlation between ρ and the number of buses needed, which we see in Figure 5.8, where we also observe that our detour-solver surpasses the baseline for every parameter configuration.

Although 21 147 buses as the best result of the detour-solver still seem disproportionally many for a period of 60 minutes, even in rush hour, the number of overall passenger transfers stays consistently low over all experiments with both solvers, as shown in Figure 5.9, with 2 031 being the highest transfer count. Looking at the comparably small amount of allowed delay being exhausted, displayed in Figure 5.10, it appears to us that there, together with the low transfer count, lies the largest potential to further reduce the number of required buses. Regarding the behaviour of transfer count seen in Figure 5.9, we also observe that with the detour-solver, the number of transfers decreases significantly when the maximum waiting time of buses is increased. This effect probably occurs since the bus would need to end its journey, when not allowed to wait until the next bus ride, and a new one would need to continue in its place, causing all contained passengers to transfer. That this does not appear with the baseline solver is consistent with the fact of it not causing much delay, and thus waiting time at stations, observed in Figure 5.10.

We also observe that the more additional travel time is allowed, controlled by increasing α, the better the results. This behaves as expected, since when detours are allowed to be longer, it is more likely to have more overlap with other passengers' detours. Significantly larger values of α may still be

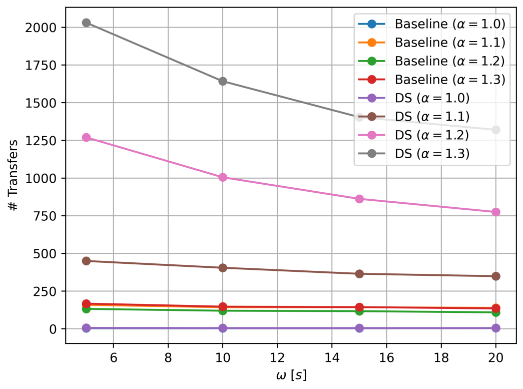


Figure 5.9.: Number of overall passenger transfers by the allowed bus waiting time ω for runs of different maximal travel time multipliers α on station graph *grid40*. Note, that for $\alpha = 1.0$ the detour-solver data obscures the baseline data.

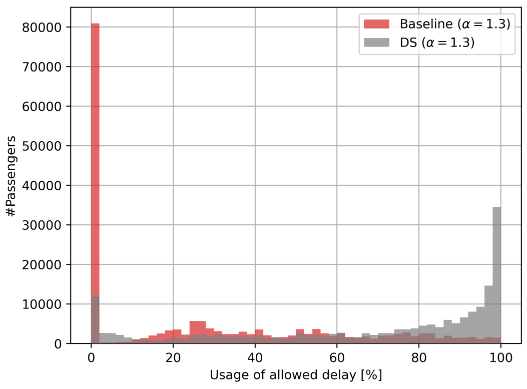


Figure 5.10.: Distribution of allowed arrival delay usage on station graph *grid40* with $\omega = 20$.

relevant for practical applications. Which range of values can be deemed realistic however, represents a fundamental design decision in real-world settings. Moreover, it could be given as part of the input data, by incorporating the delay acceptance in the demand simulation.

Through the number of detours, parameters like the grid size and the maximal travel time factor greatly influence the computation time, which can easily reach multiple hours. By controlling them, the computation time can be kept within reasonable times according to the application, which will of course influence solution quality.

6. Conclusion

In this work, we adapted the idea of using possible detour paths to solve a ride sharing problem to the domain of transit planning. After computing the DAG of possible detours for each passenger demand, we used it to heuristically compute the bus demand of each passenger by greedily choosing the route where the most other passengers may travel. As these depict time-intervals during which a passenger requests a bus ride between a pair of stations, we afterwards computed at what points in time a bus ride should happen between each pair of stations, to settle the demand. From the set of bus rides, we then constructed an instance of the Minimum Cost Flow problem and showed how solving it can be used to obtain a set of bus itineraries that serve all bus rides and thus settle all passenger demands, while minimizing the number of overall passenger transfers. We evaluated this approach by comparing it with a naïve baseline and running experiments for different input parameters on an authentic instance, finding that, while the baseline is outperformed in solution quality, there still is room for improvement in both the computation time as well as the computed number of bus itineraries. We identified the number of transfers along with the usage of allowed arrival delay of our solutions as the place where the potential for this is located.

We attribute the general tendency of the detour-solver to result in very low transfer counts but still large number of buses to the fact that the underlying approach was originally developed for ride-sharing contexts, where minimizing transfers, delays, and waiting times is probably far more critical due to user expectations. In contrast, public transit systems typically tolerate higher levels of waiting times and transfer counts. Additionally, ride-sharing tends to place greater emphasis on serving all passengers, whereas public transit operators often aim to maximize the total number of passengers served with a somewhat fixed capacity of buses, accepting that some passengers may not be accommodated, which our approach does not allow. Relaxing this requirement could also significantly reduce computation time, as the detour calculations for demands with very large maximal travel times could be skipped, with the possibility that the resulting bus system may still be able to serve those demands, but without the explicit constraint to do so.

A computational gain like this would open up opportunities that could also enhance the solution quality. It may be possible to integrate the station placement in the algorithm itself, allowing for solutions that are more adaptive to the demand. Even without integrating station placement, alternative station graph topologies beyond the simple uniform grid used in our evaluation, such as demand-aware or hierarchical layouts, could improve the solution quality of our solver and potentially its performance. Moreover, it may also be possible to further refine the solutions in a post processing stage. For example, small local optimizations like removing redundant stops in bus itineraries where no passengers board or exit could be applied. Additionally, the concept of bus ride reachability could be extended. Currently, we only consider temporal proximity, but incorporating spatial proximity could allow buses to continue with geographically nearby rides and capture real-world conditions more accurately.

Lastly, an improved heuristic for computing bus demands seems promising. The current greedy approach is prone to converge to local maxima and it appears that the information available by the computed detour-DAGs could be leveraged more effectively. Exploring this further could therefore be a valuable direction for future work.

Bibliography

[ANJ19] Sunghi An, Daisik Nam, and R. Jayakrishnan. “Impacts of integrating shared autonomous vehicles into a Peer-to-Peer ridesharing system”. In: *Procedia Computer Science* Volume 151 (Jan. 2019), pp. 511–518. ISSN: 1877-0509. DOI: *10.1016/j.procs.2019.04.069*.

[Ber+92] Dimitri P Bertsekas et al. “An auction/sequential shortest path algorithm for the minimum cost network flow problem”. In: (1992).

[Cho+20] Joseph YJ Chow, Srushti Rath, Gyugeun Yoon, Patrick Scalise, and Sara Alanis Saenz. “Spectrum of public transit operations: from fixed route to microtransit”. In: (2020).

[CMSQ19] Wenyi Chen, Martijn Mes, Marco Schutten, and Job Quint. “A Ride-Sharing Problem with Meeting Points and Return Restrictions”. en. In: *Transportation Science* Volume 53 (Mar. 2019), pp. 401–426. ISSN: 0041-1655, 1526-5447. DOI: *10.1287/trsc.2018.0832*.

[CW86] Avishai Ceder and Nigel H.M. Wilson. “Bus network design”. In: *Transportation Research Part B: Methodological* Volume 20 (1986), pp. 331–344. ISSN: 0191-2615. DOI: *https://doi.org/10.1016/0191-2615(86)90047-0*.

[GH08] Valérie Guihaire and Jin-Kao Hao. “Transit network design and scheduling: A global review”. In: *Transportation Research Part A: Policy and Practice* Volume 42 (Dec. 2008), pp. 1251–1273. ISSN: 0965-8564. DOI: *10.1016/j.tra.2008.03.011*.

[Man80] Christoph E. Mandl. “Evaluation and optimization of urban public transportation networks”. In: *European Journal of Operational Research* Volume 5 (1980), pp. 396–404. ISSN: 0377-2217. DOI: *https://doi.org/10.1016/0377-2217(80)90126-5*.

[MJ17] Neda Masoud and R. Jayakrishnan. “A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system”. en. In: *Transportation Research Part B: Methodological* Volume 106 (Dec. 2017), pp. 218–236. ISSN: 01912615. DOI: *10.1016/j.trb.2017.10.006*.

[MKV13] Nicolai Mallig, Martin Kagerbauer, and Peter Vortisch. “mobiTopp – A Modular Agent-based Travel Demand Modelling Framework”. In: *Procedia Computer Science* Volume 19 (2013), pp. 854–859. ISSN: 1877-0509. DOI: *https://doi.org/10.1016/j.procs.2013.06.114*.

[MNYJ17] Neda Masoud, Daisik Nam, Jiangbo Yu, and R. Jayakrishnan. “Promoting Peer-to-Peer Ridesharing Services as Transit System Feeders”. en. In: *Transportation Research Record: Journal of the Transportation Research Board* Volume 2650 (Jan. 2017), pp. 74–83. ISSN: 0361-1981, 2169-4052. DOI: *10.3141/2650-09*.

[MOCB21] Marlyn Montalvo-Martel, Alberto Ochoa-Zezzatti, Elías Carrum, and Denise Barzaga. “Design of an Urban Transport Network for the Optimal Location of Bus Stops in a Smart City Based on a Big Data Model and Spider Monkey Optimization Algorithm”. In: *Technological and Industrial Applications Associated with Intelligent Logistics* (2021), pp. 167–201.

[MV15] Nicolai Mallig and Peter Vortisch. “Modeling Car Passenger Trips in mobiTopp”. In: *Procedia Computer Science* Volume 52 (2015), pp. 938–943. ISSN: 1877-0509. DOI: *https://doi.org/10.1016/j.procs.2015.05.169*.

[Nam+18] Daisik Nam, Dingtong Yang, Sunghi An, Jiangbo Gabriel Yu, R. Jayakrishnan, and Neda Masoud. "Designing a Transit-Feeder System using Multiple Sustainable Modes: Peer-to-Peer (P2P) Ridesharing, Bike Sharing, and Walking". In: *Transportation Research Record* Volume 2672 (2018), pp. 754–763. eprint: *https://doi.org/10.1177/0361198118799031*.

[Sal72] Franz JM Salzborn. "Optimum bus scheduling". In: *Transportation Science* Volume 6 (1972), pp. 137–148.

[SHP11] Johannes Schlaich, Udo Heidl, and R. Pohlner. "Verkehrsmodellierung für die Region Stuttgart: Schlussbericht". In: *Unpublished* (2011).

[Stu11] Verband Region Stuttgart. "Mobilität und Verkehr in der Region Stuttgart 2009/2010: Regionale Haushaltsbefragung zum Verkehrsverhalten". In: *Schriftenreihe Verband Region Stuttgart 29* (2011), pp. 1–138.

[TMY20] Amirmahdi Tafreshian, Neda Masoud, and Yafeng Yin. "Frontiers in Service Science: Ride Matching for Peer-to-Peer Ride Sharing: A Review and Future Directions". In: *Service Science* Volume 12 (June 2020), pp. 44–60. ISSN: 2164-3962. DOI: *10.1287/serv.2020.0258*.

[WVJ22] Keji Wei, Vikrant Vaze, and Alexandre Jacquillat. "Transit Planning Optimization Under Ride-Hailing Competition and Traffic Congestion". In: *Transportation Science* Volume 56 (2022), pp. 725–749. eprint: *https://doi.org/10.1287/trsc.2021.1068*.

[ZYWY21] Yu Zhou, Hai Yang, Yun Wang, and Xuedong Yan. "Integrated line configuration and frequency determination with passenger path assignment in urban rail transit networks". en. In: *Transportation Research Part B: Methodological* Volume 145 (Mar. 2021), pp. 134–151. ISSN: 01912615. DOI: *10.1016/j.trb.2021.01.002*.

A. Appendix

In the following, we provide the detailed results of our experiments along with additional figures that did not make it in the evaluation.

Table A.1.: Results of the detour-DAG computation for varying station graphs and travel time multipliers.

Station graph	$\lvert\delta(\mathcal{P})\rvert$	α	#Explored edges	#Detour edges	#Detours	T_{detours} [s]
grid40	188 263	1.0	199 603	199 603	188 266	9
grid40	188 263	1.1	571 652	559 613	338 262	10
grid40	188 263	1.2	7 059 414	5 270 615	3 066 234	12
grid40	188 263	1.3	95 630 324	35 299 688	40 916 289	35
grid40	188 263	1.4	1 241 388 104	313 835 276	514 109 445	377
grid50	200 397	1.0	219 613	219 612	200 653	21
grid50	200 397	1.1	2 781 924	2 199 780	1 200 633	22
grid50	200 397	1.2	136 596 908	34 436 308	52 130 125	58
grid50	200 397	1.3	5 814 722 649	1 804 920 756	2 028 933 149	1 866

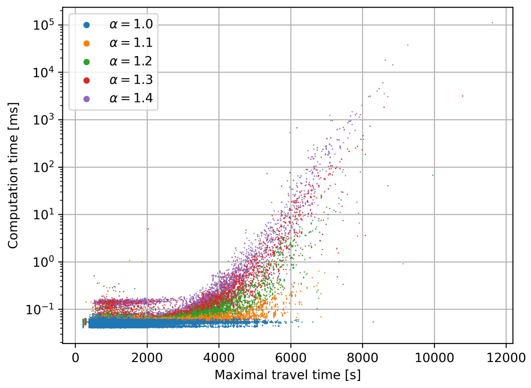


Figure A.1.: Computation time of the individual detour-DAGs by the maximal allowed travel time for station graph *grid40* and differing travel time multipliers α.

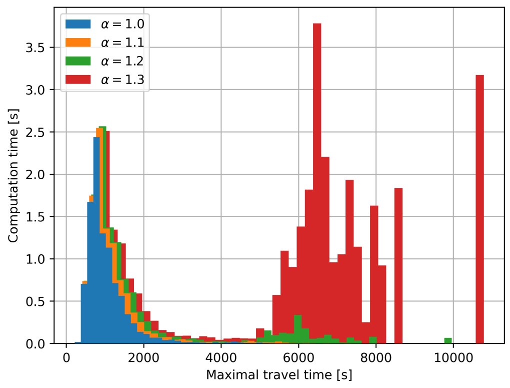


Figure A.2.: Histogram showing the cumulative computation time associated with detour-DAGs falling into the corresponding travel time bin, computed on station graph *grid40* with varying travel time multipliers α.

Table A.2.: Results of the baseline solver and the detour-solver (DS) on the station graph *grid40*.

Solver	α	ω [s]	$\lvert\Delta_B(\mathcal{P})\rvert$	T_{demands} [s]	$\lvert B\rvert$	T_{rides} [s]	ρ	T_{brf} [s]	T_{flow} [s]	$\lvert I\rvert$	$t(I)$	T_{total} [s]
Baseline	1.0	5	-	-	68 830	1	0.323	416	5 237	60 456	4	5 654
Baseline	1.0	10	-	-	68 830	1	0.488	393	5 219	59 239	4	5 613
Baseline	1.0	15	-	-	68 830	1	0.668	723	6 672	58 450	4	7 397
Baseline	1.0	20	-	-	68 830	1	0.838	569	5 860	58 028	5	6 430
Baseline	1.1	5	-	-	47 914	1	0.182	162	2 903	43 899	159	3 066
Baseline	1.1	10	-	-	47 914	1	0.275	188	3 053	42 973	143	3 242
Baseline	1.1	15	-	-	47 914	1	0.370	258	3 818	42 360	143	4 078
Baseline	1.1	20	-	-	47 914	1	0.459	219	3 409	41 975	139	3 629
Baseline	1.2	5	-	-	37 546	1	0.161	105	2 134	34 523	132	2 241
Baseline	1.2	10	-	-	37 546	1	0.239	242	2 887	33 749	120	3 130
Baseline	1.2	15	-	-	37 546	1	0.318	131	2 234	33 184	117	2 366
Baseline	1.2	20	-	-	37 546	1	0.394	136	2 371	32 760	109	2 508
Baseline	1.3	5	-	-	31 906	2	0.153	98	1 828	29 437	167	1 927
Baseline	1.3	10	-	-	31 906	2	0.221	67	1 660	28 761	147	1 728
Baseline	1.3	15	-	-	31 906	2	0.293	70	1 715	28 251	144	1 787
Baseline	1.3	20	-	-	31 906	2	0.361	85	1 800	27 825	135	1 887
DS	1.0	5	199 599	34	68 832	2	0.324	424	5 688	60 394	6	6 158
DS	1.0	10	199 599	35	68 832	2	0.494	654	7 407	59 125	5	8 107
DS	1.0	15	199 599	32	68 832	1	0.685	549	9 837	58 244	5	10 429
DS	1.0	20	199 599	39	68 832	2	0.855	609	10 384	57 765	5	11 043
DS	1.1	5	219 783	85	53 534	1	0.852	281	5 700	35 704	450	6 079
DS	1.1	10	219 783	108	53 534	2	1.260	622	9 432	33 839	405	10 174
DS	1.1	15	219 783	108	53 534	2	1.670	767	11 975	32 669	365	12 861
DS	1.1	20	219 783	109	53 534	2	2.072	1 216	9 732	31 918	349	11 069
DS	1.2	5	249 913	482	52 988	2	1.305	1 113	9 445	27 326	1 270	11 054
DS	1.2	10	249 913	433	52 988	2	1.925	628	7 303	25 077	1 005	8 378
DS	1.2	15	249 913	328	52 988	1	2.542	506	6 873	23 661	862	7 720
DS	1.2	20	249 913	671	52 988	2	3.158	1 272	8 096	22 715	775	10 053
DS	1.3	5	278 117	1 326	58 062	2	1.482	681	7 779	25 822	2 031	9 823
DS	1.3	10	278 117	1 821	58 062	2	2.192	1 329	9 100	23 472	1 642	12 287
DS	1.3	15	278 117	1 615	58 062	2	2.912	576	7 096	22 032	1 404	9 325
DS	1.3	20	278 117	1 633	58 062	2	3.631	1 202	8 874	21 147	1 320	11 746