Masterarbeit

Das dynamische Dial-a-Ride Problem mit Umstiegen

Sebastian Körner

Abgabedatum:	2. Februar 2026
Betreuer:	Prof. Dr. Marie Schmidt
	Kendra Reiter

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

Zusammenfassung

In ländlichen Räumen oder zu späten Uhrzeiten ist die Bereitstellung regulärer Buslinien aufgrund geringer Nachfrage oftmals nur in großen Zeitabständen oder gar nicht möglich. Sogenannte Bedarfsverkehr-Lösungen (On-Demand Transportation) sind eine Möglichkeit, um diese Lücken zu füllen und die Anbindung an andere ÖPNV-Angebote sicherzustellen. In dieser Masterarbeit lösen wir das deterministische, dynamische *line-based Dial-a-Ride Problem with transfers* (liDARPT). Das liDARPT ist ein liniengebundenes Vehicle Routing Problem mit festgelegten Zeitfenstern zur Abholung und Ablieferung der Fahrgäste, das auf dem viel erforschten Dial-a-Ride Problem (DARP) aufbaut. Wir stellen einen neuartigen Algorithmus vor, der das dynamische liDARPT mithilfe eines Event-Based Graph-Datentyps modelliert und mit Mixed-Integer Linear Programming optimal löst. Anschließend testen wir unsere Implementierung ausgiebig mithilfe realer Lang- und Kurzstrecken-Busnetzwerke. Dabei ergibt sich im Vergleich zum statischen liDARPT eine durchschnittliche Laufzeitverbesserung von 73 Prozent, während sich die Qualität der Ergebnisse um weniger als 4 Prozent verschlechtert.

Abstract

Providing regular bus connections to rural areas or at late hours can be a difficult and expensive task due to low demand. On-demand transportation offers a way to fill these gaps and ensure connectivity to other public transport services. In this master's thesis, we address the deterministic, dynamic line-based Dial-a-Ride Problem with transfers (liDARPT). The liDARPT is a line-based vehicle routing problem with predefined time windows for passenger pickup and drop-off, building on the well-studied Dial-a-Ride Problem (DARP). We present a novel algorithm that models the dynamic liDARPT using an Event-Based Graph and solves it optimally by leveraging Mixed-Integer Linear Programming. We extensively test our implementation using real-world long- and short-distance bus networks. Compared to the static liDARPT, this results in an average runtime improvement of 73 percent, while the solution quality degrades by less than 4 percent.

Inhaltsverzeichnis

1. Einleitung **5**
1.1. Stand der Forschung . . . 6
1.2. Problemdefinition . . . 8

2. Modellierung **10**
2.1. Routenoptionen . . . 10
2.2. Routen . . . 11
2.2.1. Erstellung des Routenbaums . . . 12
2.2.2. Aktualisierung des Routenbaums . . . 16

3. Event-Based Graph **20**
3.1. Beispiel für Event-Based Graph . . . 21
3.2. Konstruktion des Event-Based Graph . . . 23
3.2.1. Knoten . . . 23
3.2.2. Kanten . . . 25
3.3. Dynamischer Event-Based Graph . . . 26

4. MILP **29**

5. Rolling-Horizon Algorithmus **32**

6. Implementierung **34**
6.1. Codestruktur . . . 34
6.2. Ausgewählte Details der Implementierung . . . 41
6.2.1. Openrouteservice . . . 41
6.2.2. Zyklische Routen . . . 42
6.2.3. Zeitfenster . . . 42
6.2.4. DOcplex . . . 44
6.2.5. Eventerstellung . . . 44
6.2.6. Kantenentfernung . . . 45
6.2.7. Testing . . . 45

7. Auswertung **46**
7.1. Testinstanzen . . . 46
7.2. Auswertung der Vorverarbeitung . . . 49
7.3. Vergleich des statischen und dynamischen liDARPT . . . 54
7.3.1. Iterative Anwendung . . . 54
7.3.2. Dynamische Anwendung . . . 59

7.4. Weitere Erkenntnisse 62

8. Ausblick **64**
8.1. Echte dynamische Ausführung 64
8.2. Nichtdeterministisches liDARPT 64
8.3. Vorverarbeitung 65
8.4. Integration 65
8.5. Ohne MIP Gap 65
8.6. Multimodale Kapazität 66
8.7. Fahrtzeiten für Busse 66

9. Zusammenfassung und Fazit **67**

A. Variablenübersicht **70**

B. Abbildungen der Netzwerke **73**

Literaturverzeichnis **76**

1. Einleitung

Im Vergleich zum Individualverkehr bietet der öffentliche Personennahverkehr (ÖPNV) viele Vorteile. Durch den gemeinsamen Transport vieler Fahrgäste wird die insgesamt zurückgelegte Fahrstrecke verringert, was zu ökonomischen und ökologischen Einsparungen führt (siehe [SD21] und [Buh21]).

Eine hohe Auslastung des ÖPNV-Angebots ist dafür essenziell. Vor allem in ländlichen Regionen werden diese Angebote jedoch nicht ausreichend angenommen. Grund dafür ist beispielsweise die mangelnde Flexibilität der vorhandenen Buslinien, die oftmals nur wenige Male am Tag fahren und dabei häufig an Schulzeiten gebunden sind. Ein weiterer Grund sind lange Fahrtzeiten, die durch eine große Anzahl an Haltestellen und den damit einhergehenden Umweg bedingt sind.

In solchen Fällen kann der Bedarfsverkehr eine kosteneffiziente Ergänzung des ÖPNV-Angebots darstellen. Dabei werden kleinere Omnibusse mit 6-20 Sitzplätzen eingesetzt, deren Fahrpläne individuell an die Anforderungen der Passagiere angepasst werden können. Sie sind dabei weder an Linien noch an feste Zeitpläne gebunden. Üblicherweise müssen Transportanfragen, bestehend aus einem Zeitfenster und einem beliebigen Start- und Endpunkt, am Vortag angemeldet werden. Das Problem der optimalen Fahrplan-Erstellung für solche Rufbussysteme bezeichnet man als *Dial-a-Ride Problem* (DARP). In der Praxis erweist sich die Lösung des DARP als äußerst rechenintensiv. Aus dieser Motivation entstand das *linienbasierte Dial-a-Ride Problem* (liDARP) [BRS25], bei dem jedes Fahrzeug eine Linie, bestehend aus einer festen Abfolge an Haltestellen, besitzt. Die Fahrzeuge müssen entlang dieser Linie fahren, können aber Haltestellen, an denen kein Passagier ein- oder aussteigen möchte, überspringen. Die Fahrzeuge dürfen ihre Richtung außerdem nur ändern, wenn kein Insasse transportiert wird. Barth et al. [BRS25] zeigten, dass diese Vereinfachung des DARPT in statischen Anwendungsfällen sehr gute Ergebnisse liefert und eine geeignete Alternative zu bestehenden ÖPNV-Angeboten darstellt. Abbildung 1.1 gibt einen groben Vergleich zwischen beiden DARP-Varianten und anderen Verkehrsmitteln.

In realen Anwendungsfällen ist die Akzeptanz eines solchen Systems jedoch kritisch zu hinterfragen, da weiterhin eine frühzeitige Anmeldung der Passagiere erforderlich ist. Dieses Problem löst das dynamische liDARPT, mit dem wir uns in dieser Arbeit befassen. Darin können jederzeit neue Transportanfragen gestellt werden, woraufhin die Fahrpläne der Fahrzeuge sofort angepasst werden. Für Nutzer bedeutet das eine erhöhte Flexibilität in ihrer Transportplanung, was zu einer größeren Akzeptanz führt.

Um das dynamische liDARPT zu lösen, nutzen wir einen *Event-Based Graph*, der alle möglichen Zustände der Fahrzeuge als Knoten abbildet und diese mit gerichteten Kanten, die die Abfolge zweier Zustände darstellen, verbindet. Der Fahrplan eines Fahrzeugs kann anschließend über einen gerichteten Kreis vom Startzustand (leer) über eine Rei-

	Bus	Taxi	DARP	liDARPT
Route	Fest	Individuell	Flexibel	Fest
Zeitplan	Fest	Auf Anfrage	Auf Anfrage	Auf Anfrage
Modus	Geteilt	Nicht geteilt	Geteilt	Geteilt
Kapazität	Hoch	Gering	Mittel	Mittel
Zeitlich Flexibel	Nein	Ja	Ja	Ja
Räumlich Flexibel	Nein	Ja	Ja	Nein

Abb. 1.1.: Vergleich der DARP-Verwandten Probleme mit Linienbus-und Taxiverkehr (angelehnt an Ho et al. [HSK+18]).

he von Transportzuständen bis hin zum Endzustand (leer) des Fahrzeugs repräsentiert werden. Zur Suche solcher Fahrpläne verwenden wir ein *Mixed-Integer Linear Program* (MILP) und ein Programm zur mathematischen Optimierung, welches die Anzahl der transportierten Passagiere maximiert und deren zurückgelegte Strecke minimiert. Mit dieser Struktur orientieren wir uns an der Arbeit von Barth et al. [BRS25], die diese bereits für das statische liDARPT eingesetzt haben.

Unser eigener Beitrag besteht aus den folgenden drei Aspekten:

- Wir entwickeln einen neuartigen *Rolling-Horizon Algorithmus* zur Lösung des deterministischen, dynamischen liDARPT, der seine Lösung dynamisch an den Zustand der Fahrzeuge und Anfragen anpasst. Dafür erweitern wir den Event-Based Graph und das MILP um eine zeitliche Komponente. Außerdem erweitern wir die von Barth et al. [BRS25] vorgestellten Vorverarbeitungsschritte, sodass wir die Anzahl der Knoten im Event-Based Graph sowie die Anzahl der Nebenbedingungen im MILP weiter reduzieren können.

- Wir präsentieren den Routenbaum-Datentyp, den wir zur effizienten, dynamischen Verwaltung der möglichen und unmöglichen Routen für Transportanfragen verwenden.

- Wir implementieren den vorgestellten Algorithmus unter Verwendung des von Barth et al. [BRS25] entwickelten Python-Frameworks. Anschließend führen wir auf teilweise eigens entworfenen Testnetzwerken eine Vielzahl von Experimenten durch. Dabei überprüfen wir unter anderem die Eignung des dynamischen liDARPT als Heuristik zur Lösung des statischen liDARPT und den Einfluss, den die Anzahl der vorangemeldeten Anfragen auf das dynamische liDARPT hat.

1.1. Stand der Forschung

Das DARP ist ein viel erforschtes Problem in der angewandten Informatik. Es wurde erstmals im Jahr 1980 von Psaraftis [Psa80] vorgestellt. Dieser führt in seiner Arbeit sowohl die statische als auch die dynamische Variante des Problems ein und stellt

ein dynamisches Programm zur Lösung der Probleme vor. Savelsbergh [Sav85] bewies im Jahr 1985, dass das DARP NP-vollständig ist. Das DARP ist ein Sonderfall des Transport on Demand Problems (TOD), das sich mit dem Transport von Menschen und den damit einhergehenden Einschränkungen befasst. Eine Problemdefinition und Lösungsansätze für das allgemeine TOD-Problem stellen Cordeau et al. [CLPS07] zusammen. Als Erweiterung des DARP von Psaraftis wurde später auch das stochastische DARP eingeführt, wobei beispielsweise noch nicht eingetroffene Anfragen oder Verspätungen auf der Route eingeplant werden [XCC08]. Einen Überblick über diese und weitere Forschungen zum Dial-a-Ride Problem in den Jahren 2007 bis 2018 geben Ho et al.[HSK+18]. Zuvor behandelten Cordeau und Laporte[CL07] im Jahr 2007 die gängigsten Lösungsansätze für das Dial-a-Ride Problem.

Eine weitere Variante des DARP ist das *Dial-a-Ride with Transfer Problem* (DARPT). Anders als bei liDARPT sind die Fahrzeuge dabei in ihrer Route nicht an vorgegebene Haltestellen gebunden; es gibt jedoch Umsteigepunkte, an denen Fahrgäste von einem Fahrzeug in ein anderes umsteigen können. Eine einfache, aber gängige Version des DARPT hat nur einen einzigen zentralen Umsteigepunkt, an dem alle Transfers stattfinden. Diesen Ansatz verfolgen beispielsweise Gkiotsalitis et al. [GN23].

Ein Ansatz, der sich bei der dynamischen Variante des DARP etabliert hat, ist, nach zwei Heuristiken vorzugehen (siehe Kapitel „Construction insertion heuristics“ von Ho et al. [HSK+18]). Es wird eine einfache Heuristik zum Einfügen neuer Anfragen verwendet, um diese möglichst schnell zu verarbeiten. Anschließend wird in der Zeit, die zwischen den Anfragen verbleibt, eine komplexere Heuristik verwendet, um den Fahrplan zu optimieren. Die Entwicklung und Kombination verschiedener Heuristiken ist der Inhalt einer Vielzahl von Arbeiten. Andere heuristische Ansätze verwenden Varianten der lokalen Suche, wie beispielsweise Simulated Annealing (siehe [BCJ14]) oder Large Neighborhood Search (siehe [RP06]).

Neben den bereits genannten heuristischen Verfahren gibt es auch genaue Lösungsverfahren für das dynamische DARP. Gaul et al. [GKS21] verwenden einen Rolling-Horizon-Algorithmus mit einem Event-Based Graph. In einem Datensatz von 519 Anfragen, verteilt über 21 Stunden, erreichen sie eine durchschnittliche Antwortzeit von 2.9 Sekunden. Weiter konnten für mehr als 99.5% der Anfragen eine global optimale Lösung für den vorliegenden Zustand erreicht werden, was mit dem zuvor genannten heuristischen Verfahren bisher nicht möglich war.

Barth et al. [BRS25] haben das statische DARP-Model von Gaul et al. [GKP+25] für das statische liDARPT angepasst. Dafür wurden sowohl die Event-Based Graph-Datenstruktur als auch das Mixed-Integer Linear Program verändert. Außerdem entwickelten die Autoren ein Python-Framework zur Lösung des liDARPT, welches wir in dieser Arbeit aufgreifen.

1.2. Problemdefinition

Eine Übersicht über alle verwendeten Notationen ist in Anhang A aufgelistet.

Für die Definition des Problems und die verwendeten Variablen orientieren wir uns an Barth et al. [BRS25]

Wir betrachten ein dynamisches liDARPT-Problem, in dem $\mathcal{K}$ die Menge der *Fahrzeuge* darstellt. Jedes Fahrzeug $k \in \mathcal{K}$ fährt auf genau einer *Linie* $\sigma(k) \in \mathcal{I}$, die aus der geordneten Liste an Haltestellen $\lambda_i := (s_1, \ldots, s_{|\lambda_i|})$ besteht. Für alle Linien $i \in \mathcal{I}$ sei die Menge aller Haltestellen S_i von i in der Menge aller Haltestellen S enthalten. Ein Fahrzeug darf die Haltestellen in beide Richtungen durchfahren und kann dabei Haltestellen überspringen. Ein Fahrzeug darf dabei jedoch nur die Richtung ändern, falls kein Passagier transportiert wird. Linien können entweder linear oder zyklisch sein. Auf zyklischen Linien ist die letzte Haltestelle $s_{|\lambda_i|}$ mit der ersten Haltestelle s_1 verbunden, sodass ein Fahrzeug jede Haltestelle aus zwei Richtungen erreichen kann. Beispiel: In Abbildung 2.1 sind die orange- und lilafarbenen Linien zyklisch, und alle anderen Linien linear. Alle Fahrzeuge $\mathcal{K}_i$ einer Linie i haben dieselbe *Kapazität* Q_i. Diese Einschränkung macht die Fahrzeuge in der Planung einer Route austauschbar, was die Komplexität des Problems deutlich verringert.

Jede Linie i hat außerdem ein *Depot* 0_i, an dem alle Fahrzeuge ihre Fahrt beginnen und beenden müssen. Das Depot kann eine Haltestelle auf der Linie sein oder ein separater Standort. Außerdem hat jede Linie eine *Servicezeit* $[e(i), l(i)]$, d.h. ein Zeitfenster, in dem die Fahrzeuge der Linie Anfragen bearbeiten können.

Die Menge aller Linien bildet ein Busnetzwerk N, welches wir als Graph darstellen können (siehe Abbildung 2.1).

Für das so definierte Netzwerk treten n *Transportanfragen* $r \in R$ auf. Transportanfragen können dynamisch während der Ausführung des Programms zu einer *Registrierungszeit* eintreffen. Die Menge $R(t)$ enthält alle Transportanfragen, die bis zum Zeitpunkt t registriert wurden. Eine Transportanfrage r besteht aus einer Anzahl $c(r)$ an Personen, die transportiert werden, und einem *Start*- und *Zielpunkt* r^+ und r^-, die in der Menge S aller Haltestellen enthalten sein müssen. Außerdem sei ein *Zeitfenster* bestehend aus $e(r)$ und $l(r)$ gegeben, welches die früheste Startzeit und die späteste Ankunftszeit beschreibt. Jede Transportanfrage r hat außerdem eine *maximale Transportdauer* $\mathcal{L}_r$. Transportanfragen r können akzeptiert $q_r = 1$ oder abgelehnt $q_r = 0$ werden.

Eine Lösung für das dynamische liDARPT zum Zeitpunkt t besteht aus einem *Fahrplan* für jedes Fahrzeug $k \in \mathcal{K}$ und einer *Route* für jede akzeptierte Transportanfrage $\{r \mid r \in R(t) \wedge q_r = 1\}$. Der Fahrplan eines Fahrzeugs besteht aus einer geordneten Liste der Haltestellen, die das Fahrzeug anfährt (beginnend und endend mit dem Depot), den Ankunfts- und Abfahrtszeiten der jeweiligen Haltestellen sowie den zu- und aussteigenden Passagieren an jeder Haltestelle. Eine Route beschreibt die Abfolge der Streckenabschnitte, die ein Fahrgast nutzt, um von seinem Startpunkt zum Zielpunkt zu gelangen. Bei der Suche nach Fahrplänen optimieren wir zwei Faktoren:

1. Wir maximieren die Anzahl der akzeptierten Anfragen
2. Wir minimieren die Reisedauer für die Passagiere

2. Modellierung

Wir gehen im Folgenden genauer darauf ein, wie die einzelnen Elemente des Problems modelliert werden. OBdA sei das Netzwerk N zusammenhängend. Ein Fahrzeug fährt entweder in auf- oder absteigender Richtung entlang der Haltestellen. Die Menge S^T der *Umsteigepunkte* sei die Vereinigung der Schnittmengen der Linien. Für jede Linie i wird der kürzeste Abstand $w(s_1, s_2) > 0$ zwischen den Haltestellen s_1 und s_2 mit $s_1, s_2 \in S_i$ in Sekunden gemessen. Die Dauer der Umstiege b sei fix und unabhängig von der Anzahl der ein- und aussteigenden Passagiere und wurde im Vorhinein aus einem durchschnittlichen Erfahrungswert berechnet.

Für jede Transportanfrage r haben wir das Zeitfenster $[e(r), l(r)]$ bereits definiert. Wir nehmen an, dass wir außerdem die Dauer der kürzesten Strecke auf dem Netzwerk $t_{\min}(r)$ zwischen r^+ und r^- kennen. Daraus können wir Zeitfenster für den Startpunkt r^+ und den Endpunkt r^- berechnen. Das bekannte Zeitfenster der Transportanfrage r besteht aus einem frühesten Startpunkt $e(r^+) = e(r)$ und einem spätesten Endpunkt $l(r^-) = l(r)$. Wir können nun die späteste Abfahrtszeit $l(r^+) = l(r) - t_{\min}(r)$ und die früheste Ankunftszeit $e(r^-) = e(r) + t_{\min}(r)$ berechnen. Somit erhalten wir zwei Zeitfenster $[e(r^+), l(r^+)]$ und $[e(r^-), l(r^-)]$. Im liDARPT kommt es oft vor, dass der Start- und Zielpunkt einer Reise nicht auf derselben Linie liegen. Folglich sind Umstiege nötig, um zum Ziel zu gelangen. Häufig gibt es unterschiedliche Abfolgen an Umstiegen, aus denen wir wählen können-

2.1. Routenoptionen

Um einen Nutzer vom Einstiegspunkt zum Ausstiegspunkt zu transportieren, können oft unterschiedliche Routen verwendet werden. Unter einer Route verstehen wir einen eindeutigen Pfad durch das Netzwerk, der eine oder mehrere Linien durchquert und vom Start- zum Zielpunkt des Fahrgastes führt (siehe Abbildung 2.1). Routen können variieren in ihrer Transportdauer, der Anzahl der Umstiege, der Anzahl an Linien und der Anzahl an Haltestellen, die auf dem Weg liegen. Umstiege können wir dabei in zwei Unterschritte Unterteilen: Eine *Ausstiegsaktion* und eine *Einstiegsaktion*. Einen Umstieg der Anfrage $r \in R$ von Linie i nach Linie j mit $i, j \in \mathcal{I}$ am Umsteigepunkt $s \in S^T$ wird wie folgt ausgedrückt: $\rho^{i-}_{s,r,d}$ beschreibt eine Ausstiegsaktion aus Linie i und $\rho^{j+}_{s,r,d'}$ eine Einstiegsaktion in Linie j. Die gewünschte Fahrtrichtung wird durch d und d' angegeben, wobei 1 aufsteigend und 0 absteigend bedeutet. O.B.d.A. sei d für alle Ausstiegsaktionen 1. Für eine Anfrage r besteht eine Route ϕ aus einer Abfolge von mindestens zwei Aktionen: Die Einstiegsaktion bei r^+ und der Ausstiegsaktion bei r^-. Eine Route mit einem Umstieg sieht dann beispielsweise so aus: $(\rho^{i+}_{r^+,r,d}, \rho^{i-}_{s,r,1}, \rho^{j+}_{s,r,d'}, \rho^{j-}_{r^-,r,1})$. Wir können die Route weiter zerteilen in die einzelnen Abschnitte, die mit einer Linie gefahren

werden. Für das obige Beispiel ist die Route $\phi := ((\rho^{i+}_{r^+,r,d}, \rho^{i-}_{s,r,1}), (\rho^{j+}_{s,r,d'}, \rho^{j-}_{r^-,r,1}))$, und besteht aus zwei *Streckenabschnitten* $\psi^i_{r^+,s,r,d}$ und $\psi^j_{s,r^-,r,d'}$. Für zyklische Linien gibt es zwei Streckenabschnitten, die zwischen denselben Haltestellen genutzt werden können: Das Fahrzeug kann entweder in aufsteigender oder absteigender Richtung fahren. Die Fahrtrichtung des Streckenabschnitts entspricht dabei der Fahrtrichtung der zugehörigen Einstiegsaktion. Ein Streckenabschnitt kann Teil von mehreren Routen sein. Φ^r sei die Menge aller möglichen Routen für die Anfrage r und $\Psi(r)$ die Vereinigung aller Streckenabschnitte in Φ^r. Weiter sei $\mathrm{P}(r)$ die Vereinigung aller Aktionen der Streckenabschnitte $\Psi(r)$ und P die Gesamtmenge aller Aktionen $\bigcup_{r \in R} \mathrm{P}(r)$. Um später dynamisch Routen ausschließen zu können, unterscheiden wir offene Φ^r_o und geschlossene Routen Φ^r_c. Offene Routen sind im aktuellen Zustand der Fahrzeuge erfüllbar, während geschlossene Routen nicht mehr möglich sind. Zu Beginn der Routenplanung, bevor die Passagiere der Anfrage r abgeholt wurden, sind alle zeitlich plausiblen Routen erfüllbar, also offen $\Phi^r_o = \Phi^r$. Für Aktionen ρ und Streckenabschnitte ψ definieren wir außerdem die folgenden Funktionen: $i(\psi/\rho)$ gibt die Linie, $d(\psi/\rho)$ gibt die Fahrtrichtung, $s(\psi/\rho)$ die Haltestelle und $r(\psi/\rho)$ die Anfrage zurück.

2.2. Routen

Wir gehen nun genauer darauf ein, wie wir die möglichen Routen für eine gegebene Transportanfrage r und ein gegebenes Netzwerk N berechnen können.

Wir müssen dabei besonders beachten, dass die Menge der möglichen Routen dynamisch angepasst werden muss. Beispielsweise wird durch die Wahl einer Richtung auf einer zyklischen Linie jede Route unmöglich, die mit der Fahrt in die entgegengesetzte Richtung beginnt. Die Menge der möglichen Routen für jede Transportanfrage r kodieren wir deshalb in einem *Routenbaum* T^r_Φ. Die Wurzel des Routenbaums T^r_Φ sei der Startpunkt r^+ der Transportanfrage r, welcher durch einen *Routenknoten* ν_{r^+} dargestellt wird. Weiter sei jedes Blatt ν_{r^-} des Baums am Ausstiegspunkt r^-. Jeder Pfad von der Wurzel zu einem Blatt des Baums entspricht einer eindeutigen Route ϕ^r_j. Jeder Knoten dazwischen entspricht einem Umstieg, den wir im Routenbaum über seine Haltestelle identifizieren. Jede Kante im Routenbaum entspricht einem Streckenabschnitt, der zwischen den damit verbundenen Haltestellen verläuft. In Abbildung 2.2 ist ein Beispiel für einen solchen Routenbaum gegeben.

Eine weitere wichtige Funktion des Routenbaums ist die Berechnung und Aktualisierung von Zeitfenstern für die Aktionen und Streckenabschnitte der Anfrage. Wir kennen bereits die kürzeste Reisezeit $w(\psi^i_{s_1,s_2,r,d})$ für alle Streckenabschnitte $\psi^i_{s_1,s_2,r,d} \in \Psi(r)$. Betrachten wir nun die längste Reisezeit $w_{\max}$ von $\psi^i_{s_1,s_2,r,d}$. Diese tritt ein, wenn das Fahrzeug an jeder Haltestelle zwischen s_1 und s_2 halten muss. Beispiel: Die von r^+ ausgehende Kante in Abbildung 2.1, die den Streckenabschnitt $\psi^i_{r^+,s_1,r,1}$ beschreibt und deren kürzeste Reisezeit durch $w(r^+, s_1)$ gegeben ist, hat eine maximale Reisezeit $w_{\max}(\psi^i_{r^+,s_1,r,1})$ von $w(r^+, s_6) + w(s_6, s_1) + b$.

Mit dieser Information berechnen wir anschließend (1) Für alle Routenknoten des

Routenbaums die kürzeste und längste Reisezeit von der Wurzel bis zum Erreichen des Knotens und (2) Für jeden Knoten ν des Routenbaums die frühesten Startzeiten und spätesten Endzeiten von $\psi \in \Psi_\nu$ in Relation zu ν. Dabei sei Ψ_ν die Menge aller Streckenabschnitte im Subbaum mit Wurzel ν.

Mit diesen Informationen können wir die Wurzel des Routenbaums stets so wählen, dass sie die nächste „Entscheidung“ beim Transport einer Transportanfrage widerspiegelt.

Die Zeitfenster der Streckenabschnitte und Aktionen im Baum berechnen wir zunächst in Relation zur Wurzel. Anschließend addieren wir das Zeitfenster der Wurzel, sodass sich ein globales Zeitfenster ergibt.

Mithilfe dieses Verfahrens bestimmen wir zunächst sehr große Zeitfenster, die sich mit dem Voranschreiten des Transports immer weiter verkleinern. Beispielsweise wird das Startzeitfenster $[e(r^+), l(r^+)]$ zu einem eindeutigen Zeitpunkt $e(r^+) = l(r^+)$, sobald die Fahrgäste an r^+ abgeholt wurden.

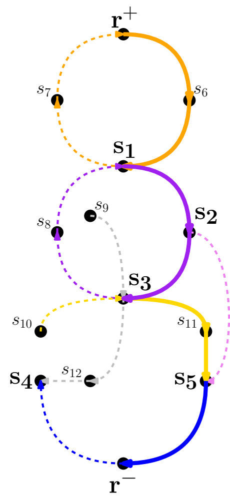


Abb. 2.1.: Ein möglicher Plan für ein liDARPT-Netzwerk, das aus fünf farblich gekennzeichneten Linien besteht. Eine beispielhafte Route ist hervorgehoben.

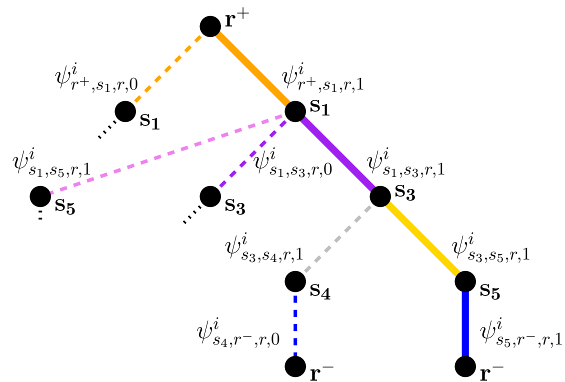


Abb. 2.2.: Der Routenbaum von Punkt r^+ nach r^- im links dargestellten Netzwerk. Redundante Subgraphen sind durch schwarz gepunktete Linien angedeutet. Dieselbe Route ist hervorgehoben.

2.2.1. Erstellung des Routenbaums

Wir gehen nun genauer darauf ein, wie wir den Routenbaum erstellen und nehmen dafür die Erstellung des Routenbaums zur Hilfe.

Zuerst treffen wir jedoch eine weitere Einschränkung für die gesuchten Routen. Wir verbieten den Passagieren, auf eine bereits verwendete Linie zurückzusteigen. Ein Fahrgast beginnt seine Fahrt beispielsweise auf Linie i und steigt später auf Linie j um. Der gleiche Fahrgast darf nicht zu einem späteren Zeitpunkt auf die Linie i zurück umsteigen, da dies eine vermeidbare Unannehmlichkeit für den Fahrgast bedeutet.

Weiterhin darf jede Haltestelle höchstens einmal besucht werden.

Zeile 1 Wir verwenden, wie bei Barth et al. [BRS25], ein vereinfachtes *Liniennetz* N_l, das die Verbindung der Linien durch Umsteigeknoten darstellt. Das Liniennetz enthält demnach keine Haltestellen, die nicht in S^T sind. Suchen wir nach einem Pfad von r^+ nach r^-, so untersuchen wir zunächst die Linien $i, j \in \mathcal{I}$, auf denen die Start- und Zielpunkte liegen. Handelt es sich bei den Haltestellen um keine Umsteigepunkte, so fügen wir einen temporären Knoten ein und verbinden ihn mit den Umsteigepunkten der zugehörigen Linie.

Zeile 5-9 Wir erstellen einen initialen Wurzelknoten ν_{r^+} für den Routenbaum T^r_Φ aus dem Einstiegspunkt der Anfrage. Die kürzeste zeitliche Entfernung $e(\nu_{r^+})$ des Wurzelknotens ist 0 (Da wir die Zeitfenster der Routenknoten in Relation zum Wurzelknoten betrachten). Die Liste der vorherigen Streckenabschnitte *prevSplits* und Haltestellen *prevStops* sei für den Wurzelknoten leer, da r^+ der Startpunkt der Routen ist. Wir rufen anschließend *MakeGraph* auf,um den Routenbaum mithilfe von Tiefensuche aufzubauen.

Zeile 13-20 Wir beginnen den rekursiven Algorithmus *MakeGraph*, indem wir für die übergebene Haltestelle s prüfen, ob sie gleich dem Zielpunkt r^- ist. Falls ja, überprüfen wir weiter, ob die gefundene Route kürzer ist als die maximale Transportdauer $\mathcal{L}_r$. Trifft diese Bedingung zu, so haben wir eine gültige Route gefunden. Wir fügen den Routenknoten zur Liste der Blattknoten und dessen Route zu den offenen Routen Φ^r_o hinzu. Anschließend geben wir „Wahr“ zurück. Ist die Route länger als erlaubt, geben wir „Falsch“ zurück.

Zeile 21-37 Falls die aktuelle Haltestelle nicht der Zielpunkt ist, prüfen wir, welche adjazenten Haltestellen s_c im Liniennetz N_l wir als mögliche Kindknoten verwenden können. Für die Haltestelle s_c überprüfen wir zunächst, ob sie in der aktuellen Route bereits besucht wurde. Falls ja, so geben wir „Falsch“ zurück. Ansonsten suchen wir mithilfe von *findSplits* alle möglichen Streckenabschnitte zwischen den Haltestellen s und s_c. Dabei beachten wir, dass unterschiedliche Linien und Fahrtrichtungen möglich sind. Für die gefundenen Streckenabschnitte $\psi \in \Psi_{\text{new}}$ überprüfen wir, ob die Linie $i(\psi)$ des Streckenabschnitts in der aktuellen Route bereits besucht wurde. Falls ja, so machen wir mit dem nächsten Streckenabschnitt weiter. Andernfalls ist der gefundene Streckenabschnitt zulässig und wir beginnen damit, einen neuen Kindknoten $\nu_{\text{new}} = \nu_{s_c}$ einzufügen. Der Elternknoten von ν_{s_c} sei der aktuelle Routenknoten ν. Die Menge *prevStops* der bereits besuchten Haltestellen übernehmen wir vom aktuellen Routenknoten ν und fügen die aktuelle Haltestelle s ein. Ebenso übernehmen wir die Menge *prevStops* der bereits befahrenen Streckenabschnitte von ν und fügen den neuen Streckenabschnitt ψ hinzu. Anschließend errechnen wir die kürzeste Entfernung $e(\nu_{\text{new}})$von ν_{new} zum Wurzelknoten aus $e(\nu)$, addiert mit der Haltezeit b und der minimalen Dauer des Streckenabschnitts $w(\psi)$ (siehe Abbildung 2.3). Abschließend rufen wir die *MakeGraph*-Funktion rekursiv auf für den neu erstellten Kindknoten ν_{new} und die dazugehörige Haltestelle s_c. Ist das

zurückgegebene Ergebnis „Wahr“, d.h. der Knoten ist Teil einer gültigen Route, so fügen wir den Kindknoten zur Liste *children* des aktuellen Routenknotens hinzu.

Zeile 38-41 Nachdem wir alle adjazenten Haltestellen und die dazugehörigen Streckenabschnitte durchlaufen haben, überprüfen wir, ob die Liste der Kindknoten von ν leer ist. Falls ja, so konnte keine gültige Route von ν aus gefunden werden und wir geben „Falsch“ zurück. Andernfalls geben wir „Wahr“ zurück.

Zeile 10-12 Nachdem wir den Routenbaum aufgebaut haben, bestimmen wir nun die Zeitfenster der Streckenabschnitte und Aktionen. Dafür durchlaufen wir den Routenbaum rückwärts, beginnend bei den Blattknoten mithilfe der Funktion *Recurse*. Dabei berechnen wir für jeden Knoten die Distanz zu allen Streckenabschnitten in ihrem Subbaum. Anschließend verwenden wir *SetTimeWindow*, um die Globalen Zeitfenster für alle Streckenabschnitte und Aktionen im Routenbaum zu berechnen.

Recurse Wir betrachten nun genauer die Funktion Recurse. Diese erhält den aktuellen und vorherigen Routenknoten ν_c und ν_p. Da wir den Routenbaum rückwärts durchlaufen, hat ν_c eine geringere Tiefe als ν_p und ist somit näher an der Wurzel. Wir überprüfen zunächst, ob der aktuelle Knoten gleich $Null$ ist und beenden die Funktion, falls dies zutrifft. Als nächstes prüfen wir, ob ν_p gleich $Null$ ist. Dieser Fall tritt nur ein, falls ν_c ein Blattknoten ist. Wir reagieren, indem wir die Funktion rekursiv mit ν_c.parent und ν_c aufrufen. Ist weder ν_c noch ν_p unbelegt, so betrachten wir den zugehörigen Streckenabschnitt ψ. Wir aktualisieren zunächst das Ende des Zeitfensters von ν_c. Anschließend setzen wir das Zeitfenster ν_c.edgeTimeWindows(ψ) auf $[0, w_{\max}(\psi)]$. Das bedeutet, dass die früheste Startzeit von ψ in Relation zu ν_c gleich 0 ist (da ψ mit ν_c verbunden ist) und die späteste Ankunftszeit von ψ in Relation zu ν_c gleich der maximalen Reisezeit von ψ ist. Anschließend überprüfen wir für alle Streckenabschnitte ψ' in ν_p.edgeTimeWindows, ob sie in ν_c.edgeTimeWindows enthalten sind (d.h., ob wir in vorherigen Durchläufen von *Recurse* bereits eine andere Route betrachtet haben, die ν_c und ψ' enthält). Falls nicht, so addieren wir die minimale Distanz $w(\psi)$ auf den frühesten Startzeitpunkt des vorherigen Zeitfensters und die maximale Distanz $w_{\max}(\psi)$ auf den spätesten Startzeitpunkt des vorherigen Zeitfensters. Ist der Streckenabschnitt stattdessen bereits in ν_c.edgeTimeWindows enthalten, so bilden wir das Minimum aus dem vorherigen und aktuellen frühesten Startzeitpunkt und das Maximum aus dem vorherigen und aktuellen spätesten Endzeitpunkt. Wir erhalten somit das größtmögliche Zeitfenster für den Streckenabschnitt ψ'. Ein Beispiel für den Inhalt der edgeTimeWindows-Dictionarys ist in Abbildung 2.4 abgebildet.

SetTimeWindows In *Recurse* haben wir die Zeitfenster der Streckenabschnitte in Relation zu den Routenknoten berechnet. Nun wollen wir die globalen Zeitfenster für alle Streckenabschnitte und Aktionen berechnen. Die Funktion SetTimeWindows bekommt einen Routenbaum T^r_Φ übergeben, für den zunächst die aktuelle Wurzel $\nu = T.root$ und

Algorithmus 1 Erstellung des Routenbaums

```
1: Sei r die gegebene Anfrage, für die wir T^r_Φ erstellen
2: Sei s.edgeTimeWindows ein Dictionary, das für jeden Knoten s das relative Zeitfens-
   ter jeder Kante im Subgraph von s speichert
3: Erstelle N_l                                                    ▷ Liniennetz
4: Die Funktion findSplits(s_1, s_2, N_l) gibt alle Streckenabschnitte zwischen s_1 und s_2
   im Liniennetz zurück
5: T^r_Φ.root=new ν_{r+}
6: (e(T^r_Φ.root), l(T^r_Φ.root)) = (0, 0)
7: T^r_Φ.root.prevSplits= ∅
8: T^r_Φ.root.prevStops= ∅
9: T^r_Φ = MakeGraph(N_l, T^r_Φ.root, r^+)
10: for Blattknoten ν_l ∈ T^r_Φ.leaves do
11:     Recurse(ν_l, Null)
12: SetTimeWindows(T^r_Φ)
13: procedure MakeGraph(LineNetwork N_l,TreeNode ν,Stop s)
14:     if s == r^- then                     ▷ falls aktuelle Haltestelle Endpunkt ist
15:         if e(r) + e(ν) < ℒ_r then
16:             T^r_Φ.leaves= T^r_Φ.leaves ∪ ν
17:             Φ^r_o+ = (ν.prevStops)
18:             return Wahr
19:         else
20:             return Falsch                        ▷ Gefundene Route ist zu lang
21:     for Kindknoten s_c von s in N_l do
22:         if s_c in ν.prevStops then      ▷ Falls Haltestelle auf aktueller Route liegt
23:             return Falsch             ▷ Erstellung des Subgraphs ist fehlgeschlagen
24:         else
25:             Ψ_new =findSplits(s, s_c, N_l)
26:             for ψ ∈ Ψ_new do
27:                 for ψ_p ∈ ν.prevSplits do
28:                     if i(ψ) = i(ψ_p) then
29:                         continue
30:                 ν_new = new ν_{s_c}                  ▷ Erstelle neuen Routenknoten
31:                 ν_new.parent = ν
32:                 ν_new.prevStops= ν.prevStops∪s
33:                 ν_new.prevSplits= ν.prevSplits∪ψ
34:                 e(ν_new) = e(ν) + b + w(ψ)
35:                 l(ν_new) = 0       ▷ Initialisierung. Richtiger Wert wird später gesetzt
36:                 if MakeGraph(N_l, ν_new, s_c) then
37:                     ν.addChild(ν_new)
38:     if ν.children==∅ then
39:         return Falsch
40:     else
41:         return Wahr
```

Algorithmus 2 Reverse Traversal

```
1: procedure Recurse(TreeNode ν_c, TreeNode ν_p)
2:     if ν_c == Null then return
3:     if ν_p ≠ Null then
4:         ψ = ν_p.prevSplits.last
5:         l(v_c) = max(l(v_c), l(ν_p) − w(ψ))
6:         ν_c.edgeTimeWindows(ψ) = (0, w_max(ψ))
7:         for ψ′,prevMin,prevMax∈ ν_p.edgeTimeWindows do
8:             if ψ′ ∉ ν_c.edgeTimeWindows then
9:                 ν_c.edgeTimeWindows(ψ′) = (prevMin+w(ψ),prevMax+w_max(ψ))
10:            else
11:                currMin,currMax = ν_c.edgeTimeWindows(ψ′)
12:                newMin = min(currMin,prevMin+w(ψ))
13:                newMax = max(currMax,prevMax+w_max(ψ))
14:                ν_c.edgeTimeWindows(ψ′) = (newMin,newMax)
15:    else
16:        l(v_c) = l(r)
17:        Recurse(ν_c.parent, ν_c)
```

deren Haltestelle $s(\nu)$ bestimmt wird. Falls in einem vorherigen Durchlauf bereits Zeitfenster für die Aktionen P berechnet wurden, so setzen wir diese auf ihre Standardwerte 0 und ∞ zurück. Anschließend durchlaufen wir die Menge der Streckenabschnitte ψ im Subgraph der Wurzel ν. Wir setzen das Zeitfenster des Streckenabschnitts $[e(\psi), l(\psi)]$ auf das relative Zeitfenster in ν.edgeTimeWindows addiert mit dem Zeitfenster des Wurzelknotens und dem Startzeitfenster der Anfrage.
Anschließend betrachten wir die beiden Aktionen ρ^+ und ρ^-, aus denen der Streckenabschnitt ψ besteht. Das Zeitfenster der Einstiegsaktion besteht aus dem frühesten und spätesten Startzeitpunkt aller Streckenabschnitte in denen er enthalten ist. Das Zeitfenster der Ausstiegsaktion wiederum besteht aus dem frühesten und spätesten Endzeitpunkt aller Streckenabschnitte in denen er enthalten ist. Diese berechnen wir aus dem vorhandenen frühesten Startzeitpunkt und spätesten Endzeitpunkt durch Addition beziehungsweise Subtraktion der kürzesten Distanz $w(\psi)$.

2.2.2. Aktualisierung des Routenbaums

Im vorherigen Abschnitt haben wir den Routenbaum erstellt. Diesen Schritt können wir durchführen, sobald eine neue Anfrage eingetroffen ist. Anschließend planen wir optimale Fahrpläne für die Fahrzeuge im Netzwerk und geben diese an die Fahrer weiter. Sobald eine Anfrage abgeholt wurde und Teile der geplanten Route durchquert sind, müssen die möglichen Routen und Zeitfenster angepasst werden, bevor eine Neuoptimierung der Routen möglich ist.
Der oben beschriebene Routenbaum kann auch als Entscheidungsbaum betrachtet werden. Die Route vom Start- zum Zielknoten ist dabei eine Folge von Umsteigepunkten.

Algorithmus 3 Set Time Windows

```
1: procedure SetTimeWindows(RouteTree T^r_Φ)
2:     ν = T^r_Φ.root
3:     s = s(ν)
4:     for ρ ∈ P do
5:         e(ρ) = ∞
6:         l(ρ) = 0
7:     for ψ ∈ ν.edgeTimeWindows do
8:         (e(ψ), l(ψ)) = ν.edgeTimeWindows+(e(ν), l(ν)) + (e(r⁺), l(r⁺))
9:         (ρ⁺, ρ⁻) = ψ
10:        e(ρ⁺) = min(e(ρ⁺), e(ψ))
11:        l(ρ⁺) = max(l(ρ⁺), l(ψ) − w(ψ))
12:        e(ρ⁻) = min(e(ρ⁻), e(ψ) + w(ψ))
13:        l(ρ⁻) = max(l(ρ⁻), l(ψ))
```

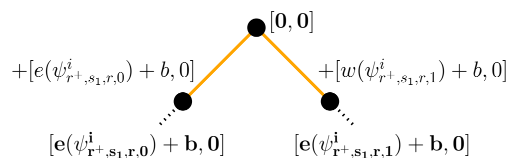


Abb. 2.3.: Darstellung zur Erstellung des Routenbaums. Für den Startknoten sei $e(\nu_{r^+}) = l(\nu_{r^+}) = 0$. Die folgenden Knoten werden durch Addition mit der kürzesten Reisedauer für die Streckenabschnitte berechnet.

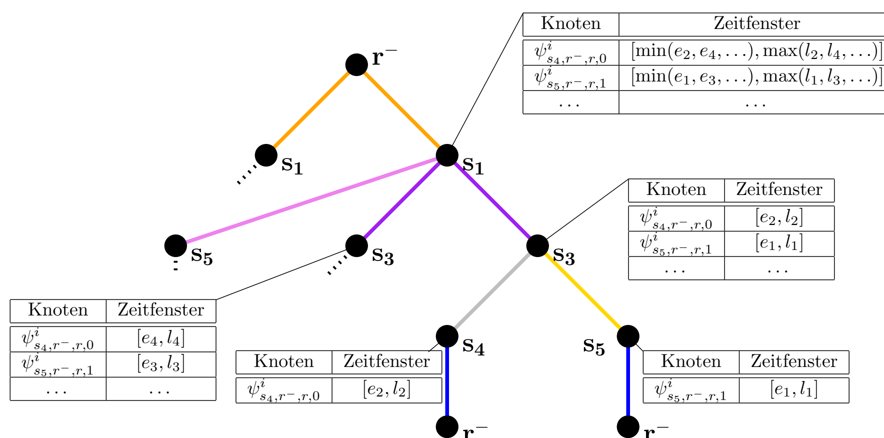


Abb. 2.4.: Ausschnitt aus den Zeitfenster-Tabellen des Routenbaums

Hat das Fahrzeug die Entscheidung getroffen, einen Umsteigepunkt zu wählen, so entfallen alle anderen möglichen Umsteigepunkte an dieser Stelle der Route. Außerdem können wir bereits besuchte Umsteigepunkte aus dem Routenbaum löschen. Hierzu ein Beispiel: Eine Transportanfrage r befindet sich zum Zeitschritt t am Startpunkt. Folglich sind noch alle Routenoptionen offen. Zum nächsten Zeitschritt $t+1$ sei die Transportanfrage abgeholt (folglich ist der Startzeitpunkt eindeutig bestimmt und $e(r^+) = l(r^+)$) und fährt entlang der gewählten Kante ψ, ist jedoch noch nicht am nächsten Umsteigepunkt s_t angekommen. Wir berechnen die Routenoptionen wie folgt neu:
Die erste Entscheidung, die am Wurzelknoten getroffen wird, welcher erste Routenabschnitt gewählt wird, ist nun fixiert. Diese Entscheidung fällt somit weg und muss nicht länger im Routenbaum beachtet werden. Als neue Wurzel des Routenbaums wählen wir den gewählten Kindknoten ν_{s_t}. Da wir zuvor bereits alle relativen Zeitfenster für jeden Knoten berechnet haben, ist kein weiteres Durchlaufen des Baums notwendig. Wir berechnen lediglich das Zeitfenster des neuen Wurzelknotens $[e(\nu_{s_t}), l(\nu_{s_t})]$ in Relation zum Startzeitpunkt. In diesem Fall $[w(\psi), w_{\max}(\psi)]$[1].

Anschließend überprüfen wir für die Blätter, ob sie noch von der Wurzel des Baums erreichbar sind. Falls nicht, so schließen wir die zugehörige Route. In Abbildung 2.2 würde beispielsweise die Wahl der Kante $\psi^i_{r^+,s_1,r,1}$ ein Wegfallen des linken Subgraphs mit $\psi^i_{r^+,s_1,r,0}$ bedeuten. Wir haben somit den neuen Routenbaum $T^r_\Phi(t+1)$ aus $T^r_\Phi(t)$ abgeleitet.

[1]Ein präziseres Modell könnte die aktuellen GPS-Koordinaten des Fahrzeugs auslesen und damit die genaue Fahrtzeit anhand des aktuellen Fortschritts besser abschätzen

Algorithmus 4 Aktualisierung des Routenbaums

Aktueller Zeitpunkt sei t
Bestimme die route = ϕ_j^r, die bisher verfolgt wurde
Die zuletzt erreichte / verlassene Haltestelle der Route sei s_{last} zum Zeitpunkt t_{last}
route(t) sind alle Streckenabschnitte, die bis zum Zeitpunkt t durchgeführt wurden
newSplits = route(t)-route(t-1)
for Streckenabschnitte ψ $\in$newSplits **do**
  for $\nu_c \in T_\Phi^r$.children **do**
    if ν_c.prevSplits.last== ψ **then** $T_\Phi^r.root = \nu_c$
  if $s(T_\Phi^r$.root) == s **then** ▷ Haltestelle erreicht aber noch nicht verlassen
    $e(T_\Phi^r$.root) $= t_{\text{last}} - e(r^+)$ ▷ späteste Abfahrtszeit bleibt gleich
  else
    $\psi_c = T_\Phi^r$.root.last
    $e(T_\Phi^r$.root) $= t_{\text{last}} - e(r^+) + w(\psi_c)$
    $l(T_\Phi^r$.root) $= t_{\text{last}} - e(r^+) + w_{\max}(\psi_c)$
for Blattknoten $\nu_l \in T_\Phi^r$.leaves **do**
  if route(t)$\notin$ ν_l.prevSplits **then**
    T_Φ^r.leaves = T_Φ^r.leaves$\setminus\nu_l$
    $\Phi_o^r- = (\nu$.prevStops)
    $\Phi_c^r+ = (\nu$.prevStops)
SetTimeWindows(T_Φ^r)

3. Event-Based Graph

Zuvor haben wir bereits die möglichen Routen für Transportanfragen mithilfe des Routenbaums bestimmt. Da wir mehrere Transportanfragen nacheinander oder sogar gleichzeitig transportieren möchten, können wir die Routen der Transportanfragen nicht einzeln betrachten. Stattdessen suchen wir nun Streckenabschnitte der Routen, die gleichzeitig transportiert werden können. Dafür verwenden wir einen sogenannten Event-Based Graph $G = (V, A)$, dessen Knoten V, die wir Events nennen, die möglichen Zustände der Fahrzeuge darstellen und dessen Kanten A eine Abfolge der Fahrzeugzustände repräsentieren. Dieser Datentyp wurde erstmals von Gaul et al. [GKS22] vorgestellt und von Barth et al. [BRS25] für das statische liDARPT aufgegriffen.

Jedes Event besteht aus einem Q-Tupel, das eine mögliche Zuordnung der Fahrgäste zu einem Fahrzeug darstellt. Die Tupel entsprechen der Form

$$v = (\rho^{i+/-}_{s,r_1,d}, \psi^i_{s',s'',r_2,d}, \dots)$$

für Fahrzeuge der Linie i. Die Aktion $\rho^{i+/-}_{s,r_1,d}$ steht an erster Stelle und beschreibt die aktuelle Ein- oder Ausstiegsaktion. Danach folgt eine Reihe von maximal $Q_i - c(r_1)$ Streckenabschnitten, die alle weiteren Passagiere beinhalten, die sich zu diesem Zeitpunkt im Fahrzeug befinden. Weiter definieren wir $R(v)$ als die Menge der Transportanfragen in v, $\Psi(v)$ als die Menge der Streckenabschnitte in v und $\mathrm{P}(v)$ als Umsteigeaktion von v. Events, deren Aktion eine Einstiegsaktion ist, nennen wir Einstiegs-Events. Wiederum bezeichnen wir solche, deren Aktionen Ausstiegsaktionen sind, als Ausstiegs-Events. Für jedes Depot 0 erstellen wir außerdem ein Idle-Event, das den Start- und Endzustand des Fahrzeugs im Depot darstellt. Die Menge aller Idle-Events sei V_0 mit $V_0 \in V$. Ein Event, im Sinne des obigen Tupels, beschreibt somit eine valide Zusammensetzung aus Streckenabschnitten, die gleichzeitig von einem Fahrzeug der Linie i behandelt werden können, während eine Aktion ρ stattfindet. Als Beispiel betrachten wir das Tupel $(\rho^{i+}_{s_2,r_3,1}, \psi^i_{s_1,s_3,r_1,1}, \psi^i_{s_1,s_4,r_2,1})$, welches ausdrückt, dass das Fahrzeug auf der Linie i an der Haltestelle s_2 die Transportanfrage 3 aufnimmt, während die Anfragen 1 und 2 bereits im Fahrzeug sitzen, die in aufsteigender Richtung zur Haltestelle s_3 beziehungsweise s_4 transportiert werden. Im Event-Based Graph verbinden wir solche Event-Tupel mit gerichteten Kanten, die eine zeitliche Ordnung der Aktionen bedeuten. Wie intuitiv verständlich ist, sind nicht alle Tupel und Folgen von Tupeln möglich. Ein Nutzer kann z.B. nicht im Fahrzeug sein, ohne vorher zuzusteigen, und kann nicht verschwinden, ohne auszusteigen. Wenn an einer Haltestelle Passagiere ein- und aussteigen möchten, so werden alle Ausstiegsaktionen zuerst erledigt, bevor andere Passagiere einsteigen dürfen. Diese und weitere Nebenbedingungen (Constraints) stellen wir mithilfe des Event-Based Graph dar.

3.1. Beispiel für Event-Based Graph

Die oben beschriebene Struktur des Event-Based Graphs wollen wir nun anhand eines einfachen Beispiels veranschaulichen. Wir beziehen uns dabei auf das Netzwerk aus Abbildung 2.1 und nehmen an, dass sich genau ein Fahrzeug auf jeder Linie befindet, welches anfangs im Depot wartet. Zu Beginn sei eine einzige Transportanfrage r_1 im System, die vom Startpunkt r^+ zum Zielpunkt s_9 transportiert werden möchte. Dafür müssen die drei Linien $0, 1$ und 2 passiert werden (in der Abbildung orange, lila und grau). Der zugehörige Event-Based Graph ist in Abbildung 3.1 abgebildet.

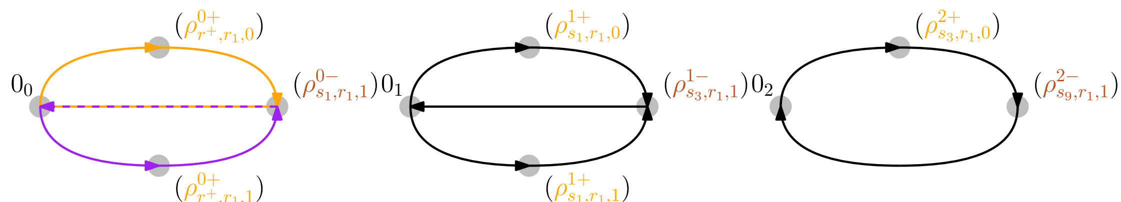


Abb. 3.1.: Event-Based Graph mit einer einzigen Anfrage. Die Einstiegsaktionen sind in einem hellen Orange und die Ausstiegsaktionen in einem dunklen Orange hervorgehoben.

Wie sofort auffällt, handelt es sich bei dem Event-Based Graph um drei unabhängige Graphen, die jeweils eine der drei benötigten Linien darstellen und anhand ihres Idle-Events unterschieden werden können. Für die Anfrage r_1 besteht auf den Linien 0 und 1 die Möglichkeit, aus zwei Fahrtrichtungen zu wählen. Dies wird im EBG durch die zwei ausgehenden Kanten des Depots dargestellt, die jeweils zu einem der beiden möglichen Einstiegs-Events führen. Beide Events führen anschließend zum gemeinsamen Ausstiegs-Event $\rho^{0-}_{s_1,r_1,1}$ beziehungsweise $\rho^{1-}_{s_3,r_1,1}$ und von dort aus wiederum zurück zum Depot (Wir erinnern uns, dass die Richtung der Ausstiegsaktionen immer 1 ist, weshalb es für zwei mögliche Streckenabschnitte zwei Einstiegsaktionen, aber nur eine Ausstiegsaktion gibt).

Auf Linie 0 beginnt ein Fahrplan beispielsweise in aufsteigender Richtung mit dem Abholen der Anfrage r_1 am Punkt r^+, um diesen anschließend an s_1 wieder abzuliefern und zum Depot 0_0 zurückzukehren. Dieser Fahrplan ist in der Abbildung lila hervorgehoben. Der alternative Fahrplan, bei dem das Fahrzeug in absteigender Richtung fährt, ist Orange hervorgehoben. Beide Fahrpläne teilen sich die Kante von $\rho^{0-}_{s_1,r_1}$ zum Depot. Für das liDARPT, wie wir es eingeführt haben, sind die Linien im Event-Based Graph nicht miteinander verbunden, da dies bedeuten würde, dass ein Fahrzeug die Linie wechseln könnte. Die Fahrpläne der Fahrzeuge werden deshalb für jede Linie unabhängig erstellt. Die spätere Koordination der Umstiege zwischen mehreren Linien wird Aufgabe des MILPs sein.

Fügen wir nun eine weitere Transportanfrage r_2 hinzu, so steigt die Komplexität des Event-Based Graphs deutlich. Die nächste Anfrage startet bei s_7 und Endet bei s_8. Sie muss folglich die Linien 0 und 1 passieren und hat Überschneidungen mit der Anfrage r_0, die wir beachten müssen. Der zugehörige Event-Based Graph ist in Abbildung 3.2 abgebildet.

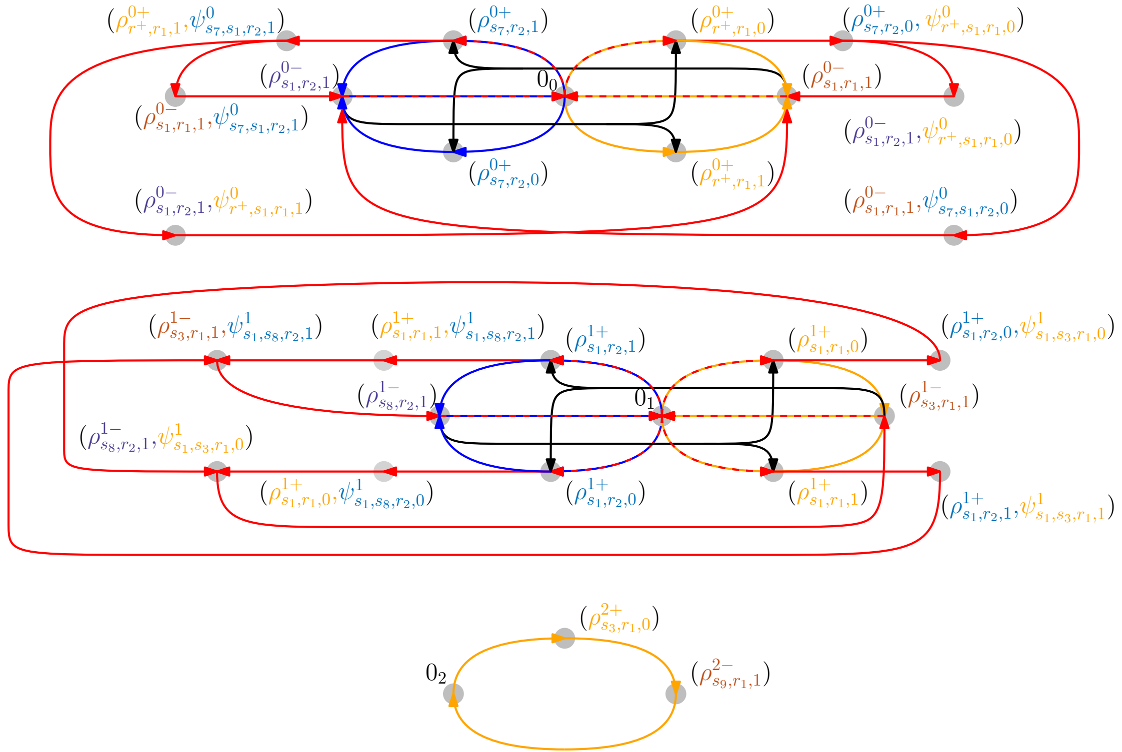


Abb. 3.2.: Event Based Graph nach Hinzufügen einer zweiten Anfrage. Die Einstiegsaktionen der zweiten Anfrage sind hellblau, die Ausstiegsaktionen dunkelblau hervorgehoben.

Wir können nun als einfachste Lösung die beiden Anfragen nacheinander transportieren, was im obigen Graphen durch die beiden Kreise, die von den Depotknoten 0_0 und 0_1 ausgehen, dargestellt wird (In der Abbildung blau und orange hervorgehoben). Sobald eine der beiden Transportanfragen erledigt ist, kann das Fahrzeug entweder zum Depot zurückkehren und seine Fahrt beenden, oder eine der schwarzen Kanten nehmen und die zweite Anfrage abholen. Alternativ können die Anfragen auch zeitgleich transportiert werden (Unter der Annahme, dass sich die Zeitfenster der Anfragen überschneiden). Solche Routen markieren wir rot.

In diesem Fall können wir frei entscheiden, welche Anfrage zuerst abgeholt wird. Sobald die erste Anfrage abgeholt ist, steht die Fahrtrichtung jedoch eindeutig fest, da die zweite Anfrage nur in einer Richtung auf dem Weg liegt. Bei der Reihenfolge des Ausstiegs haben wir freie Wahl.

Auf Linie 1 starten beide Anfragen auf demselben Knoten s_1. Wir können erneut frei wählen, welche Anfrage zuerst aufgenommen wird. Im Gegensatz zu Linie 0 können wir ebenfalls die Richtung frei wählen. In Abhängigkeit von der gewählten Richtung unterscheidet sich jedoch die Reihenfolge des Ausstiegs.

Die Optionen auf Linie 2 bleiben unverändert, da die Transportanfrage r_2 diese nicht verwendet.

Das Beispiel zeigt anschaulich

1. dass der Event-Based Graph aufgrund der Ein- und Ausstiegspunkte nicht symmetrisch ist, d.h. die Reihenfolge des Zu- und Ausstiegs sowie die Fahrtrichtung nicht immer beliebig gewählt werden können.

2. dass die einzelnen, nicht zusammenhängenden Teilgraphen jeder Linie unabhängig voneinander sind.

3. dass durch die Möglichkeit, Kreislinien in zwei Richtungen zu befahren und die Ein- und Ausstiegsreihenfolge beliebig zu wählen, die Größe und somit die Komplexität des Event-Based Graphs exponentiell ansteigt.

3.2. Konstruktion des Event-Based Graph

Wir haben bereits den Zweck des Event-Based Graphs sowie den grundlegenden Aufbau erläutert. Wir wollen nun die bereits genannten Eigenschaften präzisieren und weitere Einschränkungen formulieren, die wir als Vorverarbeitung für das MILP verwenden werden. Wie wir im vorherigen Abschnitt gesehen haben, steigt die Größe des Event-Based Graphs sehr schnell mit der Anzahl der Anfragen, weshalb der Wert einer guten Vorverarbeitung nicht genug betont werden kann.

Wir definieren nun genauer, wie wir die Knoten- und Kantenmenge des Event-Based Graphs konstruieren.

3.2.1. Knoten

Bevor wir mit den Eigenschaften der Knoten des Event-Based Graphs beginnen, führen wir eine alternative Formulierung für die Streckenabschnitte ein. Wir erinnern uns, dass jeder Streckenabschnitt $\psi^i_{s',s'',r,d}$ in zwei eindeutige Aktionen ρ^{i+}_{s',r,d_1} und ρ^{i-}_{s'',r,d_2} unterteilt werden kann. Wir verwenden deshalb auch die Schreibweise $(\rho_1, \rho_2) \in \psi$, wenn wir explizit auf die Aktionen des Streckenabschnitts zugreifen.

Wir beginnen nun mit den Events des Event-Based Graphs. Wir nehmen an, dass alle Transportanfragen $r \in R$ bereits auf Erfüllbarkeit geprüft wurden (beispielsweise mithilfe von Dijkstras Algorithmus) und eine kürzeste Route gefunden wurde, die die maximale Reisedauer $\mathcal{L}_r$ unterschreitet. Für eine gegebene Transportanfrage r kennen wir aus dem zugehörigen Routenbaum (siehe Abschnitt 2.2) bereits alle möglichen Aktionen $\rho \in \mathrm{P}(r)$ und Streckenabschnitte $\psi \in \Psi(r)$. Wir wollen nun aus der gesamten Menge aller Aktionen und Streckenabschnitte die Menge aller möglichen Events berechnen. Dafür kombinieren wir jeweils eine Aktion ρ mit maximal $Q_i - c(r(\rho))$ Streckenabschnitten ψ. Diese Kombinationen unterliegen zusätzlich den folgenden Bedingungen:

Kapazität Die maximale Kapazität Q_i des Fahrzeugs (auf der Linie i) darf nicht überschritten werden. Das bedeutet, dass für ein Event v auf Linie i die Menge der trans-

portierten Personen $\sum_{r \in r(\mathrm{P}(v)) \cup \bigcup_{\psi \in \Psi(v)} r(\psi)} c(r)$ kleiner gleich der maximalen Kapazität Q_i der Fahrzeuge auf Linie i sein muss.

Eindeutigkeit Eine Anfrage darf nur genau einmal pro Event vorkommen. Das bedeutet $r(x) \neq r(y) : \forall (x, y) \in \mathrm{P}(v) \cup \Psi(v)$ mit $x \neq y$, wobei $r(x)$ die Anfrage des Elements x beschreibt.

Linienzugehörigkeit Jedes Element x eines Events muss auf der selben Linie $i(x)$ stattfinden, da Fahrzeuge an eine einzige Linie gebunden sind. Somit gilt $i(x) = i(y) : \forall x, y \in \Psi(v) \cup \mathrm{P}(v)$.

Richtungseinschränkung Zwei Fahrgäste können nur im selben Fahrzeug sitzen, wenn sie auch in dieselbe Richtung fahren. Wir garantieren somit, dass kein Fahrgast sich von seinem Ziel entfernt, indem er in die „falsche“ Richtung fährt. Deshalb muss $\forall \psi \in \Psi(v) : d(\psi) = 1$ oder $\forall \psi \in \Psi(v) : d(\psi) = 0$ gelten. Falls v eine Einstiegsaktion ist, so verlangen wir weiter, dass $d(\rho) = d(\psi) : \forall \psi \in \Psi(v), \rho \in \mathrm{P}(v)$.

Räumliche Überlappung Damit ein Streckenabschnitt in einem Fahrzeug transportiert werden kann, während eine Aktion ρ stattfindet, muss sich der Streckenabschnitt mit der Haltestelle $s \in s(\rho)$ der Aktion überschneiden. Wäre dies nicht der Fall, so müsste aufgrund der Richtungseinschränkung der Streckenabschnitt bereits abgeliefert worden sein, bevor s erreicht wird. Wir führen dafür die Schreibweise $s_1 \xrightarrow[d]{s} s_2$ ein, welche besagt, dass Haltestelle s zwischen s_1 und s_2 liegt, wenn ein Fahrzeug in Richtung d fährt. Wir fordern also für jedes Event v, dass $s(\rho_1) \xrightarrow[d(\rho)]{s(\rho)} s(\rho_2) : (\rho_1, \rho_2) = \psi, \forall \psi \in \Psi(v), \rho \in \mathrm{P}(v)$

Zeitliche Überlappung Damit zwei Fahrgäste zusammen in einem Fahrzeug transportiert werden können, müssen sich ihre Transportzeiten überschneiden. Das bedeutet, dass für alle Events v gilt $e(\rho) + d + w(s(\rho), s(\rho_2)) \leq l(\psi)$ und
$e(\psi) + d + w(s(\rho_1), s(\rho)) \leq l(\rho) : \forall (\rho_1, \rho_2) \in \Psi(v), \rho \in \mathrm{P}(v)$.

Abfolgeeinschränkung Wenn an einer Haltestelle Nutzer sowohl ein- als auch aussteigen wollen, so wird zuerst aus- und danach eingestiegen. Wir schließen demnach Einstiegs-Events v_{enter} mit $\rho \in \mathrm{P}(v_{enter})$ aus, für die gilt
$\exists \psi = (\rho_1, \rho_2) \in \Psi(v_{enter}) : s(\rho) = s(\rho_2)$. Ebenso schließen wir Ausstiegs-Events v_{exit} mit $\rho \in \mathrm{P}(v_{exit})$ aus, für die gilt $\exists \psi = (\rho_1, \rho_2) \in \Psi(v_{exit}) : s(\rho) = s(\rho_1)$.

Zeitbeschränkung Jede Linie i im Netzwerk hat ein vorgegebenes Zeitfenster $[e(i), l(i)]$, welches die Servicezeit der Linie vorgibt. Wir wissen außerdem, dass jedes Fahrzeug der Linie zu Beginn und Ende des Zeitfensters im Depot 0_i sein muss. Jedes Event im Event-Based Graph muss demnach folgende zwei Bedingungen erfüllen:

- Bevor das Event v erreicht wird muss ein Fahrzeug alle Anfragen in v aufgesammelt haben. Wir müssen sicherstellen, dass dafür genug Zeit nach Beginn der Servicezeit ist. Wir durchlaufen dafür die Einstiegspunkte der Streckenabschnitte in $\Psi(v)$ in aufsteigender Reihenfolge und summieren die Reisezeiten zwischen den einzelnen Haltestellen auf. Für jeden Streckenabschnitt $\psi \in \Psi(v)$ überprüfen wir, ob die früheste erreichbare Zeit kleiner ist als der Beginn des Zeitfensters $e(\psi)$. Das genaue Vorgehen wird in Pseudocode 5 beschrieben.
- Nachdem das Event v erreicht wurde, muss das Fahrzeug alle Streckenabschnitte in v abliefern. Wir überprüfen deshalb, ob ein Fahrzeug genug Zeit dafür zur Verfügung hat, bevor das ende der Servicezeit $l(i)$ eintritt. Wir gehen dabei ähnlich wie in Pseudocode 5 beschrieben vor.

Algorithmus 5 Calculate time earliest execution time for event

```
i sei die Linie des Events
v sei das Event
v.action sei die Aktion von v
v.splits sei die Liste der Streckenabschnitte von v
sorted_splits = v.splits.sortList(split[0].location)
earliest_time = e(i)
last_stop = 0_i
for s in sorted_splits do
    current_stop = s[0]
    temp_time = earliest_time + b + w(last_stop,current_stop)
    earliest_time = max(temp_time,e(current_stop))
    last_stop = current_stop
return earliest_time
```

3.2.2. Kanten

Anschließend gehen wir genauer auf die Einschränkungen ein, die für die Kanten des Event-Based Graphs gelten.

Um die Kanten des Event-Based Graphs effizient zu berechnen, definieren wir zwei zusätzliche Funktionen $\Psi_{\text{in}}(v)$, $\Psi_{\text{out}}(v)$, die die möglichen Mengen der Streckenabschnitte

eines Fahrzeugs vor und nach der Ausführung eines Events v enthalten.

$$\Psi_{\text{in}}(v) = \begin{cases} \{\Psi(v)\}, \text{falls } v \text{ Einstiegs-Event} \\ \{\{\Psi(v) \cup \psi\} \mid \forall(\rho_1, \rho_2) = \psi \in \Psi, \rho \in \mathrm{P}(v) : \rho_2 = \rho\}, \text{falls } v \text{ Ausstiegs-Event} \\ \emptyset, \quad \text{ansonsten} \end{cases} \tag{3.1}$$

$$\Psi_{\text{out}}(v) = \begin{cases} \{\{\Psi(v) \cup \psi\} \mid \forall(\rho_1, \rho_2) = \psi \in \Psi, \rho \in \mathrm{P}(v) : \rho_1 = \rho\}, \text{falls } v \text{ Einstiegs-Event} \\ \{\Psi(v)\}, \quad \text{falls } v \text{ Ausstiegs-Event} \\ \emptyset, \quad \text{ansonsten} \end{cases} \tag{3.2}$$

Wir verwenden dafür die Menge aller Streckenabschnitte, die eine Aktion ρ als Start- oder Endpunkt haben.

Anschließend prüfen wir für alle Paare an Events v_1, v_2 mit $v_1, v_2 \in V$, ob $\Psi_{\text{out}(v_1)} = \Psi_{\text{in}(v_2)}$. Trifft dies zu, so ist eine Kante (v_1, v_2) zwischen den Events möglich. Die einzige Nebenbedingung, die wir dabei beachten müssen, ist die zeitliche Erfüllbarkeit: $e(v_1) + b + w(s(\mathrm{P}(v_1)), s(\mathrm{P}(v_2))) \leq l(v_2)$. Auf die genaue Berechnung der Zeitfenster gehen wir in Abschnitt 6.2.3 ein.
Für die Kantenmenge des Event-Based Graphs gelten keine weiteren Nebenbedingungen. Da wir die Fahrtrichtung der Events über die enthaltenen Streckenabschnitte kodiert haben, müssen wir beispielsweise keine weitere Richtungseinschränkung beachten. $\Psi_{\text{in}}(v_1)$ und $\Psi_{\text{out}}(v_2)$ können sich nur überschneiden, falls alle Streckenabschnitte dieselbe Richtung haben oder beide Mengen leer sind. Das Fahrzeug kann somit nur wenden, wenn keine Passagiere an Bord sind. Zur Verdeutlichung: Die Kantenmenge A besteht nun aus den folgenden Kantentypen:

1. Kanten von einer Einstiegsaktion ρ^{i+} zu einer Ausstiegsaktion ρ^{i-}
2. Kanten von einer Einstiegsaktion ρ^{i+} zu einer weiteren Einstiegsaktion ρ^{i+}
3. Kanten von einer Ausstiegsaktion ρ^{i-} zu einer Einstiegsaktion ρ^{i+}
4. Kanten von einer Ausstiegsaktion ρ^{i-} zu einer weiteren Ausstiegsaktion ρ^{i-}
5. Kanten vom Depot 0_i zu einer Einstiegsaktion ρ^{i+}
6. Kanten von einer letzten Ausstiegsaktion ρ^{i-} zum Depot 0_i
7. Kanten von einer letzten Ausstiegsaktion ρ_d^{i-} zu einer Einstiegsaktion $\rho_{d'}^{i+}$, die in eine andere Richtung fährt:

3.3. Dynamischer Event-Based Graph

Bisher haben wir die statische Variante des Event-Based Graphs, wie sie unter anderem von Barth et al. [BRS25] verwendet wurde betrachtet. Um die dynamischen

Eigenschaften unserer Problemstellung zu modellieren müssen wir jedoch zusätzlich die Veränderung des Event-Based Graphs im Verlauf der Zeit betrachten.

Um die dynamischen Eigenschaften des liDARPT zu modellieren, müssen wir zunächst mehrere Zustände der Transportanfragen unterscheiden. Dabei orientieren wir uns an den Definitionen von Gaul et al. [GKS21]:

- $\mathcal{N}(t)$ sei die Menge an neuen Transportanfragen zum Zeitpunkt t.
- $\mathcal{S}(t)$ sei die Menge der eingeplanten Transportanfragen, die also zum Zeitpunkt t angenommen, aber noch nicht abgeholt wurden.
- $\mathcal{P}(t)$ sei die Menge der abgeholten Anfragen.
- $\mathcal{D}(t)$ sei die Menge der abgelieferten Anfragen.
- $\mathcal{R}(t)$ sei die Menge der abgelehnten Anfragen.
- $\mathcal{A}(t)$ sei die gesamte Menge der aktiven Anfragen.

Wir definieren $A(t) := \mathcal{N}(t) \cup \mathcal{S}(t) \cup \mathcal{P}(t)$ mit $\mathcal{D}(t) \cup \mathcal{R}(t) \notin \mathcal{A}(t)$.

Für die Knoten und Kanten des Event-Based Graph unterscheiden wir zwischen den Zuständen Abgelehnt, Aktiv und Realisiert. Aktive Kanten können frei eingeplant werden. Realisierte Kanten waren zuvor aktiv, bis sie von einem Fahrzeug befahren wurden, was zur Folge hat, dass sie von nun an fixiert sind und in jeder Lösung enthalten sein müssen. Abgelehnte Knoten und Kanten sind solche, die zuvor aktiv waren und nun durch die Menge der realisierten Knoten und Kanten unerfüllbar geworden sind (siehe Abschnitt 2.2.2). $A_{\mathcal{R}}(t)$ sei die Menge der abgelehnten Kanten, $A_{\mathcal{A}}(t)$ die Menge der aktiven Kanten und $A^{\text{realized}}(t)$ die Menge der realisierten Kanten. Parallel dazu definieren wir die Mengen der abgelehnten, aktiven und realisierten Knoten als $V_{\mathcal{R}}(t), V_{\mathcal{A}}(t), V^{\text{realized}}(t)$.

Der bisher beschriebene Event-Based Graph spiegelt die Anfragen der Nutzer zu einem festen Zeitpunkt wider. Wir wollen jedoch das dynamische liDARPT betrachten und müssen den Graphen folglich um eine zeitliche Komponente erweitern. Wir betrachten nun den Event-Based Graph $G(t) = (V(t), A(t))$ der zum Zeitpunkt t die Knoten $V(t)$ und die Kanten $A(t)$ enthält. Wenn zwischen $t-1$ und t eine Veränderung auftritt, so wird der Graph $G(t)$ anhand des vorherigen Graphen $G(t-1)$ und der dazugehörigen Lösung $\chi(t-1)$ berechnet. Dabei werden Knoten und Kanten teilweise entfernt, die zu Transportanfragen $r \in \mathcal{R}(t) \cup \mathcal{D}(t) \cup \mathcal{P}(t)$ gehören. Anschließend werden für alle neu eingetroffenen Anfragen in $\mathcal{N}(t)$ Knoten und Kanten hinzugefügt.

Wir gehen nun genauer auf die einzelnen Teilschritte des oben beschriebenen Vorgehens ein:

Ausgangspunkt Wir beginnen mit dem Event-Based Graph aus dem vorherigen Zeitschritt $G(t-1)$ und dessen Lösung $\chi(t-1)$. Wir kennen außerdem die Mengen $\mathcal{N}(t), \mathcal{S}(t), \mathcal{P}(t), \mathcal{D}(t), \mathcal{R}(t)$ und $\mathcal{A}(t)$ für den aktuellen Zeitpunkt t.

Entfernung von Anfragen Anfragen aus $\mathcal{R}(t)$, die abgelehnt wurden, können aus dem Event-Based Graph entfernt werden. Dasselbe gilt für Anfragen, die an ihrem Ziel abgeliefert wurden und in $\mathcal{D}(t)$ enthalten sind. Wir modifizieren die Knoten und Kantenmengen folgendermaßen:

$$V_1(t) = \{v \mid v \in V(t) : \forall r_m \in (\mathcal{R}(t) \cup \mathcal{D}(t)) \wedge r_m \neq r(\mathrm{P}(v)) \wedge r_m \notin \bigcup_{\psi \in \Psi(v)} r(\psi)\}$$
$$A_1(t) = \{(v_1, v_2) \mid (v_1, v_2) \in A(t-1) : v_1 \in V_1(t) \wedge v_2 \in V_1(t)\}$$

Es werden Knoten aus dem Event-Based Graph entfernt, die Aktionen oder Streckenabschnitte enthalten, die eine Anfrage r aus $\mathcal{D}$ oder $\mathcal{R}$ transportieren. Außerdem müssen Kanten, die solche Knoten verbinden, entfernt werden.

Entfernung von unmöglichen Events Als nächstes betrachten wir die Menge $\mathcal{P}(t)$. Für diese Anfragen wurde im vorherigen Zeitschritt eine Route geplant, die bereits teilweise ausgeführt wurde. Wir können folglich Teile des Routenbaums ausschließen, wie in Abschnitt 2.2.2 beschrieben. Die Mengen der somit ausgeschlossenen Aktionen und Streckenabschnitte seien:

$$\Psi_m = \{\psi \mid \psi \in \Psi \wedge \exists \phi_j^r \in \Phi_c^r : \psi \in \phi_j^r \wedge \forall \phi_k^r \in \Phi_o^r : \psi \notin \phi_k^r\}$$
$$\mathrm{P}_m = \{\rho \mid \rho \in \mathrm{P} \wedge \rho \in \bigcup_{\psi \in \Phi_c^r} \psi \wedge \rho \notin \bigcup_{\psi \in \Phi_o^r} \psi\}$$

Die dadurch wegfallenden Knoten $V_{\mathcal{R}}(t)$ und Kanten $A_{\mathcal{R}}(t)$ entfernen wir im folgenden Schritt:

$$V_2(t) = \{v \mid v \in V_1(t) : \forall \rho \in \mathrm{P}(v) \forall \psi \in \Psi(v) : \rho \notin \mathrm{P}_m \wedge \psi \notin \Psi_m\}$$
$$A_2(t) = \{(v_1, v_2) \mid (v_1, v_2) \in A_1(t) : v_1 \in V_2(t) \wedge v_2 \in V_2(t)\}$$

Durch aktualisierte Zeitfenster im Event-Based Graph müssen außerdem verbleibende Events erneut auf Einhaltung der Zeitbeschränkung geprüft werden (siehe Zeitliche Überlappung und Zeitbeschränkung). Außerdem ersetzen wir für jedes Fahrzeug die bereits ausgeführten Kanten $a \in A^{realized}(t)$ durch eine einzige Kante, die zum zuletzt ausgeführten Knoten in $V^{1-realized}(t)$ führt. Alle anderen Knoten aus $V^{realized}(t)$ können danach aus dem Event-Based Graph entfernt werden. Wir haben somit alle bisher zurückgelegten Strecken des Fahrzeugs durch eine einzige Kante ersetzt, was bei längerer Betriebsdauer zu einer Verkleinerung des Event-Based Graphs führt. Wir erhalten somit $V_3(t)$ und $A_3(t)$.

Einfügen neuer Anfragen Abschließend fügen wir neue Transportanfragen $r \in \mathcal{N}(t)$ ein. Dafür generieren wir zunächst den Routenbaum der Anfragen und suchen alle Kombinationen aus den neuen und alten Aktionen und Streckenabschnitten. Diese fügen wir wie in Abschnitt 3.2 beschrieben in die Mengen $V_3(t)$ und $A_3(t)$ ein. Wir erhalten somit den Event-Based Graph $G(t) = (V(t), A(t))$.

Aktualisierung der Mengen Nachdem der Event-Based Graph für den Zeitpunkt t berechnet wurde, müssen die Mengen der Routen Aktionen und Streckenabschnitte Φ, P und Ψ an den neuen Event-Based Graph angeglichen werden.

4. MILP

Wir haben nun einen Event-Based Graph, der alle möglichen Fahrpläne der Busse modelliert. Wir haben diesen bereits so gestaltet, dass unmögliche und unerwünschte Zustände ausgeschlossen werden. Trotzdem haben wir bisher nur die Fahrzeugzustände und deren Abfolge betrachtet. Andere Nebenbedingungen, die beispielsweise Linien übergreifend sind, konnten wir somit nicht betrachten. Solchen Nebenbedingungen und der Suche nach optimalen Fahrplänen widmen wir das folgende Kapitel.

Für die Erstellung der *Fahrzeugpläne*, also der Abfolge der Haltestellen, die ein Fahrzeug im Verlauf des Tages anfährt, verwenden wir ein Mixed-Integer Linear Program (MILP). Dieses verwendet den Event-Based Graph zum Zeitpunkt t um für den aktuellen Zustand des Netzwerks optimale Fahrzeugpläne, unter Einhaltung weiterer Nebenbedingungen, zu finden. Unser MILP basiert auf der Formulierung des statischen liDARPT von Barth et al. [BRS25].

Bevor wir das MILP präzise formulieren können, definieren wir zunächst weitere Variablen. Jedem Knoten $v \in V(t)$ im Event-Based Graph $G(t)$ können wir eine eindeutige Position zuordnen: Die Haltestelle, an der die zugehörige Aktion durchgeführt wird. Wir können folglich jeder Kante $a \in A(t)$ die Strecke zwischen den beiden Haltestellen bestimmen und eine Transportdauer $w(a)$ zuordnen. Außerdem sei $c(a)$ die Menge der Personen, die auf der Kante a transportiert werden Für jeden Knoten $v \in V$ speichern wir die eingehende und ausgehende Kantenmenge $\delta^{in}(v) \subseteq A(t)$ und $\delta^{out}(v) \subseteq A(t)$. Die Menge der Aktionen für die Route ϕ_j^r sei $\mathrm{P}(r)_j$. Die Binärvariable y_i^r gibt an, ob die Route i der Anfrage r in der Lösung enthalten ist. Die Binärvariable q_r gibt an, ob eine Anfrage r akzeptiert oder abgelehnt wird. Parallel dazu definieren wir x_a, welches angibt, ob die Kante $a \in A(t)$ Teil der Lösung ist ($x_a = 1$) oder nicht ($x_a = 0$). Für alle Aktionen $\rho \in \mathrm{P}$ gibt $B(\rho)$ den Zeitpunkt an, zu dem die Aktion startet. Eine *Lösung* $\chi = (x, B, q, y)$ des liDARPT besteht dann aus der Menge der gewählten Kanten x, dem Zeitplan der Ausführung B, den akzeptierten Anfragen q und den verwendeten Routen y.
Wie bereits erwähnt, stellen wir die Startzeit der Aktion $\rho_{s,r,d}^{i+/-}$ mithilfe der stetigen Variable $B_{\rho_{s,r,d}^{i+/-}}$ dar. Wir verwenden eine einzige Variable, um die Startzeit für alle Events mit der Aktion $\rho_{s,r,d}^{i+/-}$ darzustellen, da jede Lösung χ höchstens ein Event für jede Aktion enthält.
Wir definieren das MILP somit folgendermaßen:

$$\min \quad \sum_{a \in A(t)} w(a) \cdot x_a \cdot (1 + c(a)) + (1 + \max_{a \in A(t)} c(a)) \cdot W \cdot \sum_{r \in \mathcal{A}(t)} (1 - q_r) \tag{4.1a}$$

s.t.

$$\sum_{a \in \delta^{in}(v)} x_a - \sum_{a' \in \delta^{out}(v)} x_{a'} = 0, \qquad \forall v \in V(t) \tag{4.1b}$$

$$\sum_{v \in V : \mathrm{P}(v) = \rho} \quad \sum_{a \in \delta^{in}(v)} x_a \geq y_j^r, \qquad \forall r \in \mathcal{A}(t), \phi_j^r \in \Phi(r), v \in V(t), \forall \rho \in \mathrm{P}(r)_j \tag{4.1c}$$

$$\sum_{a \in \delta^{out}(0_i)} x_a \leq |\mathcal{K}_i|, \qquad \forall i \in \mathcal{I} \tag{4.1d}$$

$$B_{\rho^{i+/-}_{s',r',d}} \geq B_{\rho^{i+/-}_{s,r,d}} + b + w(s, s') - M_A, \qquad \forall \rho^{i+/-}_{s',r',d}, \rho^{i+/-}_{s,r,d} \in \mathrm{P} : s \neq s' \tag{4.1e}$$

$$B_{\rho^{i'+}_{s,r',d'}} \geq B_{\rho^{i-}_{s,r,d}} - M_B, \qquad \forall \rho^{i'+}_{s,r',d'}, \rho^{i-}_{s,r,d} \in \mathrm{P} \tag{4.1f}$$

$$e(\rho^{i+/-}_{s,r,d}) \leq B_{\rho^{i+/-}_{s,r,d}} \leq l(\rho^{i+/-}_{s,r,d}), \qquad \forall \rho^{i+/-}_{s,r,d} \in \mathrm{P} \tag{4.1g}$$

$$B_{\rho^{i'-}_{r^-,r,d'}} - B_{\rho^{i+}_{r^+,r,d}} - b \leq L_r \qquad \forall r \in R, \phi_j^r \in \Phi(r) : \rho^{i'-}_{r^-,r,d'}, \rho^{i+}_{r^+,r,d} \in \phi_j^r \tag{4.1h}$$

$$B_{\rho^{i'-}_{s,r,d'}} \geq B_{\rho^{i+}_{s,r,d}} - M_2(1 - y_j^r), \qquad \forall r \in R, s \in \mathcal{S}, \phi_j^r \in \Phi(r) : \rho^{i'-}_{s,r,d'}, \rho^{i+}_{s,r,d} \in \phi_j^r \tag{4.1i}$$

$$\sum_{\phi_j^r \in \Phi(r)} y_j^r = q_r, \qquad \forall r \in R \tag{4.1j}$$

$$\sum_{a \in \delta^{out}(v)} x_a <= 2, \qquad \forall i \in \mathcal{I} \forall r \in \mathcal{A}(t) \forall v \in V : r(\mathrm{P}(v)) = r \wedge i(v) = i \tag{4.1k}$$

$$x_a \in \{0, 1\}, \qquad \forall a \in A(t) \tag{4.1l}$$

$$q_r \in \{0, 1\}, \qquad \forall r \in \mathcal{A}(t) \tag{4.1m}$$

$$y_j^r \in \{0, 1\}, \qquad \forall r \in \mathcal{A}(t), \phi_j^r \in \Phi(r) \tag{4.1n}$$

$$B_\rho \in \mathbb{R} \qquad \forall \rho \in \mathrm{P} \tag{4.1o}$$

Mit
$M_A = M_1 \cdot (1 - \sum_{(u,v) \in A(t) : u = \rho^{i+/-}_{s,r,d} \wedge v = \rho^{i'+/-}_{s',r',d}} x_{(u,v)})$
und
$M_B = M_2 \cdot (1 - \sum_{(u,v) \in A(t) : u = \rho^{i+/-}_{s,r,d} \wedge v = \rho^{i'+}_{s,r',d'}} x_{(u,v)})$

Wir gehen nun genauer auf die Bedeutung der einzelnen Zeilen ein. (4.1a) Wir optimieren primär die Anzahl der akzeptierten Anfragen. Als zweites Kriterium minimieren wir die gewichtete Fahrtzeit der Fahrzeuge. Leerfahrten haben Gewicht 1 und Nutzfahr-

ten das Gewicht $1 + c(a)$, welches der Menge der Passagiere im Fahrzeug während des Befahrens der Kante a addiert mit dem Grundgewicht 1 entspricht. Abgelehnte Anfragen werden mit einem Gewicht $W = 2w(N)|R| + 1$ bestraft, wobei $w(N)$ die Reisedauer durch das gesamte Netzwerk beschreibt (Für Beweis der Optimalität siehe Barth et al. [BRS25]). Alle Lösungen des MILPs müssen zusätzlich die folgenden Nebenbedingungen erfüllen. (4.1b) Für jeden Knoten $v \in V$ des Event-Based Graphs sei die Anzahl der aktiven eingehenden Kanten gleich der Anzahl der aktiven ausgehenden Kanten. (4.1c) Für jede akzeptierte Route müssen alle enthaltenen Streckenabschnitte verwendet werden. Deshalb muss für jede Aktion $\rho \in \mathrm{P}(r)_j$ mit $y_j^r = 1$ ein Event v mit $\mathrm{P}(v) = \rho$ besucht werden, d.h. dass es eine Kante a gibt, die zu v führt mit $x_a = 1$. (4.1d) Die Menge der aktiven ausgehenden Kanten in $\delta^{out}(0_i)$ für das Depot 0_i sei kleiner oder gleich der Anzahl an verfügbaren Bussen $|\mathcal{K}_i|$ der Linie i. (4.1e) Für aufeinanderfolgende Aktionen, die sich auf derselben Linie i, aber an unterschiedlichen Haltestellen s, s' befinden, muss genug Zeit sein, um die erste Aktion auszuführen und die Strecke (s, s') zurückzulegen. Wir wählen M_A so, dass die Anforderung nur erfüllt werden muss, falls die beiden Aktionen mit einer aktiven Kanten $a \in A$ verbunden sind. (4.1f) Ähnlich wie oben: Für aufeinanderfolgende Aktionen, die sich auf unterschiedlichen Linien aber an derselben Haltestelle befinden, muss eine eindeutige Reihenfolge gegeben sein, falls die Aktionen mit einer aktiven Kante verbunden sind. (4.1g) Für jede Aktion ρ muss das zugehörige Zeitfenster $[e(\rho), l(\rho)]$ eingehalten werden. (4.1h) Der Zeitunterschied zwischen der ersten und letzten Aktion einer Route ϕ_j^r muss kleiner sein als die maximale Reisedauer L_r. (4.1i) Für alle aktiven Routen und alle Umstiege der Routen darf ein Fahrgast nicht abgeholt werden, bevor er zuvor an der Haltestelle abgeliefert wurde. (4.1j) Für jede akzeptierte Anfrage wählen wir genau eine Route. Ist die Anfrage nicht akzeptiert, so wählen wir keine Route. (4.1k) Für jede Transportanfrage darf es maximal zwei Aktionen pro Linie geben. Eine Einstiegsaktion und eine Ausstiegsaktion oder gar keine Aktion.

5. Rolling-Horizon Algorithmus

In diesem Abschnitt präzisieren wir den Programmablauf des Rolling-Horizon Algorithmus. Bei dem Rolling-Horizon Verfahren wird der Event-Based Graph iterativ aktualisiert, um den momentanen Zustand der Fahrzeuge und Transportanfragen widerzuspiegeln. Anschließend wird das MILP gelöst, um global optimale Fahrpläne zu ermitteln. Verändert sich der Zustand, indem eine neue Transportanfrage eingeht, so wird der Event-Based Graph aktualisiert und das MILP erneut ausgeführt, um zu entscheiden, ob die Anfrage angenommen wird oder nicht. Wir können jede neu eintreffende Anfrage somit als Erweiterung des Horizonts durch neue Informationen sehen, die eine Reevaluation des vorherigen Ergebnisses veranlasst. Wir erläutern den Rolling-Horizon Algorithmus anhand von Pseudocode 6.

Algorithmus 6 Rolling-Horizon Algorithmus für dynamisches liDARPt

1: $\mathcal{T}$ sei die Menge aller Zeitschritte
2: t_1 sei der erste Zeitschritt
3: (x, B, q, y)=solve(MILP(t_1)) ▷ Verarbeitung der Anfragen vom Vortag
4:
5: **for** $t \in \mathcal{T} \setminus t_1$ **do**
6: Frage $\mathcal{N}(t)$,$\mathcal{P}(t)$,$\mathcal{D}(t)$, $\mathcal{S}(t)$ und $\mathcal{R}(t)$ ab
7: Bestimme A^{realized} und V^{realized}
8: $\mathcal{A}(t) = \mathcal{A}(t-1) \cup \mathcal{N}(t) \setminus (\mathcal{D}(t) \cup \mathcal{R}(t))$
9: Aktualisiere Routen und Zeitfenster im Routenbaum
10: Aktualisiere Event-Based Graph
11: Aktualisiere die Variablen und Nebenbedingungen des MILPs
12: $(x, B, q, y) = \text{solve}(MILP(t))$ in Zeit Δ
13: Führe den Plan bis zum nächsten Zeitschritt $t + 1$ aus

Der Algorithmus beginnt mit einer initialen Berechnung der Fahrpläne für die am Vortag angemeldeten Anfragen, die also zum Zeitpunkt t_1 bekannt sind. Anschließend wird auf eintreffende Kanten während der Ausführung der Fahrpläne gewartet. Diese Treffen zu den Zeitpunkten $\mathcal{T} \setminus t_1$ ein. Für einen solchen Zeitpunkt erfragen wir zunächst den Zustand der Fahrgäste im aktuellen System und anschließend die bereits Umgesetzten Knoten und Kanten des Event-Based Graphs. Daraufhin wird der Routenbaum, wie in Abschnitt 2.2.2 beschrieben, aktualisiert und der Event-Based Graph, wie in Abschnitt 3.3 aufgezeigt, angepasst. Nun erläutern wir genauer **Zeile 11** „Aktualisiere die Variablen und Nebenbedingungen des MILPs“.
Zunächst setzen wir die Variable q_r auf 1 für alle Anfragen r aus $\mathcal{S}(t) \cup \mathcal{P}(t)$. Wir verhindern somit, dass bereits akzeptierte Anfragen im Nachhinein abgelehnt werden. Für

die Routenoptionen ϕ ist eine solche Festlegung nicht zielführend, da wir auch Routen, die bereits teilweise ausgeführt wurden, bei Bedarf anpassen wollen. Wir fügen für jedes Fahrzeug $k \in \mathcal{K}$ eine Kante a_k in die Kantenmenge $A(t)$ ein, welche die bisher zurückgelegte Strecke ersetzt. Jede Kante a_k beginnt beim Idle-Event 0_i, für das $k \in \mathcal{K}_i$ gilt. Jede Kante a_k endet wiederum beim zuletzt ausgeführten Event v des Fahrzeugs. Die Ausführungszeit B_ρ der Aktion $\rho = \mathrm{P}(v)$ setzen wir auf die tatsächliche Ausführungszeit des Events. Hat das Fahrzeug die Aktion $\mathrm{P}(v)$ zum Zeitpunkt t bereits abgeschlossen und fährt nun auf einer Kante $a \in A(t-1)$ zum nächsten Event, so muss $x_a = 1$ fixiert werden.

In Zeile 8 haben wir bereits alle unmöglichen Routenoptionen für die Anfragen in $\mathcal{P}(t)$ entfernt. Nun müssen wir für alle verbleibenden, offenen Routenoptionen $\Phi_o^r(t)$ alle bereits abgeschlossenen Aktionen entfernen, da Gleichung (4.1c) ansonsten unerfüllbar wäre. Anschließend können wir in **Zeile 12** das MILP erneut lösen. Wir setzen dabei eine maximale Rechenzeit Δ ein. Erreicht das MILP in diesem Zeitraum keine Lösung, so lehnen wir alle Anfragen $\mathcal{N}(t)$ ab und führen den vorherigen Plan aus. Erreicht das MILP in Δ einen gültigen, aber nicht optimalen Fahrplan, so können wir mit diesem fortfahren. Anschließend wird der berechnete Plan bis zum nächsten Zeitschritt ausgeführt.

6. Implementierung

Zusätzlich zur theoretischen Ausarbeitung des dynamischen liDARPT fertigen wir eine Python-Implementierung unseres Algorithmus an, welche auf der Arbeit von Barth et al. [BRS25] aufbaut[1]. Dabei ist zu erwähnen, dass das bestehende Framework von Barth et al. vor allem den strukturellen Grundrahmen für unsere Implementierung darstellt. Um die dynamischen Aspekte des Problems und die unterschiedlich umgesetzten Anfragen, Linien und Vorverarbeitungsschritte zu implementieren, mussten viele Klassen stark angepasst oder komplett neu geschrieben werden.
Für die Erstellung der Implementierung wurde das KI-Tool Github Copilot zur Hilfe genommen.

Im folgenden Kapitel gehen wir auf die Codestruktur und ausgewählte Details dieser Implementierung ein. Dabei ist anzumerken, dass die zuvor beschriebene Einkürzung der ausgeführten Aktionen im Event-Based Graph aufgrund von zeitlichen Beschränkungen ausgelassen werden musste (siehe hierzu Abschnitt 6.2.7).

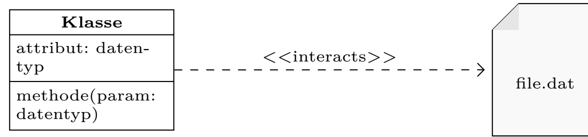
6.1. Codestruktur

Nachfolgend gehen wir genauer auf die Struktur des Codes ein und stellen die wichtigsten Klassen und Methoden vor. Wir erläutern jeden Paragraphen anhand eines zugehörigen UML-Diagramms. Eine Legende für die UML-Diagramme ist in UML-Diagramm 6.1 zu sehen. Begonnen wird mit den Klassen, die zur Modellierung des Problems verwendet werden.

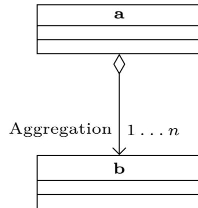
Netzwerk Im UML-Diagramm 6.2 sind die Klassen *Bus*, *Line* und *Stop* abgebildet, welche das Liniennetz modellieren. Jedes Fahrzeug der Klasse *Bus* wird über die Variable *line* einer eindeutigen Linie zugeordnet.

Eine Linie der Klasse *Line* wird über eine Liste von *Stops*, die die Haltestellen modellieren, definiert. Weiterhin verfügt jede Linie über ein Depot, an dem jede Route startet und endet, sowie über eine Kapazität an Fahrgästen, die für alle Fahrzeuge auf der Linie gilt. Die Binärvariable *circular* gibt an, ob es sich um eine Kreisroute handelt oder nicht. Das Zeitfenster, bestehend aus *startTime* und *endTime* gibt die Servicezeit der Linie an. Jede Linie verfügt außerdem über die Methoden *getTravelTime* und *getTravelDistance*, womit die Reisezeit und Entfernung zwischen zwei Haltestellen der Linie abgefragt werden können. Eine Haltestelle wird über ihren Index definiert.

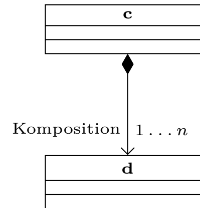
[1] https://gitlab.informatik.uni-wuerzburg.de/s411537/dynamic-lidarpt



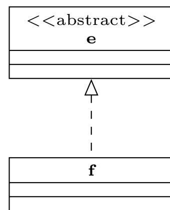
(a) Eine Klasse, bestehend aus Attributen und Methoden interagiert mit eine Datei „file.dat".



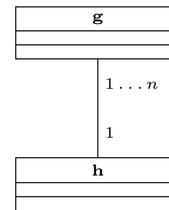
(b) Aggregation: a besteht aus einem oder mehreren b



(c) Komposition: c besteht aus einem oder mehreren d. c kann nicht ohne d existieren.



(d) f implementiert die abstrakte Klasse e



(e) Assoziation: Bidirektionale Beziehung zwischen g und h.

Abb. 6.1.: Legende der UML-Diagramme

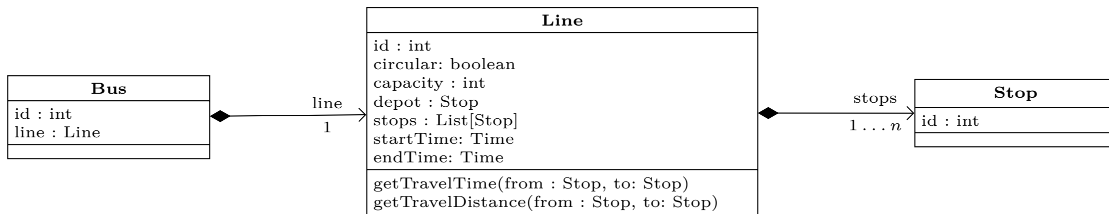


Abb. 6.2.: UML-Diagramm der Klassen, die das Netzwerk modellieren

Anfragen Transportanfragen werden über die Klasse *Request* modelliert. *Request* erweitert die abstrakte Klasse *AbstractRequest*, welche aus einer Anzahl an transportierten Passagieren, einem Start- und Endpunkt sowie jeweils einem Zeitfenster für Start und Ende der Anfrage und dem tatsächlichen Startzeitpunkt besteht. Eine Anfrage erweitert die *AbstractRequest*-Klasse um einen Routenbaum, der alle möglichen Routen zwischen dem Start- und Zielpunkt berechnet. Außerdem enthält die *Request*-Klasse ein Dictionary, welches für alle Routen des Routenbaums die ID der Route auf die Liste der zugehörigen Streckenabschnitte projiziert. Diese nutzen wir, um später im MILP zeiteffizient auf die Routen der Anfrage zuzugreifen.

Alle Streckenabschnitte einer Route werden durch die *SplitRequest*-Klasse modelliert. Diese erbt ebenfalls von der *AbstractRequest*-Klasse, da ein Streckenabschnitt, genau wie eine Anfrage, den Transport einer festen Anzahl an Personen zwischen zwei Haltestellen darstellt. Im Gegensatz zu einer Anfrage verfügt ein Streckenabschnitt über eine eindeutige Linie und Richtung, in der er verläuft. Außerdem können wir jedem Streckenabschnitt die dazugehörige Transportanfrage zuordnen.

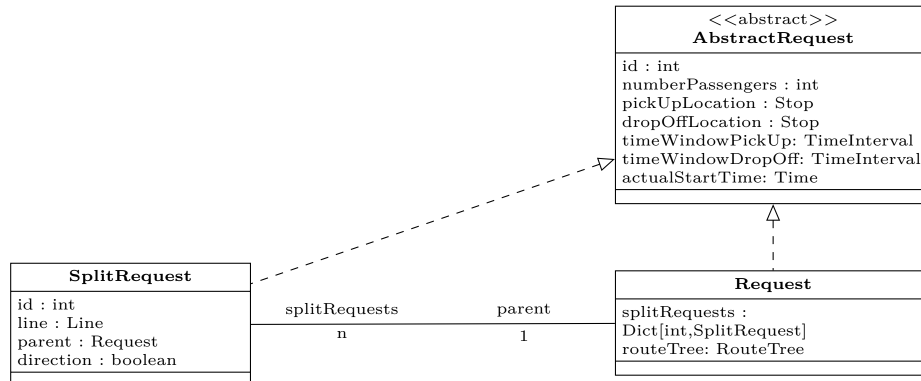


Abb. 6.3.: UML-Diagramm der Klassen, welche Anfragen modellieren

Routenbaum Der Routenbaum wird durch die *RouteTree*-Klasse modelliert. Der *RouteTree* erhält eine Referenz auf die Transportanfrage, zu der er gehört, sowie einen *LineGraph*, welcher als Basis für den Routenbaum verwendet wird und von Barth et al. übernommen wurde. Der *LineGraph* ist eine verkürzte Version des Busnetzwerks, bei dem alle Knoten, die keine Umsteigeknoten oder Start- beziehungsweise Endknoten der Anfrage sind, entfernt wurden. Alle Kanten der entfernten Knoten werden kontrahiert, sodass die Erreichbarkeit der verbleibenden Knoten erhalten bleibt. Aus diesem *LineGraph* erstellen wir nun mit *makeGraph* den Routenbaum, wie in Abschnitt 2.2 beschrieben. Dabei wird die *root*-Variable mit dem Wurzelknoten belegt. Um zukünftig, wie in Abschnitt 2.2.2 beschrieben, den Wurzelknoten des Routenbaums während der Ausführung der Anfrage zu verschieben, müssen wir die Zeit speichern, die wir vom Startpunkt der Anfrage bis zur Wurzel des Baums benötigt haben.

Alle Knoten im Routenbaum sind vom Datentyp *RouteStop*. Darin speichern wir alle vorhergehenden Streckenabschnitte, um an den Blattknoten die vollständige Route zu erhalten. Außerdem speichern wir in den Dictionarys *childNoteTimes* und *childEdgeTimes* die Zeitfenster für alle nachfolgenden Knoten und Kanten vom aktuellen Knoten aus.

RouteSplits repräsentieren die Streckenabschnitte im Routenbaum. Sie enthalten eine Referenz auf den zugehörigen Streckenabschnitt und den vorherigen und nachfolgenden *RouteStop*.

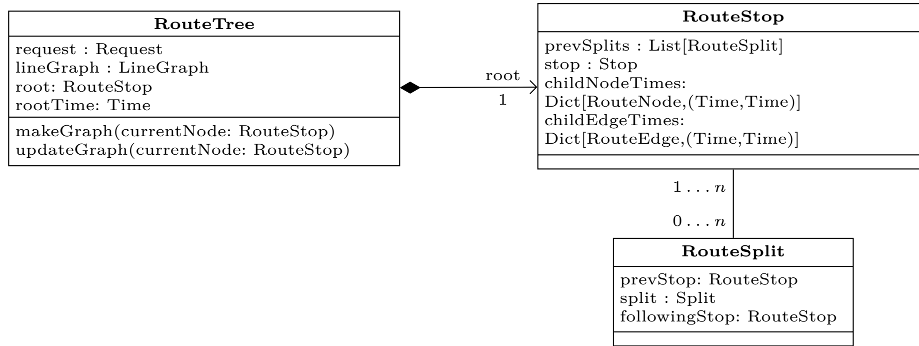


Abb. 6.4.: UML-Diagramm der Klassen, welche die Transportanfragen und Streckenabschnitte modellieren

Aktion Die *Action*-Klasse speichert, genau wie in der Definition aus Kapitel 3, eine Linie, eine Haltestelle, eine zugehörige Anfrage und eine Fahrtrichtung. Die Art der Aktion unterscheiden wir über die *type*-Variable. Ist der Wert „enter“, so handelt es sich um eine Einstiegsaktion. Ist der Wert „exit“ so handelt es sich um eine Ausstiegsaktion.

Die abstrakte *Action*-Klasse wird durch die Klassen *EnterAction* und *ExitAction* implementiert.

Events Wir modellieren den Event-Based Graph aus Abschnitt 3.3 mithilfe der *EventGraph*-Klasse. Der Graph wird über eine Liste von Knoten und einem Dictionary, das die ein- und ausgehenden Kanten der Knoten speichert, definiert.

Events werden über die zugehörige Aktion und die Menge der Streckenabschnitte definiert, die gleichzeitig transportiert werden. Weiter speichern wir die Menge der Streckenabschnitte, die mit der Aktion *first* kompatibel sind, in *inSplits* und *outSplits*. Das Event hat außerdem ein Zeitfenster, welches mithilfe der Streckenabschnitte berechnet wird. Die Methoden *setBeforeEvent* und *setAfterEvent* geben die Mengen an Split-Mengen zurück, die vor und nach dem Ausführen des Events existieren können.

Die abstrakte *Event*-Klasse wird von drei Klassen implementiert. (1) Dem *IdleEvent*, welches das verweilen am Depot darstellt. (2) Dem PickUpEvent, welches ein Event darstellt, bei dem die Aktion eine Einstiegsaktion ist. (3) Dem DropOffEvent, welches ein Event darstellt, bei dem die Aktion eine Ausstiegsaktion ist.

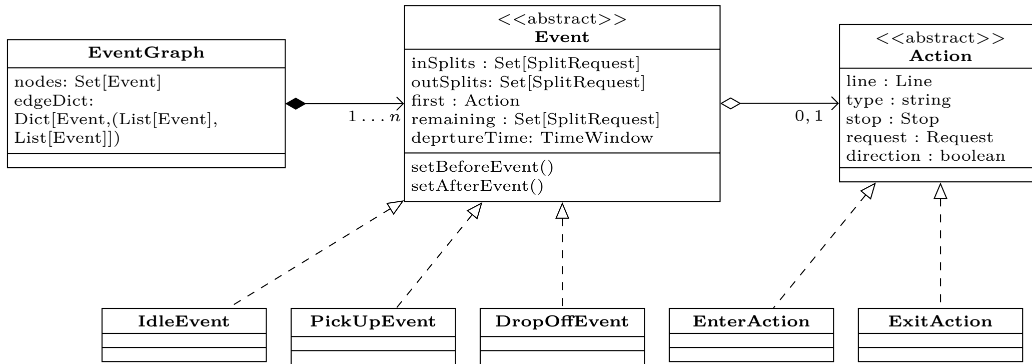


Abb. 6.5.: UML-Diagramm der Klassen, welche den Routenbaum modellieren

Fahrpläne Die Fahrpläne der Fahrzeuge modellieren wir mit der Klasse *Plan*. Jeder Fahrplan wird genau einem Fahrzeug zugeordnet, welches in der Variable *bus* gespeichert wird. Außerdem besteht ein Fahrplan aus einer geordneten Liste von Stops der Klasse *PlanStop*.

Die Klasse *PlanStop* wird mit einer Haltestelle *stop* assoziiert. Außerdem wird die genaue Ankunfts- und Abfahrtszeit des Fahrzeugs gespeichert. Zudem speichern wir, welche Anfragen an der gegebenen Haltestelle in das Fahrzeug ein- oder aussteigen.

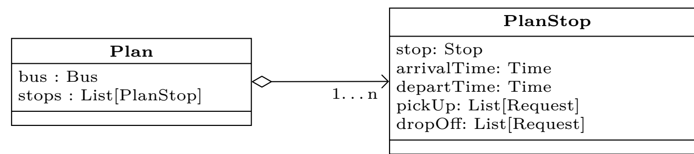


Abb. 6.6.: UML-Diagramm der Klassen, welche Fahrpläne modellieren

Funktionale Klassen Wir fahren fort mit den funktionalen Klassen, die die Programmlogik enthalten und im UML-Diagramm 6.7 abgebildet sind.

Die abstrakte Klasse *Context* steuert den Programmablauf. Sie besitzt eine Methode *createTimeTable*, die eine Menge an Transportanfragen übergeben bekommt und diese in das Dictionary *timeTable* einfügt. Die Methode *startContext* startet den Algorithmus und führt für alle Zeitpunkte in *timeTable* die Methode *triggerEvent* aus.

Dynamic implementiert die *Context*-Klasse und stellt den dynamischen Programmablauf dar. Die Methode *createTimeTable* wird so implementiert, dass *timeTable* für jede Startzeit alle Anfragen speichert, die diese Startzeit besitzen. Die Methode *startContext* kann damit die Transportanfragen nach aufsteigender Startzeit durchlaufen. Für jede Startzeit wird *triggerEvent* ausgeführt. Dies entspricht dem Auftreten der Anfragen im Bussystem. Daraufhin wird zunächst die Methode *makePlan* der *Planner*-Klasse aufgerufen, um einen neuen Plan für alle bekannten Anfragen zu erstellen. Anschließend wird mit der *executePlan*-Methode der *Executor*-Klasse eine Simulation bis zum Zeitpunkt *next* durchgeführt, wobei *next* die Startzeit der nächsten Transportanfragen darstellt.

Die abstrakte Klasse *Planner* verfügt, neben der bereits erwähnte Methode *makePlan*, über eine Liste der Fahrzeuge im Netzwerk und eine vereinfachte Darstellung des Netzwerks in Form der *LineGraph*-Klasse.

Die Klasse *EventBasedMILP* implementiert die abstrakte *Planner*-Klasse und setzt die Routenplanung mithilfe des Event-Based Graphs und MILPs um. Die Methode *makePlan* entfernt zunächst Abgelehnte Anfragen mithilfe von *removeRequests*. Anschließend werden die Anfragen, die noch transportiert werden oder abgeliefert wurden, mithilfe von *updateRequests* aktualisiert. Als letztes werden die neuen Anfragen eingefügt. Dafür verwenden wir die Methode *getCombinations*, um alle Events für die Aktionen der neuen Anfragen zu finden (Wie in Abschnitt 3.2 beschrieben). Damit aktualisieren wir den Event-Based Graph und rufen anschließend die Methoden *buildModel*, *solveModel* und *convertToPlan* des *cplexModel*-Objekts auf.

Die Klasse *CplexModel* implementiert das in Kapitel 4 beschriebene MILP. Die Methode *buildModel* erstellt alle Variablen und Nebenbedingungen aus dem Event-Based Graph und speichert sie in *currentModel*. Falls möglich, wird anschließend ein Warmstart mit den Variablenwerten aus dem vorherigen Model *prevModel* durchgeführt. Anschließend werden mithilfe der Methode *fixVars* die Variablen so gesetzt, dass die bereits ausgeführten Routen eingehalten werden (siehe Kapitel 5). Mit der Methode *solveModel* wird CPLEX gestartet und das MILP gelöst. Anschließend nutzen wir die Methode *convertToPlan*, um das Ergebnis des MILPs in Fahrpläne für die Fahrzeuge umzuwandeln.

Die Klasse *Executor* bietet die Methode *checkPlan*, welche die Gültigkeit der übergebenen Liste an Fahrplänen überprüft. Sind die übergebenen Fahrpläne gültig, so wird deren Ausführung mit der Methode *executePlan* bis zum gegebenen Zeitpunkt *next* simuliert.

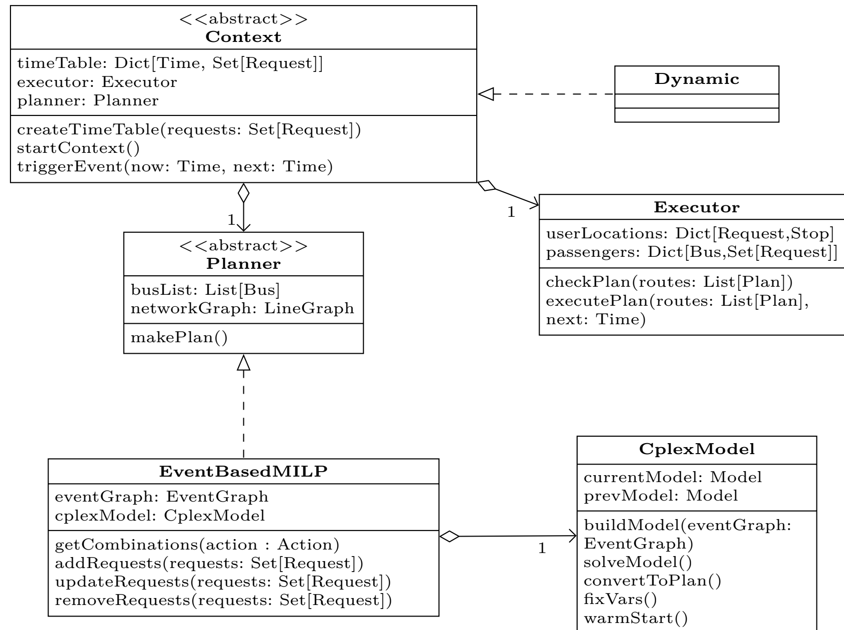


Abb. 6.7.: UML Diagram der funktionalen Klassen

IO Klassen Eine Übersicht über die IO-Module des Projekts ist in UML-Diagramm 6.8 zu sehen. Als erste Komponente des Inputs betrachten wir die Datei *network.csv*, die das Netzwerk, bestehend aus Haltestellen, Linien und Fahrzeugen, enthält. Diese Datei wird von der *RequestGenerator*-Klasse gelesen, welche für das gegebene Netzwerk zufällige Anfragen erzeugt. Dabei kann die Anzahl der Anfragen und das Zeitfenster, über das sich die Anfragen erstrecken, angegeben werden. Da wir in Kapitel 7 sowohl statische als auch dynamische Experimente durchführen wollen, gibt es zudem eine Binärvariable *static*, die angibt, ob die Anfragen statisch oder dynamisch sein sollen. Aus diesen Informationen generiert die Klasse eine Datei *requests.csv*, die alle Anfragen enthält.

Die Datei *config.json* enthält die Parameter unseres Algorithmus. Dazu gehören unter anderem ein Pfad zu *network.csv* und *requests.csv* sowie die maximale Rechenzeit des MILPs und die Umstiegszeit der Passagiere.

Die *config.json*-Datei wird anschließend von der Klasse *IOHandler* gelesen. Diese erzeugt mithilfe der Funktion *readBusNetwork* die *Bus*-, *Line*- und *Stop*-Klassen aus der Datei *network.csv*. Anschließend werden die Transportanfragen aus der *requests.csv*-Datei mithilfe der Funktion *readRequests* eingelesen und in *Request*-Objekte umgewandelt. Im Anschluss werden Objekte der Klassen *Planner* und *Executor* erstellt und an ein *Context*-Objekt übergeben, welches daraufhin über *startContext* gestartet wird. Abschließend wird mithilfe der *createOutput*-Methode das Ergebnis der Berechnung in die Datei *output.csv* geschrieben.

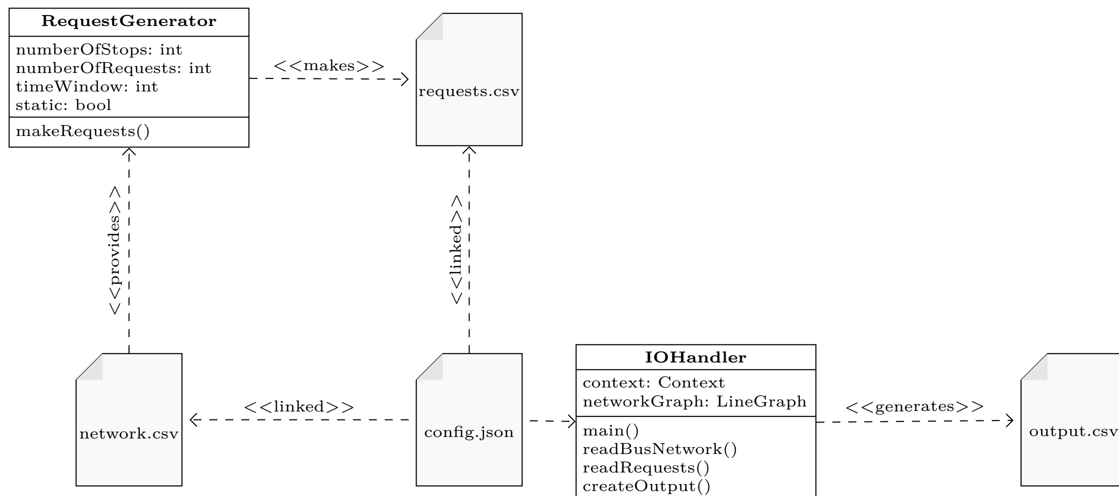


Abb. 6.8.: UML-Diagramm der IO-Module

6.2. Ausgewählte Details der Implementierung

Im folgenden Kapitel beleuchten wir besonders interessante Aspekte der Implementierung, die auch für zukünftige Arbeiten von Bedeutung sein könnten.

6.2.1. Openrouteservice

Um genauere Simulationen von Rufbussen in ihrem tatsächlichen Einsatzgebiet und folglich eine bessere Auswertung des entwickelten Algorithmus zu ermöglichen, benötigen wir möglichst genaue Informationen über die Reisezeiten der Fahrzeuge. Dafür verwenden wir die Openrouteservice-API [2]. Dabei handelt es sich um ein Open-Source-Projekt, das die Karten von OpenStreetMap verwendet und eine Vielzahl an Routing- und Optimierungsdiensten anbietet. Openrouteservice kann wahlweise auf dem lokalen System installiert oder über die öffentliche API genutzt werden. Wir haben uns zur Erleichterung des Deployments für letzteres entschieden. Kostenlose API-Schlüssel sind verfügbar und umfassen eine großzügige Anzahl an täglichen Anfragen. Der verwendete API-Schlüssel wird in der config.json Datei hinterlegt. In unserem Code verwenden wir die Time-Distance-Matrix-Funktion. Diese berechnet für eine gegebene Koordinatenmenge die Zeit, die zwischen allen Paaren von Koordinaten benötigt wird, und gibt diese in tabellarischer Form zurück. Dabei kann das Fortbewegungsmittel (zu Fuß, Fahrrad, Auto oder LKW) gewählt werden. Da wir für unseren Anwendungsfall von Kleinbussen mit einem Gewicht von weniger als 3.5 Tonnen ausgehen, berechnen wir die Reisezeiten mit dem Parameter „driving-car“. Diese Matrix ermitteln wir beim Start des Programms einmalig für jede Linie und speichern die Ergebnisse als Look-up-Tabelle. Über die Funktion *getTravelTime* in der Klasse *Line* kann anschließend auf die Werte der Matrix zu-

[2]openrouteservice.org

gegriffen werden [3]. Gleichzeitig fragen wir mit Openrouteservice die Distanzen der kürzesten Routen ab und speichern diese ebenfalls in der *Line*-Klasse. Über die Funktion *getTravelDistance* kann anschließend darauf zugegriffen werden. Wie bereits erwähnt, wird angenommen, dass innerhalb einer Linie alle Fahrzeuge bezüglich der Kapazität identisch sind. Es wäre jedoch möglich, dass auf einer Linie ein Kleinbus und auf einer anderen ein größerer Linienbus fährt. Falls sich im realen Einsatz zeigen sollte, dass diese Fahrzeuge unterschiedliche Reisezeiten haben, so könnten die entsprechenden Tabellen angeglichen werden.

6.2.2. Zyklische Routen

Wir haben uns in der Planungsphase dieser Arbeit entschieden, zyklische Routen zu erlauben. Wie sich bei der Implementierung herausstellte, sorgt dies für eine deutlich erhöhte Komplexität bei einigen Komponenten des Systems. Beispielhaft stellen wir hier die Funktion *checkOnRoute* vor. Diese Boolean-Funktion gibt für eine Haltestelle s und einen Streckenabschnitt $\psi^i_{s_1,s_2,r,d}$ an, ob $s_1 \underset{d}{\overset{s}{\rightarrow}} s_2$ gilt, d.h. ob s zwischen dem Start- und Endknoten des Abschnitts liegt. Um diese einfache Frage zu beantworten, muss zunächst geprüft werden, ob es sich um eine Kreislinie handelt. Falls dies der Fall ist, muss weiter überprüft werden, in welche Richtung das Fahrzeug auf dem Streckenabschnitt fährt. Erst anschließend können die Indizes der Haltestellen auf der Linie verglichen werden, wobei weitere Fallunterscheidungen nötig sind, da das Fahrzeug das vermeintliche Ende des Linienfahrplans überschreiten kann, falls die Start- und Endpunkte der zyklischen Linie zwischen den Haltestellen des Streckenabschnitts liegen. Der gekürzte Pseudocode zu dieser Funktion ist in Pseudocode 7 zu sehen. Wie anhand dieses Beispiels deutlich wird, kann das Einbeziehen von zyklischen Routen die Komplexität vieler Funktionen erheblich erhöhen. Dieses Problem könnte durch die Entwicklung eines geeigneteren Datentyps behoben werden, worauf wir in dieser Arbeit jedoch nicht weiter eingehen.

6.2.3. Zeitfenster

Wie bereits in Abschnitt 6.2.1 angesprochen, ist eine akkurate Reisezeit ein wichtiger Aspekt für die Genauigkeit des Algorithmus. In diesem Fall bedeutet dies vor allem, dass genaue Zeitfenster für Aktionen, Streckenabschnitte und Events benötigt werden. Die Zeitfenster müssen stets groß genug sein, um keine möglichen Routen auszuschließen und gleichzeitig klein genug, um möglichst wenige unmögliche Events zu erstellen, welche zu einer unnötig hohen Rechenlast für den CPLEX-Solver führen. Wir gehen dabei in zwei Schritten vor:

1. Für die Zeitfenster der Streckenabschnitte und Aktionen wählen wir möglichst große Zeitfenster. Die Berechnung der Reisedauer eines Streckenabschnitts basiert

[3] In der config-Datei unserer Implementierung kann der Parameter „computeNewMatrix“ auf 0 gesetzt werden, sodass eine neue API-Anfrage nur getätigt wird, falls kein lokaler Datenstand vorhanden ist um API-Tokens zu sparen.

Algorithmus 7 checkOnRoute

```
procedure CHECKONROUTE(Stop s, Split psi)
    direction = psi.direction
    start = psi.pickUpLocation
    end = psi.dropOffLocation
    if psi.line.circular then
        if direction == forward then
            if start<end then
                if start<=s<=end then
                    return True
                else
                    return False
            else
                if s<=end or s>=start then
                    return True
                else
                    return False
        else
            ...
    else
        if direction == forward and start<=s<=end then
            return True
        else if direction == backward and start>=s>=end then
            return True
        else
            return False
```

im besten Fall auf einer direkten Verbindung ohne Zwischenstopps und im schlechtesten Fall auf einem Fahrzeug, das an jeder Haltestelle auf dem Weg halten muss.

2. Im zweiten Schritt wird aus der Menge der Aktionen und Streckenabschnitte die Menge der möglichen Events berechnet. Da wir zuvor sehr große Zeitfenster berechnet haben, erhalten wir eine größere Menge an Events, die wir anschließend weiter eingrenzen, um nur relevante Events in den Event-Based Graph zu übernehmen. Wir betrachten deshalb für jedes Event v genauer die Route, die durch die Menge der Streckenabschnitte $\Psi(v)$ und die Aktion $\rho = \mathrm{P}(v)$ impliziert wird. Dafür ordnen wir alle Haltestellen im Event nach ihrer Reihenfolge. Anschließend berechnen wir die früheste Zeit $t_\rho^{\min}$, zu der wir die Aktion erreichen können, indem wir mit der frühesten Startzeit der ersten Haltestelle beginnen und von dort aus die Fahrtzeiten der kürzesten Strecken zwischen allen Haltestellen aufaddieren, bis $S(\rho)$ erreicht ist. Anschließend berechnen wir die späteste Zeit $t_\rho^{\max}$ zu der wir die Aktion verlassen können, indem wir mit der spätesten Ankunftszeit der letzten Haltestelle beginnen und von dort aus die Fahrtzeiten der kürzesten Strecken

zwischen allen Haltestellen aufaddieren, bis $S(\rho)$ erreicht ist. Gilt $t_\rho^{\min} \leq l(\rho)$ und $t_\rho^{\max} \geq e(\rho)$, so ist das Event möglich. Das Zeitfenster des Events setzen wir dann wie folgt:

$$e(v) = \max(e(\rho), t_\rho^{\min})$$
$$l(v) = \min(l(\rho), t_\rho^{\max})$$

Diese Zeitfenster nutzen wir bei der Kantenbildung im Event-Based Graph (siehe Abschnitt 3.2.2).

6.2.4. DOcplex

In der zugrundeliegenden Implementierung von Barth et al. [BRS25] verwenden die Autoren zum Konstruieren und Lösen des MILPs die Cplex Python API[4], ein schmaler wrapper für die Cplex C API. Diese API bietet fast den gesamten Funktionsumfang des IBM ILOG CPLEX Optimization Studios und verwendet eine Matrixdarstellung für die Variablen. Zum Zweck der Lesbarkeit und Wartbarkeit des Codes haben wir uns entschieden stattdessen die DOcplex-Bibiliothek[5] zu verwenden. Docplex ist anders als die Cplex-API objektorientiert und erlaubt kürzere und einfachere Formulierungen für MILP-Modelle.
Außerdem vereinfacht DOcplex die Nutzung eines Warmstarts. Dieser ermöglicht es, die Lösung des vorherigen MILPs als Startpunkt für das aktuelle MILP zu nutzen, was eine deutlich verkürzte Laufzeit zur Folge hat. Außerdem ändern sich Parameter, die weiterhin teil der optimalen Lösung sind nicht, was die Häufigkeit von kleineren Planänderungen verringern kann. Auf diesen Aspekt gehen wir im Kapitel Auswertung (siehe Kapitel 7) genauer ein.

6.2.5. Eventerstellung

Für die Erstellung der Events generieren wir, wie in Kapitel 3 beschrieben, zunächst eine Menge aller zulässigen Events und prüfen anschließend, ob diese die von uns gestellten Beschränkungen erfüllen. In unserer Implementierung setzen wir die in Abschnitt 3.2 beschriebenen Bedingungen für den Aufbau von Knoten im Event-Based Graph folgendermaßen um:

- Für eine gegebene Aktion bestimmen wir zunächst alle kompatiblen Streckenabschnitte, die die Beschränkungen der Linienzugehörigkeit, Richtungseinschränkung, Räumliche Überlappung und Zeitliche Überlappung erfüllen.
- Anschließend berechnen wir alle Kombinationen der kompatiblen Streckenabschnitte und bilden daraus die Menge aller zulässigen Events.
- Für alle zulässigen Events prüfen wir Anschließend die Kapazität, Eindeutigkeit, Abfolgeeinschränkung und Zeitbeschränkung. Besonders die Zeitbeschränkung nimmt

[4]https://www.ibm.com/docs/en/icos/22.1.2?topic=cparm-overview
[5]https://github.com/IBMDecisionOptimization/docplex-doc

dabei größere Rechenleistung in Anspruch. Deshalb überprüfen wir diese Beschränkung als letztes, nachdem bereits ein größtmöglicher Teil der Events ausgeschlossen wurde.

6.2.6. Kantenentfernung

Zuvor haben wir beschrieben, wie bereits zurückgelegte Streckenabschnitte jedes Fahrzeugs durch eine einzige Kante zum aktuellen Zustand ersetzt werden können. Auf diese Funktion haben wir in unserer Implementierung jedoch aus zeitlichen Gründen verzichtet. Der implementierte Routenbaum unterstützt jedoch alle dafür nötigen Funktionen. Durch das fixieren der x- und B-Variablen der bereits zurückgelegten Streckenabschnitte im MILP sollte der dadurch verlorene Geschwindigkeitsvorteil des MILPs minimal sein.

6.2.7. Testing

Bei der Umsetzung der Algorithmen lag ein besonderer Fokus auf der Korrektheit aller implementierten Klassen und Funktionen. Aus diesem Grund implementieren wir für jede größere Komponente Unit-Testklassen, die das erwartete Verhalten der Klassen und Funktionen überprüfen. Um die einzelnen Codesegmente unabhängig voneinander zu testen, ersetzen wir alle verwendeten Objekte durch minimale Platzhalter, wofür wir die Mock-Klasse aus dem Unittest-Paket verwenden. Globale Variablen und Funktionen ersetzen wir ebenfalls durch konstante Rückgabewerte mithilfe der patch-Funktion. Somit wird die getestete Klasse von allen übrigen Komponenten des Systems entkoppelt. Dieses Vorgehen hat jedoch auch den unerwünschten Nebeneffekt, dass unsere Testklassen zum Teil Umfangreiche Setup-Funktionen benötigen und unübersichtlich zu lesen sind.
Zudem wird auf Penetration-Tests verzichtet, die eine unsachgemäße Verwendung der Klassen überprüfen. Da das implementierte Programm ausschließlich zu Forschungszwecken verwendet wird, ist kein Missbrauch durch Nutzer zu befürchten. Der implementierte Code ist folglich nicht auf Robustheit optimiert.

7. Auswertung

In diesem Abschnitt evaluieren wir die Effektivität des vorgestellten Algorithmus für das dynamische liDARPT. Wir betrachten dabei vor allem die folgenden Aspekte.

Zuerst evaluieren wir in Abschnitt 7.2 die Effektivität unserer Vorverarbeitung (siehe Abschnitt 6.2.5). Dafür untersuchen wir zunächst, welche Eigenschaften des liDARPT einen Einfluss auf die Laufzeit des MILPs haben und untersuchen anschließend die Effektivität der Vorverarbeitungsschritte.

Nachfolgend evaluieren wir in Abschnitt 7.3.1 die Effektivität der dynamischen Ausführung statischer Anfragen. Zu diesem Zweck wandeln wir mithilfe einer einfachen Heuristik statische Testinstanzen in dynamische um und vergleichen anschließend die Ergebnisse des MILPs. Anhand dieses Experiments lassen sich zudem allgemeine Aussagen über unseren dynamischen Algorithmus im Vergleich zu einem statischen Algorithmus treffen.

Anschließend untersuchen wir den dynamischen Algorithmus auf Anwendbarkeit in realen, weder rein statischen noch dynamischen Anwendungen. Wir nehmen dafür an, dass ein Teil der Transportanfragen im Vorhinein bekannt sei und ein anderer Teil dynamisch während der Ausführung dazukommt. Dies entspricht im wesentlichen dem realen Einsatzgebiet eines dynamischen Rufbussystems.
In Abschnitt 7.3.2 vergleichen wir die eben beschriebene, hybride Anwendung des dynamischen Algorithmus mit einem rein dynamischen Datensatz.

Abschließend gehen wir in Abschnitt 7.4 auf weitere Erkenntnisse aus unseren Experimenten ein.

7.1. Testinstanzen

Für die Auswertung des Algorithmus nehmen wir die Netzwerke (1) *Markt-Karl-Lohr*, (2) *SW-Geo_full* und (3) *SW-Schlee_3* von Barth et al. [BRS25] zur Hand. Dabei handelt es sich um Fernverkehrstrecken zwischen den Orten (1) Marktheidenfeld, Karlstadt und Lohr, (2) Schweinfurt, Gerolzhofen und Volkach und (3) Schweinfurt und Schleerieth. Darstellungen der Netzwerke sind in Anhang B beigelegt. Da wir, anders als Barth et al. [BRS25], die echten GPS-Standorte der Haltestellen verwenden, um realistischere Reisezeiten zu erhalten, mussten die vorhandenen Datensätze angepasst werden. Die Gesamtlänge unserer angepassten Netzwerke ist geringfügig kleiner als zuvor, wobei *SW-Schlee_3* mit 28.3 km die größte Abweichung aufweist.

Name	Linien	Haltestellen	Umsteigepunkte	Grad der Umsteigepunkte	Länge des Netzwerks in km
Markt-Karl-Lohr	5	26	5	14	157.9
SW-Geo_full	4	31	4	13	110.5
SW-Schlee_3	3	21	3	11	64.5
Wue	3	35	9	39	19.4
Wue-simplified	3	29	3	11	17.4

Abb. 7.1.: Übersicht über alle Netzwerke

Außerdem erstellen wir eigene Testinstanzen, die ein urbanes Netzwerk *Wue* modellieren. Für unsere Testinstanzen wählen wir die Buslinien 12, 14 und 114 im Stadtgebiet Würzburg (siehe Abbildung 7.2). Diese laufen in der Stadtmitte, bei den Haltestellen *Busbahnhof*, *Barbarossaplatz* und *Mainfrankentheater* zusammen, welche die zentralen Transferpunkte des Netzwerks darstellen. Die Buslinien 12 und 14 überschneiden sich zudem an der Haltestelle *Sozialgericht*, und die Linien 14 und 114 an den fünf Haltestellen zwischen *Letzter Hieb* und *Philosophische Fakultät*. Die große Menge an Überschneidungen in diesem Netzwerk sorgt für eine deutlich erhöhte Komplexität im Vergleich zu den Netzwerken von Barth et al. [BRS25]. Um dies auszugleichen und genauer zu untersuchen, erstellen wir eine vereinfachte Version *Wue-simplified* des Netzwerks. Darin wurden die Haltestellen *Barbarossaplatz*, *Mainfrankentheater* sowie *Zollhaus Galgenberg* bis *Philosophische Fakultät* entfernt. Wir haben somit alle redundanten Umsteigepunkte entfernt und erreichen damit einen summierten Grad der Umsteigepunkte von 11, was den Netzwerken von Barth et al. [BRS25] ähnelt. Eine Übersicht über alle verwendeten Netzwerke mit den wichtigsten Kennzahlen gibt die Tabelle 7.1. Für unsere Tests nehmen wir an, dass auf jeder Linie zwei Fahrzeuge mit jeweils sechs freien Sitzplätzen fahren. Außerdem sei die Umsteigedauer b auf zwei Minuten fixiert.
Für die nachfolgenden Experimente erstellen wir zunächst vier unterschiedliche Datensätze:

Statisch Den ersten Datensatz nutzen wir, um die statische Performance unseres Algorithmus zu testen. Wir generieren für jedes Netzwerk 10 zufällige Testinstanzen mit 3 bis 30 Anfragen, die auf ein kurzes Zeitfenster von drei Stunden verteilt sind. Barth et al. [BRS25] zeigten, dass die Länge dieses Zeitfensters keinen signifikanten Einfluss auf die Ergebnisse hat. Vielmehr ist die Anfragendichte entscheidend, weshalb wir nur ein einziges Zeitfenster betrachten. Jede Anfrage hat eine maximale Reisedauer von genau 60 Minuten, welche in allen Netzwerken genügt, um von jeder Haltestelle zu jeder anderen Haltestelle zu gelangen. Weiterhin bestimmen wir eine früheste Abfahrtszeit und eine späteste Ankunftszeit für jede Anfrage. Die beiden Zeiten liegen in unseren Datensätzen genau 60 Minuten auseinander und werden mittels einer Gleichverteilung zufällig

in das dreistündige Zeitfenster platziert, sodass beide Zeitpunkte darin liegen. Die Anfrage beinhaltet mit einer Wahrscheinlichkeit von 90 Prozent einen Passagier, mit einer Wahrscheinlichkeit von 9 Prozent zwei Passagiere und mit einer Wahrscheinlichkeit von 1 Prozent drei Passagiere. Für diesen Datensatz sei der Registrierungs-Zeitpunkt stets der Beginn der Servicezeit.

Iterativ Für den iterativen Datensatz modifizieren wir die Instanzen des statischen Datensatzes, indem wir für alle Anfragen die Registrierungszeit auf genau zehn Minuten vor der frühesten Startzeit setzen. Alle anderen Parameter werden nicht verändert[1].

Dynamisch Der dynamische Datensatz besteht ebenfalls aus jeweils 10 Instanzen für jedes Netzwerk, die zwischen 3 und 30 Transportanfragen enthalten. Die Start- und Endzeiten werden wie im statischen Datensatz bestimmt, und die maximale Reisedauer beträgt ebenfalls 60 Minuten.

Die Registrierungszeit der Anfragen bestimmen wir nach dem folgenden Schema: Mit einer Wahrscheinlichkeit von 50 Prozent wird die Registrierungszeit auf die früheste Startzeit gesetzt. Dies entspricht einem Passagier, der ab dem Zeitpunkt, zu dem er seine Anfrage anmeldet, losfahren kann. Dies stellt einen Extremfall im dynamischen liDARPT dar und ist für das System besonders schwer zu erfüllen. Mit der verbleibenden Wahrscheinlichkeit von 50 Prozent wird die Registrierungszeit auf einen beliebigen Zeitpunkt zwischen Beginn der Servicezeit und der frühesten Startzeit gesetzt

Hybrid Für den vierten, hybriden Datensatz kopieren wir die zuvor generierten dynamischen Instanzen und setzen für jede dritte Anfrage die Registrierungszeit auf den Beginn der Servicezeit. Dies entspricht dem „Voranmelden“ der Transportanfrage. Somit erhalten wir einen realitätsnahen Datensatz, bei dem ein Teil der Transportanfragen zu Beginn eingeplant werden, und anschließend weitere Anfragen während deren Ausführung eintreffen.

Aus praktischen Gründen setzen wir eine maximale Rechenzeit für das MILP von 15 Minuten für statische Instanzen und 3 Minuten für iterative, dynamische und hybride Instanzen. Alle Experimente wurden auf einem 8 Kern AMD Ryzen-7700x Rechner mit 32GB RAM und Windows 11 Pro ausgeführt. Wir verwenden CPLEX 22.1.1. und Python 3.10.

Zum Zweck der Reproduzierbarkeit sind alle Netzwerke, Testinstanzen und Ergebnisse im bereitgestellten GitLab-Repository zu dieser Arbeit enthalten (siehe Seite 34).

[1] In unserer Implementierung können statische Instanzen zur Laufzeit in iterative umgewandelt werden. Hierfür muss lediglich ein Parameter in der config-Datei gesetzt werden.

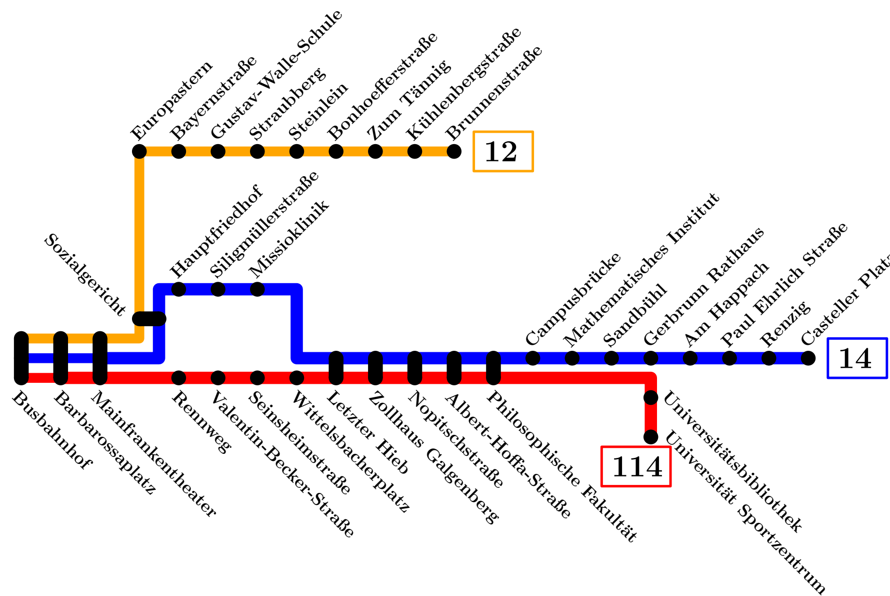


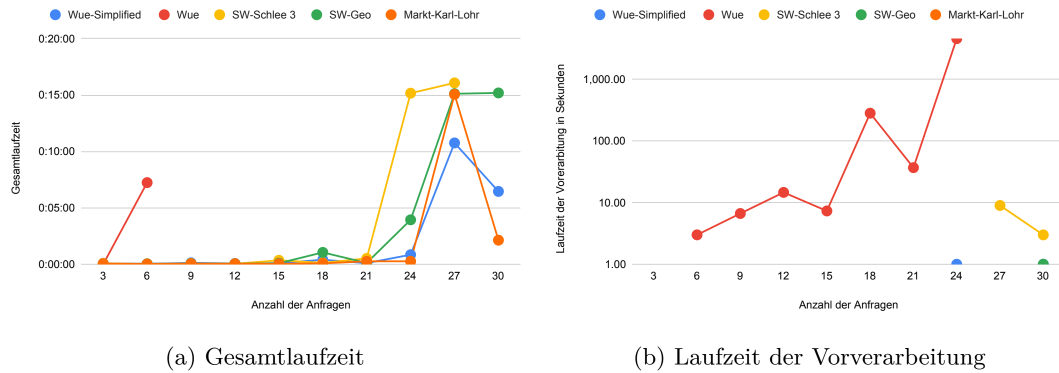
Abb. 7.2.: Schematische Darstellung des Netzwerks *Wue*

7.2. Auswertung der Vorverarbeitung

In Abschnitt 3.2 haben wir ausführlich beschrieben, wie der Event-Based Graph erstellt wird. Die dort genannten Schritte machen neben dem Routenbaum den Großteil der Vorverarbeitung aus, die wir vor der Ausführung des MILPs durchführen. Im Vergleich zu den Arbeiten von Barth et al. [BRS25] und Gaul et al. [GKS21] stellen wir dabei mehr Nebenbedingungen auf, was zu einem kleineren Event-Based Graph führen sollte. In diesem Abschnitt evaluieren wir, (1) wie stark die Laufzeit unseres MILPs von Faktoren wie der Menge an Nebenbedingungen und der Menge der Variablen abhängt, (2) wie effektiv die verwendete Vorverarbeitung darin ist, diese Faktoren zu reduzieren und (3) ob die somit eingesparte Laufzeit geringer ist als die Zeit, die wir für die zusätzliche Vorverarbeitung aufwenden müssen.

Wir beginnen mit einer kurzen Auswertung der wichtigsten Kennzahlen um die Ergebnisse der Vorverarbeitungsschritte besser einordnen zu können.
Wir verwenden für die Auswertung der Vorverarbeitungsschritte den statischen Datensatz (siehe Statisch). In Abbildung 7.3a ist die Gesamtlaufzeit unseres Algorithmus in Relation zur Anzahl der Transportanfragen für alle Netzwerke abgebildet. Dabei sind alle Werte über drei Durchläufe gemittelt. Es fällt auf, dass die Lösungsdauer für fast alle Netzwerke mit mehr als 21 Anfragen schnell ansteigt. Von einer einstelligen Sekundenzahl auf 15 Minuten, also die Zeitbegrenzung des MILPs. Ausgenommen davon ist das Netzwerk *Wue* (in Rot), das bereits mit 6 Anfragen eine Laufzeit von sieben Minuten und 14 Sekunden benötigt und ab 9 Anfragen nicht mehr innerhalb der Zeitbegrenzung gelöst werden kann (Da CPLEX keine ganzzahlige Lösung finden konnte). In Abbildung 7.3b ist die Laufzeit der Vorverarbeitung in Sekunden dargestellt. Diese steigt für

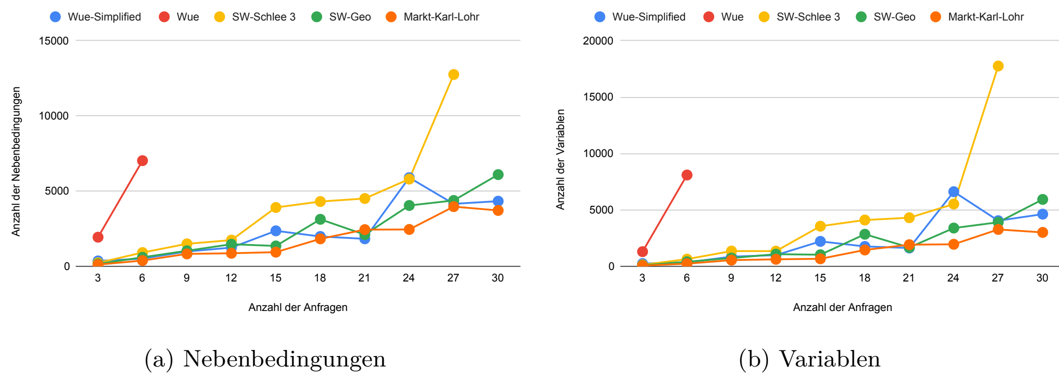
das Netzwerk *Wue* ab 18 Transportanfragen sehr schnell an, sodass die Vorverarbeitung für 24 Anfragen bereits mehr als eine Stunde und 16 Minuten benötigt. Wir haben uns deshalb entschieden, keine weiteren Tests mit noch größerer Anfragendichte im Netzwerk *Wue* durchzuführen. Im Gegensatz dazu benötigt unsere Vorverarbeitung für alle anderen getesteten Netzwerke und alle Anfragendichten weniger als zehn Sekunden, was die deutlich höhere Komplexität des *Wue*-Netzwerks hervorhebt.



(a) Gesamtlaufzeit (b) Laufzeit der Vorverarbeitung

Abb. 7.3.: Vergleich der Gesamtlaufzeit mit der Laufzeit der Vorverarbeitungsschritte in Relation zur Anzahl der Anfragen.

Die Produkt-Moment-Korrelation der Variablenanzahl des MILPs zur Laufzeit und der Anzahl der Nebenbedingungen zur Laufzeit ist 0.68 und 0.73. Dies zeigt, dass es einen starken linearen Zusammenhang zwischen den beiden Parametern und der Laufzeit des MILPs gibt. Die Menge der Variablen und Nebenbedingungen hängen wiederum direkt von der Größe des Event-Based Graphs ab, weshalb die Aufgabe einer guten Vorverarbeitung sein sollte, die Größe des Event-Based Graphs zu reduzieren. Abbildung 7.4 bildet die Anzahl der Nebenbedingungen und Variablen ab.



(a) Nebenbedingungen (b) Variablen

Abb. 7.4.: Die Anzahl der der Nebenbedingungen und Variablen im MILP in Relation zur Anzahl der Anfragen.

Für das Netzwerk *SW-Schlee-3* (in Gelb) liegt die Grenze der Erfüllbarkeit bei 27

Transportanfragen. Alle weiteren Netzwerke können mit bis zu 30 Anfragen erfüllt werden. Nicht immer ist die dabei zurückgegebene Lösung optimal. Die MIP Gap[2] für den statischen Datensatz ist in Abbildung 7.5 dargestellt. Darin ist zu sehen, dass das MILP für die Netzwerke *SW-Geo-Full* und *Markt-Karl-Lohr* einen Höhepunkt von etwa 100 Prozent erreicht, was darauf schließen lässt, dass der CPLEX-Solver für diese Netzwerke mit mehr als 30 Anfragen schnell überlastet wäre.

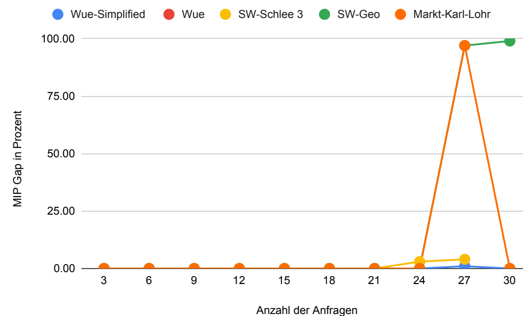
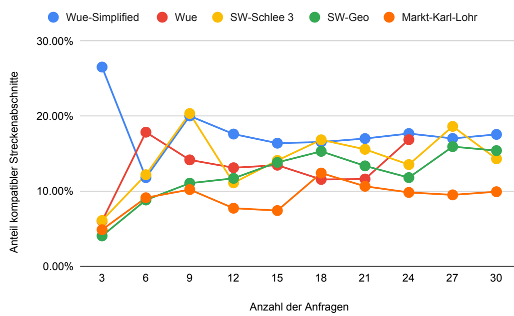


Abb. 7.5.: Relative MIP Gap des MILPs in Relation zur Anzahl der Anfragen.

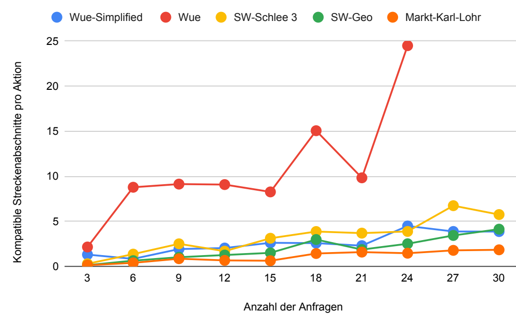
Nun, da wir einige der Hauptkennzahlen für den statischen Algorithmus genannt haben, können wir uns genauer der weiteren Performance der Vorverarbeitung widmen.

In Abschnitt 6.2.5 haben wir bereits die genaue Umsetzung der Vorverarbeitung beschrieben. Wir untersuchen nun die Effektivität der einzelnen, dort beschriebenen Schritte. Wir bestimmen zunächst für jede Aktion die Menge an kompatiblen Streckenabschnitten. Wie in Abbildung 7.6a zu sehen ist, bleibt der durchschnittliche Anteil der kompatiblen Streckenabschnitte in etwa konstant und liegt, abhängig vom Netzwerk, zwischen 10 Prozent und 20 Prozent. Mit steigender Anzahl an Anfragen steigt jedoch auch die Anzahl der möglichen Streckenabschnitte auf jeder Linie. Deshalb steigt auch die Anzahl der kompatiblen Streckenabschnitte pro Aktion, wie in Abbildung 7.6b zu sehen ist.

[2]Die Differenz zwischen dem zurückgegebenen Ergebnis und der besten bisher bekannten unteren Grenze.



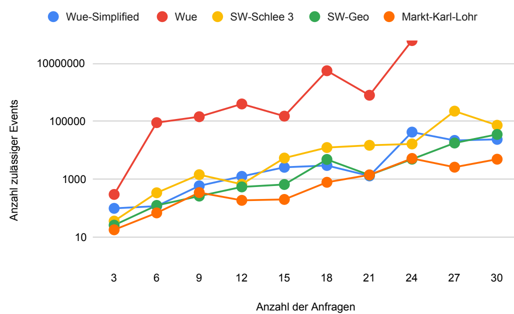
(a) Anteil kompatibler Streckenabschnitte



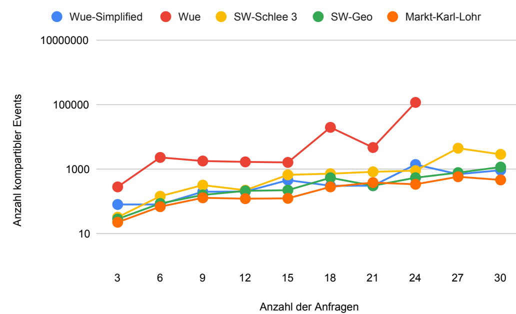
(b) Kompatible Streckenabschnitte pro Aktion

Abb. 7.6.: Der durchschnittliche Anteil kompatibler Streckenabschnitte (links) und die durchschnittliche Anzahl kompatibler Streckenabschnitte pro Aktion (rechts) in Abhängigkeit von der Anzahl der Transportanfragen. Der durchschnittliche Anteil kompatibler Streckenabschnitte wird berechnet als Mittelwert aus den Anteilen der kompatiblen Streckenabschnitte jeder Aktion.

Anschließend berechnen wir alle Kombinationen der kompatiblen Streckenabschnitte und bilden daraus die Menge aller zulässigen Events (siehe Abbildung 7.7a).



(a) Anzahl zulässiger Events



(b) Anzahl Events im Event-Based Graph

Abb. 7.7.: Die Anzahl aller zulässigen Events und der Events im Event-Based Graph in Abhängigkeit von der Anzahl der Transportanfragen.

Für alle zulässigen Events prüfen wir anschließend die Kompatibilität. Mithilfe dieses Schrittes können wir weiter bis zu 80 Prozent der zulässigen Events ausschließen (siehe Abbildung 7.8). Der Anteil der entfernten Events steigt dabei mit steigender Anzahl der Anfragen. Dieser Effekt ist darauf zurückzuführen, dass die steigende Anzahl der kompatiblen Streckenabschnitte pro Aktion dafür sorgt, dass die Menge der Events mit vielen Passagieren steigt. Folglich sinkt die Wahrscheinlichkeit, dass die Abfolgeeinschränkung und Zeitbeschränkung erfüllt werden.

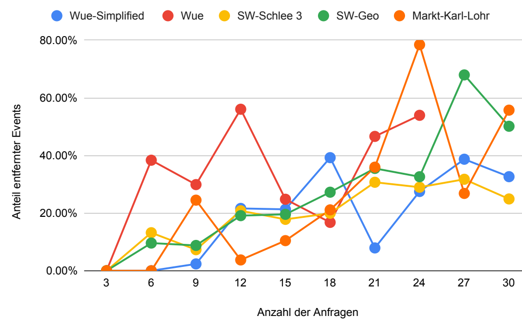


Abb. 7.8.: Anteil der Events, die durch die Vorverarbeitung entfernt werden.

Anschließend werden die verbleibenden kompatiblen Events mithilfe des in Abschnitt 3.2.2 beschriebenen Verfahrens durch Kanten verbunden. Anschließend entfernen wir sämtliche Knoten, die keine eingehenden oder ausgehenden Kanten haben, da diese nicht Teil eines Kreislaufs sein können. Wie in Abbildung 7.9 zu sehen ist, steigt dieser Anteil zunächst mit steigender Anzahl der Anfragen, bis im Bereich zwischen 12 und 18 ein Höhepunkt von etwa 25 Prozent erreicht wird. Steigt die Anfragendichte weiter, so verringert sich der Anteil wieder. Dieses Phänomen erklären wir uns folgendermaßen: Bei geringer Anfragendichte werden, vor allem auf weitläufigeren Netzwerken wie *SW-Geo-Full* und *Markt-Karl-Lohr*, Anfragen meist einzeln Transportiert, d.h. die Events enthalten keine Streckenabschnitte, sondern nur eine Aktion. Solche Events sind stets miteinander und mit einem Idle-Event verbunden und dürfen deshalb nicht entfernt werden. Mit steigender Anzahl an Anfragen machen solche Events einen immer kleineren Anteil der Events aus. Steigt die Anfragendichte weiter, so gibt es immer mehr Events, die potenziell miteinander verbunden sein können, was die Wahrscheinlichkeit wieder verringert, dass ein Event unverbunden bleibt. Diese Theorie wird unterstützt durch den deutlich unterdurchschnittlichen Wert des *Wue* Netzwerks, in dem die Anzahl der Events sehr hoch ist (siehe Abbildung 7.7).

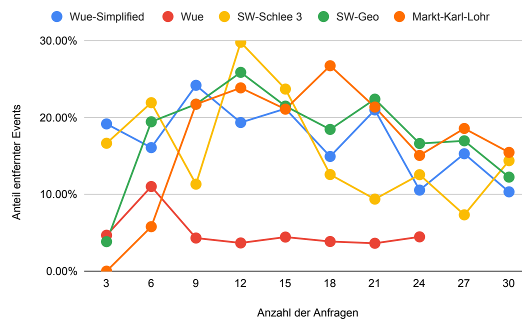


Abb. 7.9.: Anteil der Events, die wegen fehlenden Kanten entfernt werden.

Insgesamt erreichen wir mit den beiden oben genannten Vorverarbeitungsschritten eine erhebliche Verkleinerung des Event-Based Graphs. Bei ausreichend großer Anfra-

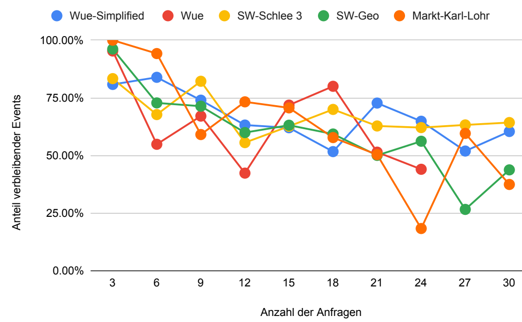


Abb. 7.10.: Anteil verbleibender Events nach der Vorverarbeitung.

gendichte erreichen wir in allen Netzwerken eine Reduzierung von 45 Prozent und bis zu 82 Prozent für das *Markt-Karl-Lohr* Netzwerk (siehe Abbildung 7.10).

Abschließend können wir zusammenfassen, dass die hier vorgestellten Vorverarbeitungsschritte sehr effektiv in der Verkleinerung des Event-Based Graphs sind. Besonders im Hinblick auf den im Vergleich zur Gesamtlaufzeit geringen Laufzeitaufwand, der damit einhergeht.

7.3. Vergleich des statischen und dynamischen liDARPT

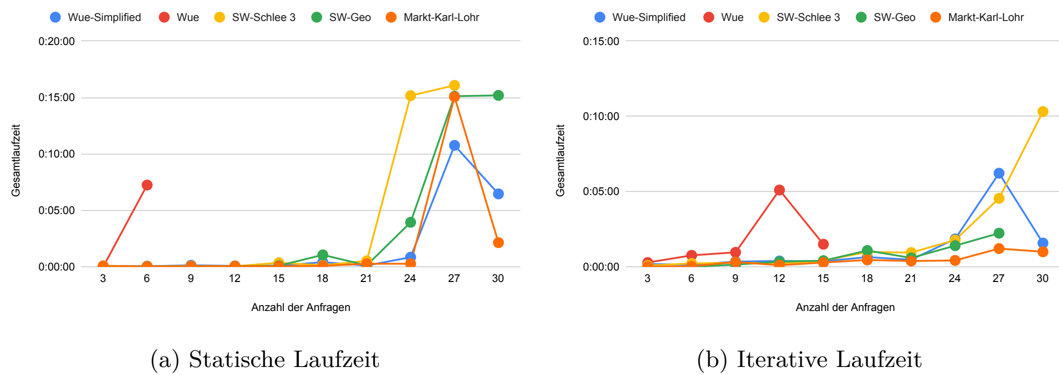
In der Einleitung dieser Thesis haben wir das dynamische liDARPT vor allem mit der Motivation eingeführt, in einer Echtzeitanwendung auf Veränderungen des Systems, beispielsweise durch neue Anfragen, zu reagieren. Es gibt jedoch noch eine weitere Einsatzmöglichkeit unseres Algorithmus. Für größere statische liDARPT-Instanzen steigt die Laufzeit des MILPs enorm (siehe Abbildung 7.3a). Um diese zu verringern, können wir das statische Problem in ein dynamisches Umwandeln. Diesem Verfahren widmen wir die erste Hälfte dieses Abschnitts. Anschließend untersuchen wir den Einfluss der im Voraus bekannten Anfragen auf das dynamische liDARPT.

7.3.1. Iterative Anwendung

Im folgenden Abschnitt vergleichen wir die Ergebnisse des statischen und iterativen Datensatzes (siehe Statisch und Iterativ). Der iterative Datensatz entspricht dabei der Ausführung einer statischen Instanz in $|R|$ Iterationen. Mit diesem Vorgehen kann die Gesamtlaufzeit erheblich reduziert werden, da Teile der Routen bereits ausgeführt wurden und wir den Event-Based Graph entsprechend vereinfachen können. Ein Nachteil besteht allerdings darin, dass ein Fahrgast, der am Vortag bereits seinen Transportwunsch geäußert hat, erst 10 Minuten vor Beginn seines Transportzeitfensters erfährt, ob er angenommen oder abgelehnt wird. Die gewählte Zeit, zu der die Anfrage in das System aufgenommen wird, bietet dabei einen einstellbaren Parameter zwischen Nutzerfreundlichkeit und Qualität der Ergebnisse bei früherem Einfügen und geringerer Laufzeit bei späterem Einfügen. Anhand der folgenden Experimente evaluieren wir die

Tauglichkeit der iterativen Heuristik zur Vereinfachung komplexer statischer Instanzen. Gleichzeitig können wir aus den Ergebnissen allgemeine Schlüsse über die Performance des dynamischen Algorithmus ziehen.

Zunächst betrachten wir die Laufzeit. Die durchschnittliche Laufzeit des statischen Datensatzes beträgt 2 Minuten und 44 Sekunden, während die des iterativen Datensatzes 1 Minute und 9 Sekunden beträgt. Dabei ist zu Erwähnen, dass im iterativen Datensatz in den Netzwerken *Wue* und *SW-Schlee-3* größere Instanzen gelöst werden können als im statischen Datensatz. Entfernen wir diese Instanzen, so erhalten wir eine durchschnittliche Laufzeit von nur 49 Sekunden, was einer Verbesserung von 73 Prozent entspricht. In Abbildung 7.11 ist außerdem zu sehen, dass der Anstieg der Laufzeitkurve mit dem iterativen Verfahren deutlich langsamer ist als mit dem statischen Verfahren.

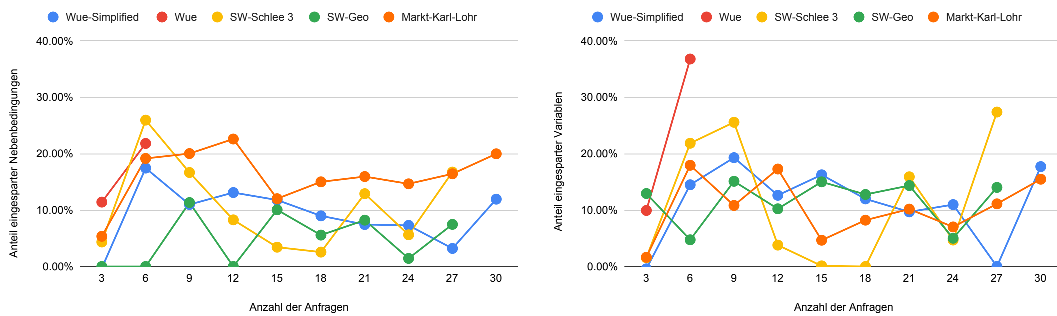


(a) Statische Laufzeit

(b) Iterative Laufzeit

Abb. 7.11.: Vergleich der Gesamtlaufzeit des statischen und iterativen Datensatzes in Relation zur Anzahl der Anfragen.

Wie wir in Abschnitt 7.2 festgestellt haben, hängt die Laufzeit des MILPs maßgeblich von der Größe des Event-Based Graphs ab. Wir betrachten deshalb den Anteil der eingesparten Nebenbedingungen und Variablen in der letzten Iteration des Algorithmus im Vergleich zur statischen Variante in Abbildung 7.12. Dabei fällt auf, dass die Einsparungen in Relation zur Größe des MILPs (siehe Abbildung 7.4) eher gering ausfallen. Eine Beschleunigung der Laufzeit tritt trotzdem ein, da viele der Variablen im iterativen Verfahren bereits festgelegt sind (wie in Kapitel 5 beschrieben) oder durch den Warmstart des MILPs bereits mit einer guten Lösung starten (siehe Abschnitt 6.2.4).



(a) Anteil eingesparter Nebenbedingungen

(b) Anteil eingesparter Variablen

Abb. 7.12.: Der Anteil aller eingesparten Nebenbedingungen und Variablen im Vergleich zum statischen Datensatz in Abhängigkeit von der Anzahl der Transportanfragen.

Nun vergleichen wir die Qualität der Ergebnisse für den statischen und den iterativen Datensatz. Wir überprüfen anhand gängiger Kennzahlen (aus [Deu25] und [BRS25]), wie weit die Lösungen des iterativen Datensatzes von den optimalen Lösungen abweichen, die wir für den statischen Datensatz berechnet haben. Die einfachste Qualitätsmetrik für ein Bedarfsverkehrssystem ist die Quote der akzeptierten Anfragen. Insgesamt werden in statischen Instanzen durchschnittlich 99.39 Prozent der Anfragen akzeptiert und in iterativen Instanzen 98.66 Prozent. Im statischen Datensatz werden in nur vier von 50 Instanzen nicht alle Anfragen akzeptiert, von denen alle mindestens 27 Anfragen enthalten und drei eine MIP Gap von über 90 Prozent haben. Die niedrigste Akzeptanzrate aus allen Instanzen liegt bei 93.33 Prozent.

Im iterativen Datensatz werden in 6 von 50 Instanzen nicht alle Anfragen akzeptiert, wobei die niedrigste Akzeptanzrate bei 80 Prozent liegt. In keinem der Fälle ist eine signifikante MIP Gap zu vermerken. Weiterhin treten Ablehnungen bereits bei Instanzen mit 24 Transportanfragen auf.

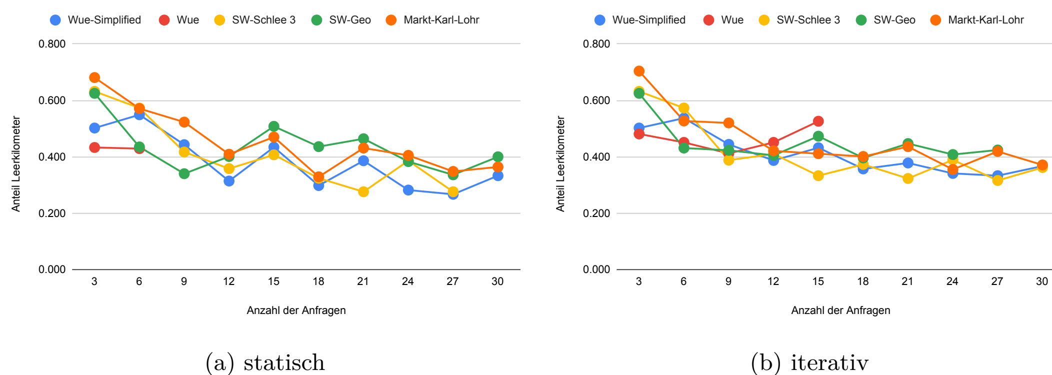
Kommen wir nun zur zweiten Eigenschaft, die wir direkt in der Zielfunktion des MILPs optimieren: Der Reisezeit der Passagiere. Um wettbewerbsfähig zu sein, sollte die Fahrzeit im ÖPNV nicht länger als das 1.5-fache einer Autofahrt betragen (siehe [Deu25]). Der durchschnittliche Umweg, den die Fahrzeuge in unseren Instanzen nehmen, liegt sowohl für den statischen als auch für den iterativen Datensatz bei 1.77 und damit leicht oberhalb des Zielwerts.

Ein weiterer wichtiger Aspekt für die Zufriedenheit der Passagiere ist eine mittlere Wartezeit von unter 30 Minuten (siehe [Deu25]). Im statischen Datensatz erreichen wir einen durchschnittlichen Wert von 10 Minuten und im iterativen 9 Minuten und 14 Sekunden. Werte über 30 Minuten treten dabei vor allem im Netzwerk *Markt-Karl-Lohr* auf, da dieses die größten Distanzen enthält und Fahrzeuge somit mehr Zeit benötigen, um die Abholpunkte zu erreichen. Eine Verbesserung der Wartezeit im iterativen Datensatz ist möglich, da die Wartezeit kein Wert ist, der direkt in das Optimierungsproblem einfließt.

Ähnlich wichtig aus Nutzersicht ist eine geringe Anzahl an Umstiegen. Diese bleibt, ähnlich wie die Wartezeit, identisch in beiden Datensätzen, bei etwa 0.7 Umstiegen pro Anfrage.

Nun betrachten wir genauer solche Metriken, die für ein Busunternehmen interessant wären, das das vorgestellte System verwenden würde.
Wir beginnen mit der Systemeffizienz. Diese Metrik beschreibt den Anteil der Summe aller gebuchten Kilometer an der Summe der zurückgelegten Kilometer aller Fahrzeuge. Die gebuchten Kilometer sind die Kilometer der kürzesten Strecke zwischen dem Start- und Endpunkt einer Anfrage. Werte > 1 bedeuten, dass die Fahrzeuge insgesamt weniger Strecke zurücklegen als die einzelnen Passagiere, wenn sie mit dem PKW fahren würden. Auf dem statischen Datensatz erreicht unser Algorithmus eine durchschnittliche Systemeffizienz von 0.54, während die iterativen Instanzen einen Mittelwert von 0.52 haben. Diese Effizienz ist vergleichsweise niedrig, da aufgrund der gewählten Zielfunktion des MILPs nur ein geringer Anreiz besteht, die Leerfahrten möglichst kurz zu halten.

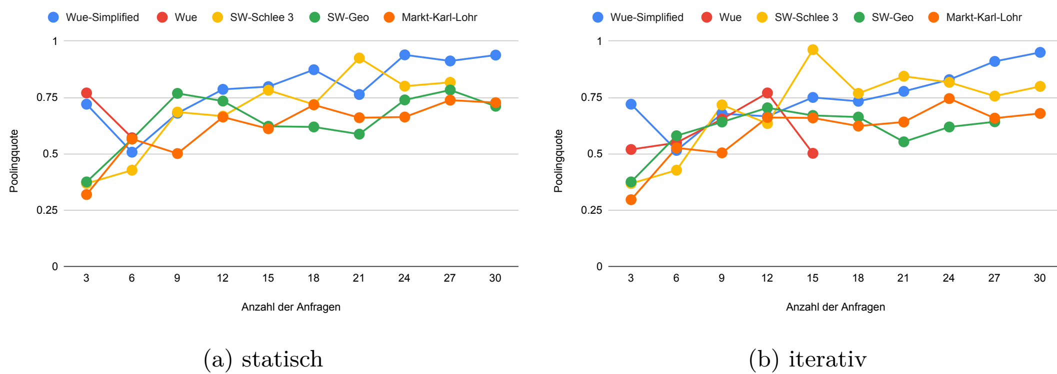
Ähnliche Ergebnisse liefert uns der Anteil der Leerkilometer an der gesamten zurückgelegten Strecke durch die Fahrzeuge (siehe Abbildung 7.14). Hier erreicht der Algorithmus auf dem statischen Datensatz im Durchschnitt einen Wert von 41.9 Prozent und im iterativen Datensatz 43.4 Prozent. Besonders lange Netzwerke wie *Markt-Karl-Lohr* und *SW-Geo-Full* haben einen größeren Anteil an Leerkilometern.



(a) statisch (b) iterativ

Abb. 7.13.: Der Anteil der Leerkilometer an der insgesamt zurückgelegten Strecke in Relation zur Anzahl der Anfragen.

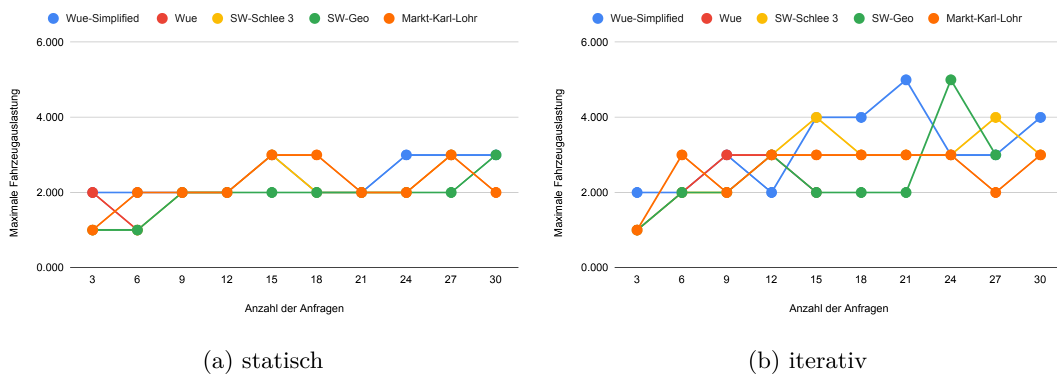
Die Poolingquote (auch Bündelungsquote) berechnet sich aus der Summe aller gebuchten Kilometer, geteilt durch die insgesamt zurückgelegte Strecke aller Fahrzeuge. Der Wert liegt im statischen Datensatz bei 0.686 und beim iterativen mit 0.660 geringfügig niedriger. Es fällt vor allem auf, dass die Poolingquote bei kürzeren Netzwerken wie *Wue*, *Wue-simplified* und *SW-Schlee_3* höher ausfällt als bei anderen Netzwerken. Diese Erkenntnisse spiegeln die Ergebnisse aus der Untersuchung der Leerkilometer wider.



(a) statisch (b) iterativ

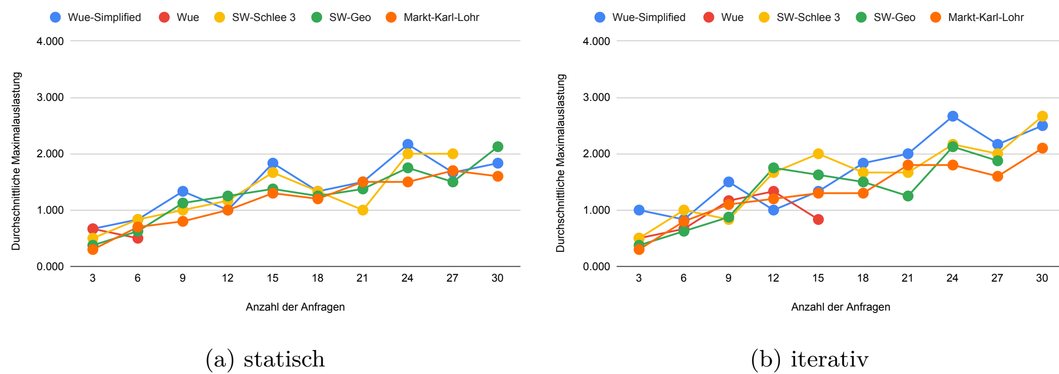
Abb. 7.14.: Die Poolingquote in Relation zur Anzahl der Anfragen.

Als letztes betrachten wir die Auslastung der Fahrzeuge. Dafür werten wir zunächst die maximale Fahrzeugauslastung in Abbildung 7.15 aus. Dabei fällt auf, dass die Maximalauslastung im iterativen Datensatz deutlich höher ist als im statischen. Dies widerspricht der intuitiven Annahme, dass eine höhere Fahrzeugauslastung auch für höhere Effizienz spricht. Denn wie wir bereits sehen konnten, ist die Poolingquote und Systemeffizienz beim iterativen Verfahren geringfügig niedriger als beim statischen. Bei der Betrachtung der durchschnittlichen maximalen Fahrzeugauslastung zeichnet sich ein ähnliches Bild ab (siehe Abbildung 7.16).



(a) statisch (b) iterativ

Abb. 7.15.: Die maximale Auslastung der Fahrzeuge in Relation zur Anzahl der Anfragen



(a) statisch

(b) iterativ

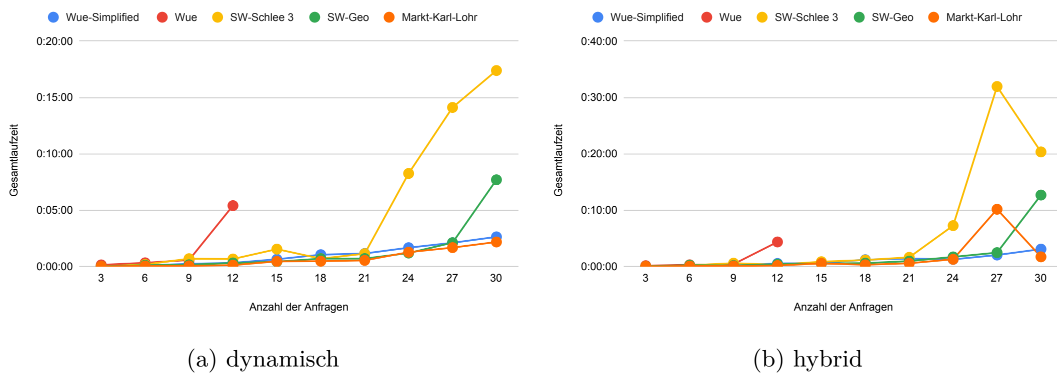
Abb. 7.16.: Die durchschnittliche maximale Auslastung der Fahrzeuge in Relation zur Anzahl der Anfragen

Zusammenfassend können wir sagen, dass die iterative Ausführung einer statischen Instanz zu einer deutlich verringerten Laufzeit führt und somit besonders in Instanzen mit einer hohen Dichte an Anfragen nützlich sein kann. Unser Algorithmus liefert in dynamischen Instanzen in vielen Metriken wie der zurückgelegten Strecke, der Wartezeit und der Anzahl der Umstiege annähernd identische Ergebnisse. In anderen Metriken wie der Akzeptanzrate, der Systemeffizienz, dem Anteil der Leerkilometer und der Poolingquote sind die erzielten Ergebnisse im Durchschnitt weniger als 4 Prozent schlechter als die des statischen Datensatzes.

7.3.2. Dynamische Anwendung

In diesem Abschnitt untersuchen wir, welchen Einfluss die Anzahl der vorab bekannten Transportanfragen auf die Performance des Algorithmus hat. Dafür vergleichen wir den dynamischen Datensatz (siehe Dynamisch) mit dem hybriden Datensatz (siehe Hybrid).

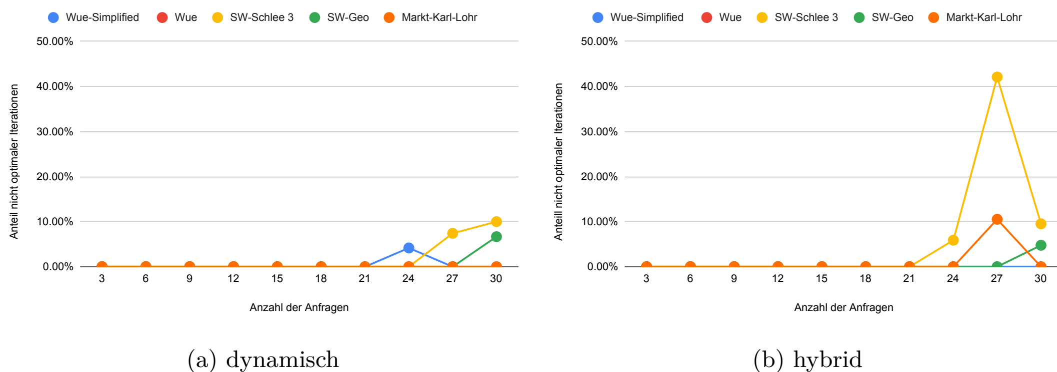
Wir beginnen die Auswertung erneut mit dem Vergleich der Laufzeiten der Datensätze. In Abbildung 7.17 ist zu sehen, dass die Laufzeit des hybriden Datensatzes deutlich länger ist als die des dynamischen Datensatzes. Der Grund dafür ist, dass die vorangemeldeten Transportanfragen dafür sorgen, dass die Größe des Event-Based Graphs bereits in den ersten Iterationen sehr hoch ist. Dieser Effekt kann nicht durch die um ein Drittel verringerte Anzahl an Iterationen kompensiert werden. Der hybride Datensatz erreicht deshalb eine durchschnittliche Laufzeit von 2 Minuten und 36 Sekunden, während der Algorithmus für den dynamischen Datensatz im Mittel nur 1 Minute und 50 Sekunden benötigt, was einer Verschlechterung von 40 Prozent entspricht. Es fällt jedoch deutlich auf, dass die Verschlechterung der Laufzeit vor allem in Instanzen mit höherer Anfragendichte auftritt, sodass die schlechteste Laufzeit im dynamischen Datensatz bei 17 Minuten 23 und die schlechteste Laufzeit im hybriden bei 31 Minuten und 57 Sekunden liegt, was einem Anstieg von 84 Prozent entspricht.



(a) dynamisch (b) hybrid

Abb. 7.17.: Die Gesamtlaufzeit des Algorithmus für den dynamischen und hybriden Datensatz in Relation zur Anzahl der Anfragen.

Wie in Abbildung 7.18 zu sehen ist, erhöht sich der Anteil der nicht optimalen Lösungen des MILPs. Das Netzwerk *SW-Schlee_3* schneidet dabei am schlechtesten ab, sodass CPLEX in einer Instanz mit 27 Anfragen in 8 von 18 Iterationen im hybriden Datensatz suboptimale Ergebnisse liefert. In anderen Netzwerken wie *SW-Geo_full* und *wue-simplified* hingegen treten solche Fälle weniger häufig im hybriden als im dynamischen Datensatz auf, was vermuten lässt, dass diese Kennzahl stark von der konkreten Auswahl der vorangemeldeten Anfragen abhängt.



(a) dynamisch (b) hybrid

Abb. 7.18.: Die Anzahl der nicht optimalen MILP-Lösungen auf den beiden Datensätzen in Relation zur Anzahl der Anfragen.

Die Größe der durchschnittlichen MIP Gap verhält sich grundsätzlich ähnlich zur Anzahl der nicht optimalen Iterationen. Es fällt jedoch auf, dass der Anstieg des Höchstwerts im *SW_Schlee 3* Netzwerk mit 42 Prozent geringer ausfällt.

(a) dynamisch (b) hybrid

Abb. 7.19.: Der durchschnittliche MIP Gap-Wert auf den beiden Datensätzen in Relation zur Anzahl der Anfragen.

Ein gutes Maß für die Komplexität der Instanzen ist die durchschnittliche Zeit zwischen Registrierung und der frühesten Startzeit der Anfragen. In dieser Zeit befinden sich die Anfragen im Zustand $\mathcal{S}$ und haben noch alle Routen offen, was zu einem großen Event-Based Graph führt. Diese Kennzahl steigt im hybriden Datensatz durchschnittlich um 206 Prozent gegenüber dem dynamischen, wie in Abbildung 7.20 zu sehen ist.

Netzwerk	Zeitdifferenz
Markt-Karl-Lohr	17.00
SW-Geo_full	16.64
SW-Schlee_3	15.12
Wue	17.28
Wue-simplified	17.82

(a) dynamisch

Netzwerk	Zeitdifferenz
Markt-Karl-Lohr	52.62
SW-Geo_full	52.68
SW-Schlee_3	48.92
Wue	50.61
Wue-simplified	51.80

(b) hybrid

Abb. 7.20.: Durchschnittliche Zeit zwischen Registrierungs- und Startzeit (in Minuten) der Anfragen pro Netzwerk im dynamischen und hybriden Datensatz.

Anschließend untersuchen wir die Unterschiede in den Ergebnissen des Algorithmus. Wir beginnen erneut mit der Quote der akzeptierten Anfragen. Diese liegt für beide Datensätze bei 97.75 Prozent.

Eine Verbesserung im hybriden Datensatz ist hingegen bei den zurückgelegten Strecken der Fahrzeuge zu sehen. Die durchschnittlich zurückgelegte Leerstrecke ist im dynamischen Datensatz 122.58 km und im hybriden 114.38 km, was einer Verbesserung von 7.2 Prozent entspricht, während sich die durchschnittlich zurückgelegte Strecke mit Passagieren um lediglich 1.1 Prozent von 168.42 km auf 166.53 km verbessert. Dies ist erneut darauf zurückzuführen, dass wir im MILP kurze Transportstrecken für Fahrgäste stärker gewichten als kurze Leerstrecken.

Als nächstes betrachten wir einige Metriken der Kundenzufriedenheit. Wir beginnen

mit der durchschnittlichen Wartezeit, die für den dynamischen Datensatz mit 9 Minuten und 58 Sekunden leicht unter den 10 Minuten und 18 Sekunden des hybriden Datensatzes liegt. Bei der durchschnittlichen Reisezeit der Anfragen konnte kein Unterschied festgestellt werden. Beide Datensätze liegen bei 26 Minuten und 58 Sekunden.

In der Anzahl der Umstiege konnte ebenfalls kein signifikanter Unterschied zwischen den Datensätzen gemessen werden.

Wir betrachten nun die Effizienzmetriken des Algorithmus und beginnen mit der Systemeffizienz. Diese beträgt für den dynamischen Datensatz 0.507 und für den hybriden Datensatz 0.521, was einer Verbesserung von 2.8 Prozent entspricht. Bei der Poolingquote stellen wir ebenfalls eine geringfügige Verbesserung von 0.635 auf 0.656 (3.3%) fest.

Wir können zusammenfassend sagen, dass der hybride Datensatz besonders in Bezug auf die Laufzeit und die MIP Gap deutliche Schwierigkeiten für unseren Algorithmus bringt. Die schnell ansteigende Größe des MILPS durch die vorangemeldeten Anfragen bedeutet, dass die festgelegte Laufzeit von drei Minuten oftmals nicht genügt. Trotzdem bringt das Voranmelden von Anfragen eine Verbesserung in der Länge der Leerfahrten, der Systemeffizienz und der Poolingquote.

7.4. Weitere Erkenntnisse

In diesem Abschnitt behandeln wir kurz einige weitere Erkenntnisse, die sich aus den durchgeführten Tests ergeben haben.

Routenwahl Im Gegensatz zum Algorithmus von Barth et al. [BRS25] haben wir nicht die Einschränkung getroffen, dass Routen nur maximal einen Umstieg mehr enthalten dürfen, als mindestens nötig wäre. Wir hatten uns dabei erhofft, vor allem bei Instanzen mit hoher Anfragendichte bessere Ergebnisse zu erhalten. Unser Algorithmus wählte jedoch in keinem der Tests eine solche Route. Wir schließen daraus, dass die von Barth et al. [BRS25] getroffene Einschränkung für ein System mit maximal 10 Anfragen pro Stunde sinnvoll ist. Eine weitere Untersuchung mit noch höherer Anfragendichte könnte jedoch zu einem anderen Ergebnis kommen.

Startzeitänderung Unser dynamischer Algorithmus erlaubt es, bereits akzeptierte, aber noch nicht abgeholte Anfragen umzuplanen. Daraus folgt, dass eine kurzfristige Änderung der Startzeit möglich wäre, was eine schlechte Planbarkeit für den Nutzer bedeuten würde. In den von uns durchgeführten Tests kamen jedoch keine solchen Ereignisse vor. Grund dafür sind vermutlich zwei Faktoren: (1) Durch den Einsatz eines Warmstarts startet jede Lösungssuche mit denselben Ausführungszeiten B (2) Da unser MILP keine Optimierung der Wartezeiten durchführt, ist der konkrete Startzeitpunkt irrelevant, solange die gefahrene Strecke gleich bleibt. Es gibt demnach keinen Anreiz für CPLEX den Startzeitpunkt zu verschieben.

Außerdem ist die durchschnittliche Anzahl der Neuberechnungen vor Beginn der Startzeit in den Testinstanzen sehr gering (siehe Abbildung 7.21). Für Testinstanzen mit noch höherer Anfragendichte würden solche Fälle häufiger auftreten.

Netzwerk	# Neuberechnungen
Markt-Karl-Lohr	2.59
SW-Geo_full	2.24
SW-Schlee_3	2.18
Wue	2.55
Wue-simplified	2.6

(a) dynamisch

Netzwerk	# Neuberechnungen
Markt-Karl-Lohr	4.16
SW-Geo_full	3.85
SW-Schlee_3	3.62
Wue	3.76
Wue-simplified	3.98

(b) hybrid

Abb. 7.21.: Durchschnittliche Anzahl an Neuberechnungen zwischen Registrierungs- und Startzeit der Anfragen pro Netzwerk im dynamischen und hybriden Datensatz.

Urbane Netzwerke Nun wollen wir evaluieren, wie gut ein dynamisches, linienbasiertes Rufbussystem mit Umstiegen für städtische Umgebungen geeignet ist.Dafür vergleichen wir die Netzwerke *Wue* und *Wue-simplified* mit den verbleibenden Netzwerken. Mögliche Anwendungsfälle wären beispielsweise die Anbindung von Randgebieten an das zentrale öffentliche Verkehrsnetz oder als Nachtbus. Als Vorteil eines urbanen Einsatzgebietes erweist sich die, durch die kürzeren Distanzen, sinkende Länge der Leerfahrten, was folglich die Kosten für den Betreiber verringert. Außerdem beobachten wir eine höhere Poolingquote und Fahrzeugauslastung, was für eine bessere Kosteneffizienz spricht. Ein Nachteil der urbanen Umgebung ist die deutlich gestiegene Komplexität des MILPs. Diese entsteht, da die kürzesten Reisezeiten im Verhältnis zu den maximal erlaubten Reisezeiten sehr niedrig sind, was zu einem größeren Spielraum in der Planung führt. Ein weiterer Faktor ist die tendenziell höhere Anzahl und Dichte an Umsteigepunkten. Letzteres kann jedoch in der Planung der Netzwerke berücksichtigt werden und, wie wir bei *wue-simplified* gesehen haben, bereits zu einer deutlich verbesserten Laufzeit führen.

8. Ausblick

In diesem Kapitel geben wir einen Ausblick auf die zukünftige Forschung im Bereich des Dial-a-ride Problems und Ansatzpunkte zur Verbesserung des vorgestellten Algorithmus.

8.1. Echte dynamische Ausführung

Bei der Implementierung des dynamischen Algorithmus haben wir einige Vereinfachungen getroffen, die für den Real-World Einsatz ungeeignet wären. Wir nehmen beispielsweise an, dass die Berechnungen unseres Algorithmus keine Zeit benötigen. Hierzu ein Beispiel: Angenommen, eine neue Anfrage r erreicht zum Zeitpunkt t das System, und die früheste Startzeit $e(r)$ sei ebenfalls t. Unser MILP hat eine maximale Rechenzeit von drei Minuten. Das bedeutet, dass wir spätestens zum Zeitpunkt $r + 3$ einen gültigen Plan errechnet haben. In diesem Plan ist es jedoch unmöglich, die Anfrage r noch zum Zeitpunkt t abzuholen, da dieser bereits überschritten ist. Dem müsste vorgebeugt werden, indem die früheste Startzeit der Anfrage im Vorhinein um drei Minuten nach hinten verschoben wird. Ebenfalls sind die Fahrzeuge in der Zwischenzeit nach dem bestehenden Plan weitergefahren, was bedeutet, dass Position und Zustand der Fahrzeuge sich bereits verändert haben können. Für eine Real-World Anwendung müssen solche Effekte beachtet werden und beispielsweise durch Simulation die zukünftigen Zustände der Fahrzeuge zum Ende der Rechenzeit geschätzt werden. Diese Abschätzung wird noch schwieriger, wenn wir das System als nichtdeterministisch betrachten.

8.2. Nichtdeterministisches liDARPT

Das hier vorgestellte liDARPT ist grundsätzlich deterministisch. Das bedeutet, dass die vorhergesehenen Reisezeiten immer eingehalten werden. Der hier vorgestellte Algorithmus könnte mit einigen Veränderungen jedoch auch für das stochastische liDARPT verwendet werden, da das verwendete System dynamisch bei Verspätungen umplanen könnte. Hierfür müsste jedoch beispielsweise die Einhaltung der spätesten Zeitfenster im MILP gelockert werden, da ansonsten eine größere Verspätung auf der Route dazu führen könnte, dass eine akzeptierte Route nicht mehr fristgerecht abgeliefert wird und das MILP somit unmöglich wird.
In diesem Kontext könnte auch die Zielfunktion des MILPs angepasst werden, sodass gewisse Wartezeiten der Busse als „Puffer“ eingeplant werden, damit kleinere Verspätungen nicht gleich zu Planänderungen führen.

8.3. Vorverarbeitung

Die zuvor genannte Abfolgeeinschränkung können wir weiter verschärfen: Wollen mehrere Nutzer an einer Haltestelle einsteigen, so darf der Fahrgast mit der frühesten Startzeit zuerst einsteigen. Umgekehrt gilt, dass der Fahrgast mit der spätesten Ankunftszeit zuletzt aussteigt. Wir beschränken somit die möglichen Event-Abfolgen, die während eines Halts stattfinden können. Wir schließen deshalb Ausstiegs-Events v mit Ausstiegsaktionen ρ aus, bei denen die Aussage $\exists \psi = (\rho^+, \rho^-) \in \Psi(v) : s(\rho^-) = s(\rho) \wedge l(r(\psi)) < l(r(\rho))$ wahr ist. Ähnlich gehen wir für Einstiegs-Events v mit Einstiegsaktionen ρ vor, bei denen die Aussage $\exists \psi = (\rho^+, \rho^-) \in \Psi(v) : s(\rho^+) = s(\rho) \wedge e(r(\psi)) > e(r(\rho))$ wahr ist. Einzelne Tests, die wir mit diesem zusätzlichen Vorverarbeitungsschritt durchgeführt haben, deuten an, dass die Anzahl der zulässigen Events damit um weitere 20 Prozent reduziert wird. Eine genauere Untersuchung wäre im Rahmen weiterer Forschung von Interesse.

8.4. Integration

Rufbusse sind meist nur ein Teil eines größeren ÖPNV-Konzepts (siehe [Deu25]). Als solches muss es sich in ein bestehendes Bus- und Zugnetzwerk integrieren, um mit bestmöglicher Effizienz genutzt zu werden. Beispielsweise könnte ein Passagier angeben, dass er an seiner Ankunftshaltestelle mit einem Linienbus weiterfahren möchte. Ein effizientes DARP-System, könnte daraufhin die Ankunftszeit des Passagiers so optimieren, dass die Wartezeit möglichst gering bleibt.

8.5. Ohne MIP Gap

Wie wir in Abbildung 7.19 gesehen haben, steigt die MIP Gap bei Instanzen mit hoher Anfragendichte, was zu einer Verschlechterung der Lösungsergebnisse führen könnte. Besonders problematisch ist, falls das MILP in mehreren aufeinanderfolgenden Iterationen eine stetig steigende MIP Gap aufweist, da neue Anfragen eintreffen, aber keine bereits aufgenommenen Anfragen abgeliefert werden. Im schlechtesten Fall, wie im Netzwerk *Wue*, wird das MIP Gap dabei so groß, dass keine gültige Lösung mehr gefunden werden kann. Diesem Phänomen können wir vorbeugen, indem wir nicht optimale Lösungen anders behandeln: Anstatt eine gültige, aber suboptimale Lösung, wie bisher, einfach in Kauf zu nehmen und auszuführen, lehnen wir alle neu eingetroffenen Anfragen dieser Iteration ab und fahren mit dem letzten optimalen Plan fort. Mit dieser Strategie können wir den Event-Based Graph wieder vereinfachen und die nächste Iteration hoffentlich optimal Lösen. Eine experimentelle Auswertung dieses Verfahrens steht jedoch noch aus. Ein sinnvoller Vergleich ist nur bei Instanzen möglich, bei denen der reguläre dynamische Algorithmus ein nicht optimales Ergebnis liefert, aber noch lösbar ist. Dies kommt bei unserem Algorithmus in nur wenigen fällen vor, da die Laufzeit des MILPs schnell wächst. Mit den von uns getesteten Fällen erreichen wir deshalb keine aussagekräftigen Ergebnisse. Eine genauere Auswertung dieses Verfahrens wäre deshalb ein Ansatzpunkt für zukünftige Forschung.

8.6. Multimodale Kapazität

Eine weit verbreitete Variante des DARP ist das sogenannte Nutzerorientierte DARP. Diese wird beispielsweise von Nasri et al. [NBM21] behandelt. Die Autoren definieren die Servicequalität als Kombination aus Komfort, Zuverlässigkeit, Bequemlichkeit der Reservierung, Ausmaß des Service, Zugänglichkeit (im Sinne der Barrierefreiheit) und Sicherheit. Vor allem die Barrierefreiheit spielt in vielen Arbeiten zum Dial-a-Ride Problem, wie beispielsweise „Handicapped Person Transportation“ von Rekiek et al. [RDS06], eine Rolle, da Rufbusse in der Praxis oftmals für den Transport von körperlich eingeschränkten Personen eingesetzt werden. In diesem Kontext kann ebenfalls das mehrdimensionale DARP-Problem von Wong et al. genannt werden [WH06]. Dabei wird die Kapazität eines Fahrzeugs unterteilt in beispielsweise Gepäckplatz, Rollstuhlfahrerplätze und normale Sitzplätze.
Eine Erweiterung des liDARPT in Richtung der Servicequalität oder Multimodalität wäre ebenfalls ein Ansatzpunkt für zukünftige Forschungen.

8.7. Fahrtzeiten für Busse

Bisher legen wir für alle Busse einer Linie dieselbe Servicezeit fest. Stattdessen könnte jedes Fahrzeug eine eigene Start- und Endzeit zugewiesen bekommen. Dies ermöglicht beispielsweise, dass abhängig von der Uhrzeit unterschiedlich viele Busse auf der Linie eingesetzt werden können. Über diese Funktion können außerdem verpflichtende Pausen für die Fahrer modelliert werden. Indem ein Fahrzeug zu einem festgelegten Zeitpunkt zum Depot zurückkehren muss, kann der Beginn der Pausenzeit festgelegt werden. Nach Ablauf der Pausenzeit kann das Fahrzeug dann unter einer anderen Nummer weiterfahren. Diese Erweiterung erfordert jedoch, dass Fahrzeuge im MILP unterschiedlich behandelt werden müssen, sodass die Servicezeiten eingehalten werden.

9. Zusammenfassung und Fazit

In dieser Arbeit haben wir, aufbauend auf der Arbeit von Barth et al. [BRS25], einen dynamischen Lösungsalgorithmus für das liDARPT entwickelt und implementiert. Wir verwenden eine eigens entwickelte Routenbaum-Datenstruktur um dynamisch die möglichen Routen der Transportanfragen zu speichern. Anschließend modellieren wir mithilfe des Event-Based Graphs die möglichen Fahrzeugzustände und ihre Übergänge. Dabei nutzen wir mehrere Vorverarbeitungsschritte, um die Anzahl der Knoten des Event-Based Graphs um bis zu 82 Prozent zu reduzieren. Anschließend modifizieren wir das MILP von Barth et al. [BRS25], um optimale Fahrpläne für die Busse zu finden, die die Anzahl der akzeptierten Anfragen maximieren und die Fahrtstrecke der Passagiere minimieren.

Diesen Algorithmus implementieren wir in Python und führen anhand von vier unterschiedlichen Datensätzen Experimente durch. Diese Datensätze enthalten Instanzen aus fünf Netzwerken, von denen drei an den Netzwerken von Barth et al. angelehnt sind und zwei selbst anhand realer Buslinien aus dem Stadtgebiet Würzburg abgeleitet wurden. Dabei verwenden wir realitätsnahe Koordinaten und Reisezeiten um möglichst wirklichkeitsnahe Ergebnisse zu erhalten.

Eine eingangs durchgeführte Auswertung der Vorverarbeitungsschritte ergibt, dass wir in verhältnismäßig sehr kurzer Zeit einen Großteil der Knoten des Event-Based Graphs entfernen können und somit die Laufzeit des Algorithmus insgesamt deutlich verbessern.

Anschließend testen wir einen heuristischen Ansatz zur Lösung des statischen liDARPT bei dem wir die Anfragen der Instanzen erst kurz vor Beginn ihres Transportzeitfensters in das System einfügen, um eine Verringerung der Laufzeit zu erreichen und die Planung von noch größeren Instanzen zu ermöglichen. Dabei erreichten wir eine Laufzeitverbesserung von 73 Prozent, während sich die Ergebnisse des Algorithmus anhand verschiedener Metriken um weniger als 4 Prozent verschlechtern.

Im Anschluss untersuchten wir dynamische Testinstanzen und evaluierten, welchen Einfluss die Anzahl der im voraus bekannten Anfragen auf das Ergebnis hat. Dabei ergibt sich, dass sich die Laufzeit des Algorithmus bei einem Drittel vorangemeldeter Anfragen deutlich verschlechtert, aber die Effizienz der Fahrpläne steigt.

Außerdem erkennen wir, dass die Einschränkung von Barth et al. [BRS25], auf Routen mit vielen Umstiegen zu verzichten, in den getesteten Instanzen sinnvoll ist. Weiterhin konnten wir beobachten, dass sich die Startzeiten der Anfragen zwischen Registrierungszeit und Startzeit nicht verändern, obwohl wir dies nicht explizit festlegen, was wir auf die Umsetzung des MILPs zurückführen. Abschließend erkennen wir, dass ein liDARPT-System vor dem Aspekt der Effizienz vor allem in urbanen Regionen Sinn macht. In der Praxis wird ein solches System dort jedoch seltener benötigt und führt häufiger zu sehr hohen Laufzeiten, was die Umsetzung erschwert.

Wir können jedoch bestätigen, dass das dynamische liDARPT eine hohe Skalierbarkeit für Instanzen mit hoher Anfragendichte bietet. Es hat somit das Potenzial, eine noch größere Effizienz als das statische liDARPT zu erreichen.

Anhang

A. Variablenübersicht

Zustand	Beschreibung
$\mathcal{A}(t)$	Aktive Anfragen zum Zeitpunkt t
$\mathcal{N}(t)$	Neue Aufträge zum Zeitpunkt t
$\mathcal{S}(t)$	Angenommene, noch nicht abgeholte Aufträge zum Zeitpunkt t
$\mathcal{P}(t)$	Abgeholte Aufträge zum Zeitpunkt t
$\mathcal{D}(t)$	Abgelieferte Aufträge zum Zeitpunkt t
$\mathcal{R}(t)$	Abgelehnte Aufträge zum Zeitpunkt t

Abb. A.1.: Übersicht über die Zustände der Transportaufträge

Variable	Beschreibung
0_i	Depot der Linie i
$A(t)$	Kanten zum Zeitpunkt t
$A_{\mathcal{A}}(t)$	Aktive Kanten zum Zeitpunkt t
$A_{\mathcal{R}}(t)$	Realisierte Kanten zum Zeitpunkt t
$A^{\text{realized}}(t)$	Realisierte Kanten zum Zeitpunkt t
b	Dauer der Haltestopps
B_ρ	Zeitpunkt, zu dem die Aktione ρ ausgeführt wird
$c(r)$	Anzahl der Personen in der Anfrage r
d	Fahrtrichtung auf einem Streckenabschnitt
$d(\psi/\rho)$	Die Fahrtrichtung eines Streckenabschitts oder einer Aktion
$e(x)$	Beginn des Zeitfensters für Anfrage, Aktion, Streckenabschnitt oder Event x
$\mathcal{I}$	Linien des Netzwerks
$i(\rho/\psi)$	Die Linie eines Streckenabschnitts oder einer Aktion
k	Fahrzeug
$\mathcal{K}$	Menge der Fahrzeuge
$\mathcal{K}_i$	Menge der Fahrzeuge auf Linie i
$l(x)$	Ende des Zeitfensters für Anfrage, Aktion, Streckenabschnitt oder Event x
$\mathcal{L}_r$	Maximale Reisezeit für die Transportanfrage r
N	Busnetzwerk
N_l	Liniennetzwerk
$\rho^{i+/-}_{s,r,d}$	Aktion von r an Haltestelle s in Richtung d
P	Menge aller Aktionen
$\mathrm{P}(r)$	Alle Aktionen für Transportanfrage r
$\mathrm{P}(r)_j$	Aktionen der Route ϕ^r_j
$\mathrm{P}(v)$	Aktion im Event v
q_r	Akzeptanzvariable für Transportanfrage r
Q_i	Kapazität der Fahrzeuge auf Linie i
$r \in R$	Transportanfragen
$r(\rho/\psi)$	Die Transportanfrage eines Streckenabschnitts oder einer Aktion
r^+	Startpunkt der Transportanfrage r
r^-	Endpunkt der Transportanfrage r
R	Menge aller Transportanfragen
$R(v)$	Anfragen im Event v
$s(\psi/\rho)$	Die Haltestelle eines Streckenabschnitts oder einer Aktion
S	Menge aller Haltestellen
S^T	Menge der Umsteigepunkte
S_i	Menge der Haltestellen der Linie i
$s_1 \xrightarrow[d]{s} s_2$	Haltestelle s liegt zwischen s_1 und s_2 bei Fahrtrichtung d
$T^r_\Phi(t)$	Baum der offenen Routen von Transportanfrage r zum Zeit t
$t_{\min}(r)$	Kürzeste Reisedauer zwischen Start- und Zielpunkt der Transportanfrage r
$w(s_i, s_j)$	Zeit, die ein Fahrzeug mindestens von Haltestelle s_i zu s_2 braucht
$w_{\max}(s_i, s_j)$	Zeit, die ein Fahrzeug maximal von Haltestelle s_i zu s_2 braucht

Abb. A.2.: Übersicht über die verwendeten Variablen und Funktionen 1

Variable	Beschreibung
ν	Knoten im Routenbaum
v	Knoten im Event Based Graph
V_0	Menge der Idle-Events
$V(t)$	Knoten zum Zeitpunkt t
$V_{\mathcal{A}}(t)$	Aktive Knoten zum Zeitpunkt t
$V_{\mathcal{R}}(t)$	Realisierte Knoten zum Zeitpunkt t
$V^{\text{realized}}(t)$	Realisierte Knoten zum Zeitpunkt t
$V^{\text{1-realized}}(t)$	Zuletzt realisierter Knoten zum Zeitpunkt t
$V^{\text{realized}}(t)$	Realisierte Knoten
$w(s_1, s_2)$	Kürzeste Reisezeit zwischen den Haltestellen s_1 und s_2 in Sekunden
$w_{\max}(s_1, s_2)$	längste Reisezeit zwischen den Haltestellen s_1 und s_2 in Sekunden
$w_{\min}(r)$	Kürzeste Strecke zwischen Start- und Endpunkt der Anfrage r
x	Gewählte Kanten im MILP
x_a	Einplanung der Kante a
$\chi(t)$	Lösung des liDARPT zum Zeitpunkt t
$\delta^{\text{in}}(v, t)$	Eingehende Kanten des Events v zur Zeit t
$\delta^{\text{out}}(v, t)$	Ausgehende Kanten des Events v zur Zeit t
Δ	Maximale Rechenzeit des MILPs
λ_i	Haltestellen der Linie i
ϕ_j^r	Route j von Anfrage r
Φ^r	Menge aller möglichen Routen von Anfrage r
Φ_c^r	Geschlossene Routen von r
Φ_o^r	Offene Routen von r
$\psi^i_{s_1,s_2,r,d}$	Streckenabschnitt von r auf Linie i zwischen s_1 und s_2 in Richtung d
Ψ_ν	Kanten im Subbaum des Routenknotens ν
$\Psi_{\text{in}}(v)$	Streckenabschnitte im Fahrzeug vor der Ausführung von Event v
$\Psi_{\text{out}}(v)$	Streckenabschnitte im Fahrzeug nach der Ausführung von Event v
$\Psi(v)$	Streckenabschnitte im Event v
Ψ_j^r	sei die Menge aller Streckenabschnitte der Route j von Anfrage r
$\sigma(k)$	Linie des Fahrzeugs k

Abb. A.3.: Übersicht über die verwendeten Variablen und Funktionen 2

B. Abbildungen der Netzwerke

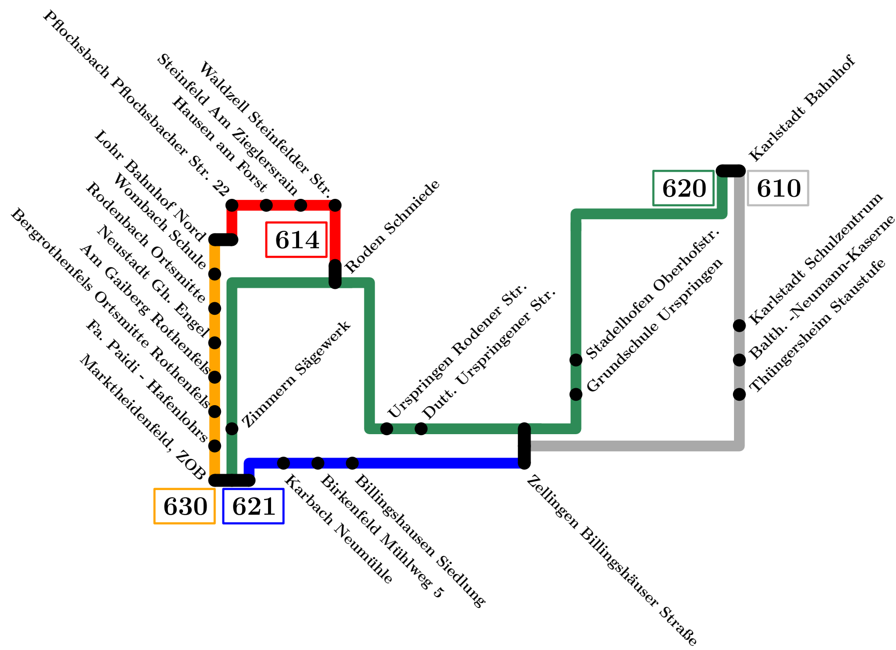


Abb. B.1.: Schematische Darstellung des Netzwerks *Markt-Karl-Lohr*

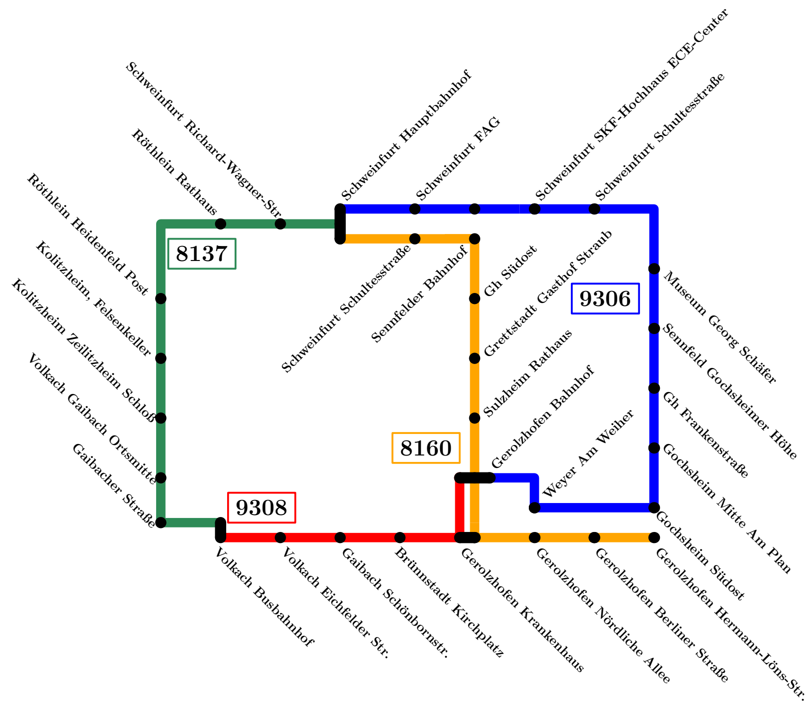


Abb. B.2.: Schematische Darstellung des Netzwerks *SW-Geo-Full*

Niederwerrn Geldersheim Flugplatz
Niederwerrn Schweinfurter Straße
Niederwerrn Brühlweg
Stadion
Nikolaus-Hofmann-Straße
Theater
Schweinfurt SKF Werk 2
Friedrichstraße
Schweinfurt Hauptbahnhof
Schweinfurt Gericht
8139
Werneck Schnackenwerth
Containerterminal
Thws
8148
Schweinfurt Kunsthalle
8134
Arnstein ZOB
Mühlhausen Am Sportheim
Werneck Eckartshausen Feuerwehr
Werneck Julius-Echter-Straße
Schweinfurt Ignaz-Schön-Str.
Schweinfurt ZF-Werk Süd

Abb. B.3.: Schematische Darstellung des Netzwerks *SW-Schlee-3*

Europastern
Bayernstraße
Gustav-Walle-Schule
Straubberg
Steinlein
Bonhoefferstraße
Zum Tännig
Kühlenbergstraße
Brunnenstraße
12
Sozialgericht
Hauptfriedhof
Siligmüllerstraße
Missioklinik
Campusbrücke
Mathematisches Institut
Sandbühl
Gerbrunn Rathaus
Am Happach
Paul Ehrlich Straße
Renzig
Casteller Platz
14
Busbahnhof
Rennweg
Valentin-Becker-Straße
Seinsheimstraße
Wittelsbacherplatz
Letzter Hieb
114
Universitätsbibliothek
Universität Sportzentrum

Abb. B.4.: Schematische Darstellung des Netzwerks *Wue-Simplified*

Literaturverzeichnis

[BCJ14] Kris Braekers, An Caris und Gerrit K. Janssens: Exact and meta-heuristic approach for a general heterogeneous dial-a-ride problem with multiple depots. *Transportation Research Part B: Methodological*, 67:166–186, 2014, https://doi.org/10.1016/j.trb.2014.05.007, ISSN 0191-2615. https://www.sciencedirect.com/science/article/pii/S0191261514000800.

[BRS25] Jonas Barth, Kendra Reiter und Marie Schmidt: The Line-Based Dial-a-Ride Problem with Transfers. In: Jonas Sauer und Marie Schmidt (Herausgeber): *25th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2025)*, Band 137 der Reihe *Open Access Series in Informatics (OASIcs)*, Seiten 17:1–17:20, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, ISBN 978-3-95977-404-8, 10.4230/OASIcs.ATMOS.2025.17. https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.ATMOS.2025.17.

[Buh21] Eberhard Buhl: *Urbane Mobilität im Wandel*, Seiten 103–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2021, ISBN 978-3-662-61352-8, 10.1007/978-3-662-61352-8$_9$. https://doi.org/10.1007/978-3-662-61352-8_9.

[CL07] Jean Francois Cordeau und Gilbert Laporte: The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, September 2007, 10.1007/s10479-007-0170-8. https://ideas.repec.org/a/spr/annopr/v153y2007i1p29-4610.1007-s10479-007-0170-8.html.

[CLPS07] Jean François Cordeau, Gilbert Laporte, Jean Yves Potvin und Martin W.P. Savelsbergh: Chapter 7 Transportation on Demand. In: Cynthia Barnhart und Gilbert Laporte (Herausgeber): *Transportation*, Band 14 der Reihe *Handbooks in Operations Research and Management Science*, Seiten 429–466. Elsevier, 2007, https://doi.org/10.1016/S0927-0507(06)14007-4. https://www.sciencedirect.com/science/article/pii/S0927050706140074.

[Deu25] V. Deutsch: Linien- und Bedarfsverkehre in der Region: Integriert, datenbasiert, effizient. 2025. https://zukunftsnetz-mobilitaet.nrw.de/media/2025/6/11/02ffc59704712045535d20043301d706/vdv-linien-und-bedarfsverkehr-leitfaden-on-demand-05-2025.pdf?v=1749637905.

[GKP+25] Daniela Gaul, Kathrin Klamroth, Christian Pfeiffer, Michael Stiglmayr und Arne Schulz: A tight formulation for the dial-a-ride problem.

European Journal of Operational Research, 321(2):363–382, 2025, https://doi.org/10.1016/j.ejor.2024.09.028, ISSN 0377-2217. https://www.sciencedirect.com/science/article/pii/S0377221724007318.

[GKS21] Daniela Gaul, Kathrin Klamroth und Michael Stiglmayr: Solving the Dynamic Dial-a-Ride Problem Using a Rolling-Horizon Event-Based Graph. Januar 2021, 10.4230/OASIcs.ATMOS.2021.8.

[GKS22] Daniela Gaul, Kathrin Klamroth und Michael Stiglmayr: Event-based MILP models for ridepooling applications. *European Journal of Operational Research*, 301(3):1048–1063, None 2022, 10.1016/j.ejor.2021.11.053. https://ideas.repec.org/a/eee/ejores/v301y2022i3p1048-1063.html.

[GN23] K. Gkiotsalitis und A. Nikolopoulou: The multi-vehicle dial-a-ride problem with interchange and perceived passenger travel times. *Transportation Research Part C: Emerging Technologies*, 156:104353, 2023, https://doi.org/10.1016/j.trc.2023.104353, ISSN 0968-090X. https://www.sciencedirect.com/science/article/pii/S0968090X23003431.

[HSK+18] Sin C. Ho, W.Y. Szeto, Yong Hong Kuo, Janny M.Y. Leung, Matthew Petering und Terence W.H. Tou: A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological*, 111:395–421, 2018, https://doi.org/10.1016/j.trb.2018.02.001, ISSN 0191-2615. https://www.sciencedirect.com/science/article/pii/S0191261517304484.

[NBM21] Sonia Nasri, Hend Bouziri und Wassila Mtalaa: *Customer-Oriented Dial-A-Ride Problems: A Survey on Relevant Variants, Solution Approaches and Applications*, Seiten 111–119. Januar 2021, ISBN 978-3-030-53439-4, 10.1007/978-3-030-53440-$0_1$3.

[Psa80] Harilaos N. Psaraftis: A Dynamic Programming Solution to the Single Vehicle Many-to-Many Immediate Request Dial-a-Ride Problem. *Transportation Science*, 14(2):130–154, Mai 1980, 10.1287/trsc.14.2.130, ISSN 1526-5447. https://doi.org/10.1287/trsc.14.2.130.

[RDS06] Brahim Rekiek, Alain Delchambre und Hussain Saleh: Handicapped Person Transportation: An application of the Grouping Genetic Algorithm. *Engineering Applications of Artificial Intelligence*, 19:511–520, August 2006, 10.1016/j.engappai.2005.12.013.

[RP06] Stefan Ropke und David Pisinger: An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, 40:455–472, November 2006, 10.1287/trsc.1050.0135.

[Sav85] Martin Savelsbergh: Local search in routing problems with time windows. *Annals of Operations Research*, 4:285–305, Dezember 1985, 10.1007/BF02022044.

[SD21] Carsten Sommer und Volker Deutsch: *Grundlagen und Formen des ÖPNV*, Seiten 207–253. Springer Berlin Heidelberg, Berlin, Heidelberg, 2021, ISBN 978-3-662-59697-5, 10.1007/978-3-662-59697-5$_5$. https://doi.org/10.1007/978-3-662-59697-5_5.

[WH06] K. I. Wong und Bell. G. H.: The Vehicle Routing Problem with Time Windows: Minimizing Route Duration. *International Transactions in Operational Research*, 13:195–208, Mai 2006, 10.1111/j.1475-3995.2006.00544.x.

[XCC08] Zhihai Xiang, Chengbin Chu und Haoxun Chen: The study of a dynamic dial-a-ride problem under time-dependent and stochastic environments. *European Journal of Operational Research*, 185(2):534–551, 2008, https://doi.org/10.1016/j.ejor.2007.01.007, ISSN 0377-2217. https://www.sciencedirect.com/science/article/pii/S0377221707000926.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur selbständigen Leistungserbringung

Titel der Masterarbeit:
Das dynamische Dial-a-Ride Problem mit Umstiegen

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):
Prof. Dr. Marie Schmidt, Lehrstuhl für Informatik I

Eingereicht durch (Vorname, Nachname, Matrikel):
Sebastian Körner, 2597126

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

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

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

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

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

Erbach, 31.01.2026

Ort, Datum, Unterschrift

Ref. 2.3 / Stand: 24.11.2023