

Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Das Treffpunktproblem im öffentlichen Personenverkehr

Jurek Sander

Studiengang: Informatik

Prüfer/in: Prof. Dr. Stefan Funke

Betreuer/in: Dr. rer. nat. Claudius Proissl

Beginn am: 27. Februar 2024

Beendet am: 31. Juli 2024

Kurzfassung

Diese Masterarbeit beschäftigt sich mit der Lösung des Treffpunkt-Problems in öffentlichen Personenverkehrsnetzen. Wir suchen dabei ausgehend von einer gegebenen Menge an n Ausgangshaltestellen und einer Startzeit τ_s nach dem optimalen Treffpunkt gemäß zweier Optimalitätskriterien. Das minimale Summe (MinSum) Kriterium sucht nach dem Treffpunkt mit der minimalen aufsummierten Fahrtzeit aller Beteiligter und das minimale Maximum (MinMax) Optimalitätskriterium nach dem Treffpunkt, an dem das Treffen frühestmöglich stattfinden kann. Für dieses Problem sind uns keine Algorithmen für öffentliche Personenverkehrsnetze bekannt. Als Grundlage für unsere Arbeit dient der Round-Based Public Transit Routing (RAPTOR) Algorithmus von Delling et al. (2012) in [DPW12], der zur Berechnung von schnellsten Reisen in öffentlichen Verkehrsnetzen genutzt werden kann. Ausgehend von diesem entwickeln wir zwei neue Algorithmen, die über die Verwendung von Prioritätenfunktionen und Heuristiken zielgerichtete Berechnungen ermöglichen. Als Heuristik führen wir eine Methode ein, die diese über Landmarken in öffentlichen Verkehrsnetzen berechnet. Ausgehend von diesen lässt sich der Suchraum des RAPTOR Algorithmus so einschränken, dass wir mit zwei neu entwickelten Algorithmen effizientere Berechnungen durchführen können. Diese Algorithmen nutzen wir, um das Treffpunkt-Problem für kleine Anzahlen an Ausgangshaltestellen optimal effizient lösen zu können. Für größere Anzahlen an Ausgangstopps entwerfen wir eine iterative Herangehensweise zur Lösung des Treffpunkt-Problems, die uns den frühestmöglichen Treffpunkt liefert. Der Vorteil des Algorithmus ist dabei, dass die iterative Auswahl einer kleinen Teilmenge der gegebenen Ausgangshaltestellen ausreicht, um den optimalen Treffpunkt zu finden und zu validieren. Mit diesem Algorithmus lässt sich das Problem selbst für sehr große Anzahlen an Ausgangstopps effizient lösen. Wir analysieren die Ausführungszeiten und Eigenschaften der neu entwickelten Algorithmen auf dem kompletten Netz des öffentlichen Personenverkehrs in Deutschland.

Inhaltsverzeichnis

1	Motivation	15
2	Grundlagen	17
2.1	Definitionen	17
2.2	Problembeschreibungen	20
2.3	Datensatz	21
3	Verwandte Arbeiten	25
3.1	Treffpunkt-Problem in Straßennetzwerken	25
3.2	Prioritätsfunktionen und Heuristiken	26
3.3	Unterschiede zum öffentlichen Personenverkehr	29
3.4	Berechnung frühester Ankunftszeiten in öffentlichen Verkehrsnetzwerken . .	30
4	Bestehende Algorithmen	31
4.1	RAPTOR	31
4.2	Untere Schranken über zeitunabhängige Graphen	33
5	Neue Algorithmen	37
5.1	Untere Schranken über Landmarken	37
5.2	Früheste Ankunftszeit	38
5.3	Treffpunkt-Problem	45
6	Experimente	55
6.1	Untere Schranken	55
6.2	Früheste Ankunftszeit	57
6.3	Treffpunkt-Problem	58
7	Fazit	71
	Literaturverzeichnis	73

Abbildungsverzeichnis

2.1	Beispiel Verkehrsnetz	18
2.2	Beispiel Reise	20
6.1	Ausführungszeiten vollständige RAPTOR Algorithmen	61
6.2	Anzahl expandierter Routen der vollständigen RAPTOR Algorithmen	63
6.3	Ausführungszeiten der selektiven RAPTOR Algorithmen	67

Tabellenverzeichnis

6.1	Absolute Differenzen zu unteren Schranken	56
6.2	Relative Differenzen zu untere Schranken	56
6.3	Ausführungszeiten Earliest Arrival Time (EAT)	57
6.4	Anzahl expandierte Routen EAT	58
6.5	Kostenfunktion der optimalen Lösung	59
6.6	Ausführungszeiten Treffpunktsuche MinSum	60
6.7	Ausführungszeiten Treffpunktsuche MinMax	60
6.8	Expandierte Routen Treffpunktsuche MinSum	62
6.9	Expandierte Routen Treffpunktsuche MinMax	63
6.10	Genauigkeit Round-Based Public Transit Routing First (RAPTOR1)	64
6.11	Genauigkeit RAPTOR1 mit Abweichung $\leq 10\%$	65
6.12	Ausführungszeiten selektive Treffpunktsuche MinSum	66
6.13	Ausführungszeiten selektive Treffpunktsuche MinMax	67
6.14	Genauigkeit der selektiven Treffpunktsuche MinSum	68
6.15	Genauigkeit der selektiven Treffpunktsuche MinMax	68
6.16	Anzahl genaue Berechnungen von Round-Based Public Transit Routing Selection (RAPTOR SELECT) LOOP für ≤ 150 #Ausgangsstopps	69
6.17	Ausführungszeiten von RAPTOR SELECT LOOP für ≥ 500 #Ausgangsstopps	70
6.18	Anzahl genaue Berechnungen von RAPTOR SELECT LOOP für ≥ 500 #Ausgangsstopps	70

Verzeichnis der Algorithmen

5.1	RAPTOR PQ*	43
5.2	RAPTOR B	49

Abkürzungsverzeichnis

CSA	Connection Scan Algorithm.	30
EAT	Earliest Arrival Time.	9
GTFS	Google Transit Feed Specification.	21
LDT	Latest Departure Time.	21
MinMax	minimales Maximum.	3
MinSum	minimale Summe.	3
PHAST	Parallel hardware-accelerated shortest path trees.	33
RAPTOR	Round-Based Public Transit Routing.	3
RAPTOR PQ	Round-Based Public Transit Routing Priority Queue.	39
RAPTOR B	Round-Based Public Transit Routing Bound.	39
RAPTOR SELECT	Round-Based Public Transit Routing Selection.	9
RAPTOR1	Round-Based Public Transit Routing First.	9

1 Motivation

Das Treffpunkt-Problem sucht für eine gegebene Menge an Agenten und ihren zugehörigen Ausgangspunkten nach ihrem optimalen Treffpunkt gemäß unterschiedlicher Optimalitätskriterien. Die Lösung dieses Problems findet unterschiedliche Anwendungsfälle. So kann es für Personengruppen, die sich in unterschiedlichen Teilen einer Region oder eines Landes befinden, bei der Planung ihres nächsten Wiedersehens von entscheidender Bedeutung sein, dass sie insgesamt möglichst kurz fahren müssen, um diesen Treffpunkt zu erreichen. Ein weiteres mögliches Optimalitätskriterium, das auch in der Praxis seine Anwendung findet, ist die Suche nach dem Treffpunkt, an dem sie sich am schnellsten treffen können. Die Lösung des Treffpunktproblems kann zudem im Bereich der Städteplanung Vorteile bieten. So sollten Gebäude, die oft von vielen Menschen aus unterschiedlichen Richtungen erreicht werden müssen, so gebaut werden, dass diese Personen möglichst kurze Anfahrtswege besitzen. Beispiele für solche Bauten sind Einkaufszentren oder Stadien, die ein großes Einzugsgebiet besitzen und gut erreichbar sein sollten. Über die Berechnung von optimalen Treffpunkten von Personen aus diesen Gebieten lassen sich ideale Standorte ermitteln.

Für Straßennetzwerke existieren bereits mehrere Algorithmen, um das Treffpunkt-Problem effizient zu lösen. In Hinblick auf die angestrebte Klimaneutralität und die zunehmende Bedeutung der öffentlichen Verkehrsmittel im Vergleich zum Auto, ist es jedoch von Interesse, das Problem auch für öffentliche Personennetzwerke zu lösen. So wäre es dann möglich, einerseits optimale Treffpunkte von Personengruppen, die mit öffentlichen Verkehrsmitteln reisen, zu bestimmen und andererseits könnten ideale Standorte in der Stadtplanung auch unter Berücksichtigung dieser geplant werden. Dies ist gerade bei dem Beispiel der Stadien von Vorteil, da Menschen zu Großereignissen immer häufiger bevorzugt mit Alternativen zum Auto anreisen. Da öffentliche Personenverkehrsnetze sich unter anderem über die Zeitabhängigkeit von Straßennetzwerken unterscheiden, können deren Algorithmen nicht angewendet werden. Daher besteht Forschungsbedarf hinsichtlich der Entwicklung von Algorithmen zur Lösung des Treffpunkt-Problems in öffentlichen Personennetzwerken.

In unserer Arbeit gehen wir dabei so vor, dass wir zunächst benötigte Definitionen einführen und anschließend die Grundkonzepte der bestehenden Algorithmen zur Lösung des Treffpunkt-Problems im Straßenverkehr beschreiben. Entscheidend dabei sind Heuristiken und Prioritätsfunktionen, da mit diesen der Suchraum an möglichen Treffpunkten eingeschränkt werden kann. Wir führen dazu für die öffentlichen Verkehrsnetze zwei unterschiedliche Ansätze ein, mit denen untere Schranken für die minimale Reisedauer zwischen zwei Haltestellen des Netzwerkes berechnet werden können. Diese verwenden wir anschließend als Grundlage für die

Prioritätenfunktionen. Um diese zur Lösung des Treffpunkt-Problems auch nutzen zu können, müssen wir zunächst die Basis-Algorithmen zur Routenberechnung in öffentlichen Verkehrsnetzen so anpassen, dass sie unter Verwendung der Heuristiken zielgerichtet den Suchraum einschränken können. Daher betrachten wir in dieser Arbeit jeweils zunächst das Problem der frühesten Ankunftszeit. Dabei suchen wir nach der Reise mit der minimalen Fahrtzeit zwischen zwei gegebenen Haltestellen und einem Startzeitpunkt. Die dafür neu entworfenen Algorithmen und darin enthaltenen Heuristiken benötigen wir in leicht abgewandelter Form als Bestandteil der Herangehensweisen zur Lösung des Treffpunkt-Problems. Die Beschreibung dieser Algorithmen stellt den Hauptteil unserer Arbeit dar. Für eine kleine Anzahl an Ausgangshaltestellen lösen wir das Treffpunktproblem optimal mit Algorithmen, die durch Prioritätenfunktionen und Heuristiken den Suchraum einschränken. Für sehr große Anzahlen an Ausgangshaltestellen haben wir einen einfachen iterativen Algorithmus entwickelt, mit dem das Treffpunkt-Problem effizient gelöst werden kann. Abschließend führen wir Experimente für die neu entworfenen Algorithmen beider betrachteter Probleme durch und analysieren dabei unter anderem den Einfluss der Heuristiken.

2 Grundlagen

Wir benötigen zunächst einige Definitionen, um anschließend die Probleme zu definieren, die wir mit unseren Algorithmen lösen. Abschließend beschreiben wir den Datensatz und die Anpassungen dessen, damit wir die Algorithmen ausführen können.

2.1 Definitionen

Wir definieren zunächst allgemein Graphen und anschließend die Bestandteile eines öffentlichen Personenverkehrsnetzes.

2.1.1 Graphen

Definition 2.1.1

Ein Graph $G(V, E, c)$ besteht aus einer Knotenmenge V und Kantenmenge E , wobei für alle $e = (u, v) \in E$ gilt, dass u und v in V sind. Die Kostenfunktion $c : E \rightarrow \mathbb{N}$ definiert die Kantengewichte.

Definition 2.1.2

Ein Pfad π im Graph ist eine Folge an Knoten $v_1, v_2, \dots, v_n$ mit $(v_i, v_{i+1}) \in E$ für $1 \leq i \leq n - 1$. Die Länge des Pfades ist über die Summe der Kosten der Kanten des Pfades definiert: $C(\pi) = \sum_{i=1}^{n-1} c((v_i, v_{i+1}))$. Die minimale Distanz zwischen zwei Knoten u und v entspricht der Länge des kürzesten Pfades zwischen diesen beiden Knoten.

2.1.2 Öffentliche Verkehrsnetze

Wir verwenden die nachfolgenden Definitionen analog zu Delling et al. (2015) in [DPW12] und Sander (2022) in [San24].

Definition 2.1.3

Passagiere können an Haltestellen innerhalb des Verkehrsnetzes in Verkehrsmittel einsteigen oder zwischen ihnen umsteigen. Diese Punkte werden als Stopps (Stops) bezeichnet. Alle Stopps eines Verkehrsnetzes sind in der Menge S gespeichert.

Definition 2.1.4

Die Verkehrsmittel fahren auf fest definierten Routen (Routes) R durch das Verkehrsnetz. Eine Route stellt dabei eine Teilmenge der Stopps S dar. Für diese ist innerhalb der Route eine feste Reihenfolge definiert, in der sie abgefahren werden.

In der Abbildung 2.1 wird ein beispielhaftes Verkehrsnetz dargestellt. Dabei repräsentieren die Punkte Stopps, zwischen denen die Routen r_1 bis r_4 verlaufen. Verkehrsmittel dieser Routen müssen die zugehörigen Stopps in der dargestellten Reihenfolge abfahren.

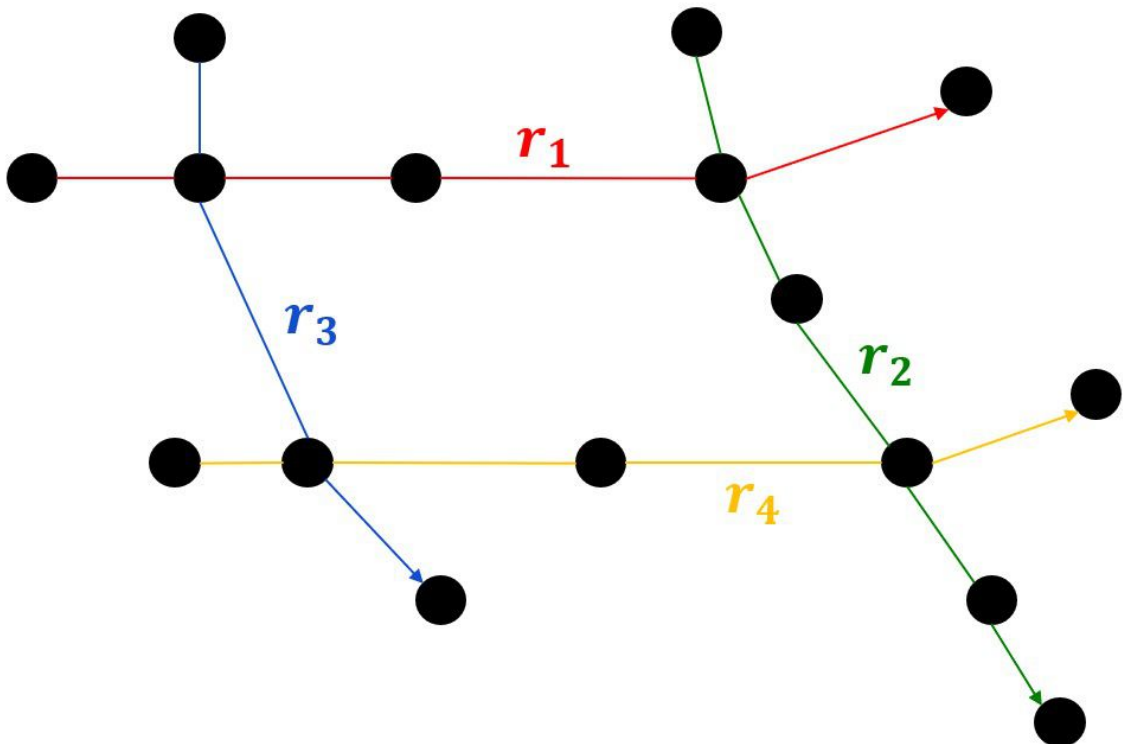


Abbildung 2.1: Ein beispielhaftes Verkehrsnetz. Stopps sind als Punkte dargestellt, zwischen denen die Routen r_1 bis r_4 verlaufen.

Definition 2.1.5

Für jede Route existieren Trips. Diese definieren die Ankunfts- und Abfahrtszeiten von Verkehrsmitteln an den Stopps der Route. Somit besitzt jeder Trip t' einer Route $r \in R$ an jedem Stopp p von r eine Ankunftszeit $\tau_{arr}(t', p)$ und Abfahrtszeit $\tau_{dep}(t', p)$. Für diese gilt $\tau_{arr}(t', p) \leq \tau_{dep}(t', p)$. Die Ankunftszeit des ersten Stopps einer Route und die Abfahrtszeit des letzten Stopps sind undefiniert. Für die Trips einer Route gilt zudem, dass sie sich gegenseitig nicht überholen dürfen. Alle Trips sind in der Menge T gespeichert.

Definition 2.1.6

Um Wege von Passagieren im Verkehrsnetz beschreiben zu können, benötigen wir Streckenabschnitte (Legs). Jeder Streckenabschnitt l ist einem Trip l_{trip} zugeordnet und definiert sich über

den Einstiegspunkt $l_{\text{dep_stop}}$ und Ausstiegspunkt $l_{\text{arr_stop}}$ des Passagiers und deren Zeiten innerhalb des Trips. Somit ist l über das Tupel $l = (l_{\text{dep_stop}}, l_{\text{arr_stop}}, l_{\text{dep_time}}, l_{\text{arr_time}}, l_{\text{trip}})$ gegeben.

Definition 2.1.7

Mögliche Fußwege von Passagieren werden über Fußwege (Footpaths) F definiert. Falls es möglich ist, von Stopp p zu Stopp q zu laufen, existiert der Fußweg $f \in F$ mit Ausgangsstopp $f_{\text{dep_stop}} = p$ und Zielstopp $f_{\text{arr_stop}} = q$. Die Dauer des Fußweges ist in f_{dur} gespeichert. Ein Fußweg f wird somit über das Tupel $f = (f_{\text{dep_stop}}, f_{\text{arr_stop}}, f_{\text{dur}})$ angegeben. Wir speichern zudem für jeden Stopp p einen Fußweg mit p als Ausgangs- und Ankunftsstopp und der minimalen Umstiegsdauer an diesem Stopp ab. So können wir festlegen, wie viel Zeit für einen Umstieg an diesem Stopp eingeplant werden muss. Die Menge $F(p)$ gibt für einen Stopp p alle Fußwege an, die von ihm ausgehen. Die Fußwege erfüllen die Eigenschaft der Transitivität. Falls die Stopps p_1 und p_2 indirekt über Fußwege verbunden sind, ist auch ein direkter Fußweg zwischen diesen Stopps in F enthalten.

Definition 2.1.8

Als Umstieg (Transfer) definieren wir den Wechsel zwischen zwei Verkehrsmitteln, die zu unterschiedlichen Route gehören. Ein Umstieg erfolgt dabei immer an Stopps und kann einen Fußweg zwischen dem Ausstiegstopp des ersten Verkehrsmittels und dem Einstiegstopp des zweiten beinhalten.

Definition 2.1.9

Die Reise (Journey) j eines Passagiers zwischen dem Startstopp s und dem Zielstopp t innerhalb des Verkehrsnetzes kann nun über die Sequenz an aufeinanderfolgenden Fußwegen und Streckenabschnitten $j = f^0, l^0, f^1, l^1, \dots, f^k, l^k, f^{k+1}$ mit $f_{\text{dep_stop}}^0 = s$ und $f_{\text{arr_stop}}^{k+1} = t$ definiert werden. Sie besitzt die Abfahrtszeit $j_{\text{dep_time}}$ an s und Ankunftszeit $j_{\text{arr_time}}$ an t . Die Dauer der Reise ist in j_{dur} gespeichert. Dabei muss für alle $i = 0, \dots, k - 1$ gelten:

$$\begin{aligned} f_{\text{arr_stop}}^i &= l_{\text{dep_stop}}^i \\ l_{\text{arr_stop}}^i &= f_{\text{dep_stop}}^{i+1} \\ l_{\text{arr_time}}^i + f_{\text{dur}}^{i+1} &\leq l_{\text{dep_time}}^{i+1} \end{aligned}$$

Dies gewährleistet, dass ein Umstieg zwischen den Trips zweier benachbarter Streckenabschnitte der Reise möglich ist. Der Umstieg kann einen Fußweg zwischen zwei Stopps beinhalten oder an einem Stopp erfolgen. Die Reise j besteht dabei aus $k + 1$ Trips, zwischen denen der Passagier k -Mal umsteigen muss. Jede Reise enthält mindestens einen Fußweg. Für Reisen können Mengen an Optimierungskriterien wie die minimale Fahrtzeit oder Anzahl an Umstiegen definiert werden.

In der Abbildung 2.2 kann eine mögliche Reise zwischen den Stopps s und t über die Streckenabschnitte l^0 bis l^2 angegeben werden. Die Streckenabschnitte werden dabei jeweils über einen Trip der zugehörigen Route und deren Ein- und Ausstiegstopps definiert. Dabei entspricht der Abfahrtsstopp von l^0 dem Startstopp s der Reise und der Ankunftsstopp von l^2 dem Zielstopp t . Die Fußwege f^0 bis f^3 dieser Reise besitzen jeweils den gleichen Ausgangs- und Zielstopp und stellen somit die minimale Umstiegsdauer dar.

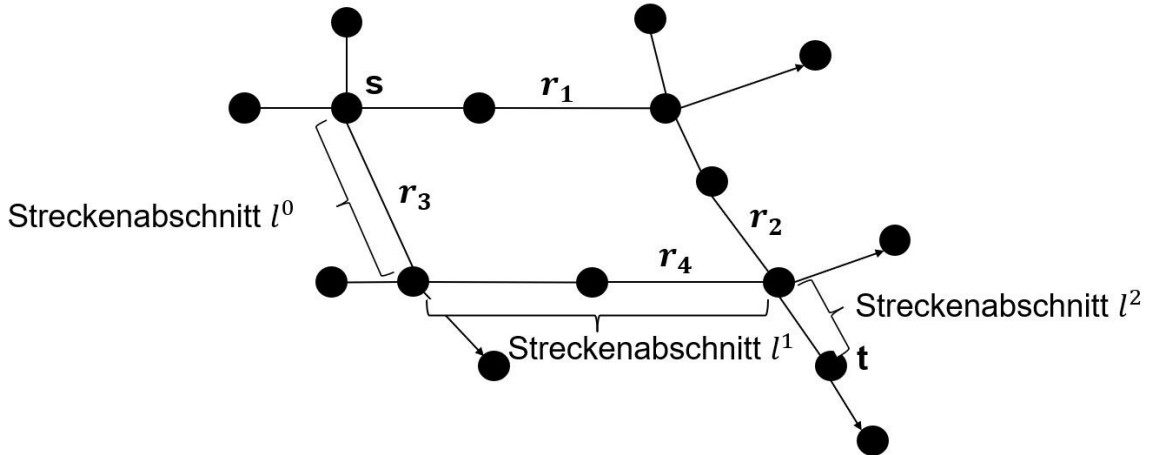


Abbildung 2.2: Eine mögliche Reise zwischen den Stopps s und t kann über die Streckenabschnitte l^0 bis l^2 angegeben werden.

Definition 2.1.10

Für ein gegebenes öffentliches Verkehrsnetz ist sein zeitunabhängiger Graph $G(V, E)$ über die Entfernung aller zeitabhängigen Informationen definiert. Dabei wird $V = S$ gesetzt und eine Kante $e = (u, v)$ erstellt, falls die Stopps u und v in einer Route direkt aufeinanderfolgen oder ein Fußweg zwischen diesen Stopps existiert. Die Kosten $c(e)$ der Kante entspricht dem Minimum der Fahrtzeit aller Trips zwischen diesen Stopps und der Dauer des Fußweges zwischen ihnen. Für einige der Algorithmen werden ungerichtete Graphen vorausgesetzt. Dies lässt sich für die zeitunabhängigen Graphen gewährleisten, indem fehlende Rückkanten hinzugefügt werden oder ihre Kosten auf das Minimum der Kosten von Vorwärts- und Rückwärtskante gesetzt wird.

Definition 2.1.11

Die minimale Fahrtzeit zwischen zwei Stopps u und v ohne Berücksichtigung der zeitabhängigen Faktoren bezeichnen wir mit $d_{zu}(u, v)$. Sie stellt die minimale Distanz im zeitunabhängigen Graphen dar. Die kürzeste Fahrtzeit inklusive der zeitabhängigen Bedingungen zu einer beliebigen Abfahrtszeit τ_s geben wir mit $d(u, v)$ an.

2.2 Problembeschreibungen

Wir beschäftigen uns mit zwei Problemstellungen und lösen diese mit neu entwickelten Algorithmen.

Analog zu Delling et al. (2015) in [DPW12] definieren wir das EAT Problem. Für einen gegebenen Startstopp $s \in S$, Zielstopp $t \in S$ und eine Startzeit τ_s wird dabei die minimale Ankunftszeit aller Reisen gesucht, die ausgehend von Stopp s nach τ_s zu Stopp t führen. Falls der Zielstopp t nicht gegeben ist, berechnen wir die minimale Ankunftszeit an allen Stopps in S .

Dieses Problem lässt sich in das Latest Departure Time (LDT) Problem umwandeln. Hier wird die späteste Abfahrtszeit an Startstopp s gesucht, sodass man vor der Zielzeit τ_t an Zielstopp t ankommt. Auch hier kann die späteste Abfahrtszeit an allen Stopps in S berechnet werden.

Wir definieren das Treffpunkt-Problem analog zu Atzmon et al. (2020) in [ALF+20] für öffentliche Personen- statt Straßennetzwerke. Gegeben sind dabei n Ausgangsstopps $N = \{s_1, s_2, \dots, s_n\} \subseteq S$ und eine Startzeit τ_s . Wir suchen den optimalen Treffpunkt $t \in S$ und die n Reisen der Agenten ausgehend von ihren n Startstopps zu t . Wir definieren die zwei Optimalitätskriterien MinSum und MinMax. Die Kosten für einen Treffpunkt t nach diesen beiden Kriterien lassen sich mit den minimalen Fahrtdauern $j_{\text{dur}}(s_i, t)$ von $s_i \in S$ nach $t \in S$ mit der Startzeit τ_s wie folgt definieren:

$$C_{\text{MinSum}}(t) = \sum_{s_i \in N} j_{\text{dur}}(s_i, t)$$

$$C_{\text{MinMax}}(t) = \max_{s_i \in N} j_{\text{dur}}(s_i, t)$$

Für MinSum suchen wir somit nach dem Treffpunkt mit der minimalen Summe der Fahrtdauern von allen n Reisen. MinMax ermittelt den Treffpunkt mit der minimalen längsten Anreise und stellt somit den frühesten Zeitpunkt dar, zu dem sich die n Agenten treffen können. Der gesuchte Treffpunkt ist der Treffpunkt t^* mit der minimalen Kostenfunktion nach dem jeweiligen Optimalitätskriterium. Der optimale Treffpunkt lässt sich zudem über die minimale Anzahl an Umstiegen definieren, die die Agenten auf ihren Reisen vornehmen müssen.

2.3 Datensatz

Die Daten von öffentlichen Personenverkehrsnetzen liegen im Google Transit Feed Specification (GTFS) Format vor. Diese importieren wir und modifizieren sie so, dass sie den Definitionen aus Kapitel 2.1 entsprechen und von unseren Algorithmen genutzt werden können. Wir gehen dabei analog zu Sander (2022) in [San24] vor.

2.3.1 GTFS Format

Die Daten im GTFS Format liegen in Tabellenform vor. Entscheidend für unsere Algorithmen sind die Tabellen Stops, Routes, Trips, Stop_Times und Calendar.

In der Tabelle Stops ist für jeden Stopp eine ID, über die er referenziert werden kann, sein Name als String und seine Koordinaten gespeichert. Die Tabelle Routes enthält die IDs der Routen, ihre Namen und Typen. Über die Tabelle Trips erfolgt die Zuweisung der Trips zu den Routen, da für jeden Trip dessen ID und die zugehörige Routen-ID gespeichert ist. Zusätzlich befindet sich in der Trip-Tabelle der Service ID Eintrag. Dieser verweist auf die Calendar Tabelle. Jeder Eintrag dieser Tabelle besitzt eine eindeutige Service-ID und gibt für die Tage

Montag bis Sonntag an, ob der Trip an diesen Tagen verfügbar ist. Dies ist entscheidend für unsere Algorithmen, da für sie die Information benötigt wird, an welchen Wochentagen Trips und somit ihre zugehörigen Verkehrsmittel genutzt werden können. Die Tabelle `Stop_Times` definiert die Abfahrtszeiten der Trips an den zugehörigen Stopps. Daher besitzt diese Tabelle die Felder `Arrival_Time` und `Departure_Time` zur Angabe der Zeiten. Die Felder `Stop_ID` und `Trip_ID` verweisen auf den zugehörigen Stopp und Trip. Der Eintrag `Sequence` gibt an, als wievielter Stopp der zugehörige Stopp innerhalb der Route angefahren wird.

Die Tabellen importieren wir jeweils so, dass alle Einträge in einem zugehörigen Vektor gespeichert werden. Die Position innerhalb des Vektors stimmt dabei jeweils mit der ID des zugehörigen Eintrages überein, um einen direkten Zugriff zu ermöglichen.

Zusätzlich sind in weiteren Tabellen der GTFS Dateien Informationen über die Agenturen der Verkehrsmittel des Netzes gespeichert. Diese haben jedoch für unsere Algorithmen keine weitere Bedeutung und müssen nicht importiert werden.

2.3.2 Anpassungen

Für die Algorithmen werden zusätzlich zu den Daten weitere Datenstrukturen benötigt, die aus den Tabellen des GTFS Formats generiert werden können. Diese sind notwendig, um die Algorithmen effizient zu gestalten und müssen einmalig nach dem Einlesen eines GTFS Datensatzes erstellt werden. Zudem passen wir die importierten Datenstrukturen so an, dass sie die Bedingungen unserer Definitionen aus dem Kapitel 2.1 erfüllen. Abschließend müssen wir aus den Informationen der Stopps die Fußwege des Verkehrsnetzes erstellen.

Zunächst ist für die Routen des GTFS Formats nicht gesichert, dass alle zugehörigen Trips die exakt gleichen Stopps in derselben Reihenfolge abfahren. Da dies durch unsere Definitionen vorgegeben ist und für die RAPTOR Algorithmen benötigt wird, müssen wir basierend auf den Trips neue Routen so erstellen, dass die Bedingung erfüllt ist. Zusätzlich füllen wir in dem gleichen Schritt vier Zeiger-Vektoren, die uns einen schnellen Zugriff auf die Daten ermöglichen. Hier speichern wir für jede Routen-ID die IDs der zugehörigen Stopps in der Reihenfolge, in der sie in der Route abgefahren werden. Andersherum benötigen wir zudem für jede Stopp-ID die IDs der Routen, in denen der Stopp enthalten ist. Für die Routen müssen wir zusätzlich alle IDs der Trips speichern, die zu der Route gehören. Zuletzt benötigen wir für jeden Trip die Position seines ersten Stoppzeit-Eintrages, um effizient auf alle seine Abfahrts- und Ankunftszeiten innerhalb des Vektors zugreifen zu können.

Damit wir die neuen Routen und die zusätzlichen Zeiger-Vektoren erstellen können, sortieren wir zunächst das Array der Stoppzeiten nach ihren Trips und innerhalb eines Trips nach der Sequenz seiner Stopps. Die Zeiger Vektoren initialisieren wir jeweils mit einem leeren Vektor. Anschließend iterieren wir einmal über den Stoppzeit-Vektor. Dabei speichern wir so lange die zugehörigen Stopps, bis sich die Trip-ID der betrachteten Einträge ändert. An diesem Punkt überprüfen wir, ob wir bereits eine neue Route mit der Reihenfolge dieser Stopps erstellt haben. Ist dies der Fall, fügen wir die Trip-ID in den Vektor, der die Trips einer Route

speichert, für diese Route hinzu. Andernfalls erstellen wir eine neue Route mit zugeordneter Route-ID. Dazu kopieren wir die Routen-Informationen der zugehörigen Route aus der GTFS Tabelle und ersetzen deren ID mit der neuen. Für diese speichern wir die Reihenfolge der zugehörigen Stopps und die Routen ID für jeden der Stopps. Zusätzlich speichern wir die Trip-ID für diese Route. Abschließend aktualisieren wir in beiden Fällen den Eintrag der Routen-ID in dem Vektor der Trips, die aus den GTFS Daten importiert wurden. Für den darauffolgenden ersten Stoppzeit-Eintrag des nächsten Trips speichern wir dessen Position innerhalb des Stoppzeiten-Vektors für den zugehörigen Trip.

Nach der Erstellung der neuen Routen, nutzen wir die Abfahrtszeiten der Trips, um den Zeiger Vektor, der alle Trips einer Route speichert, zu sortieren. Damit erreichen wir, dass alle Trips einer Route aufsteigend bezüglich der Abfahrtszeit der Trips an dem ersten Stopp gespeichert sind. Dies können wir somit in den Algorithmen als Annahme verwenden.

Wie in Kapitel 2.1 geschildert, benötigen wir für unsere Algorithmen Fußwege. Dazu iterieren wir über alle importierten Stopps des Netzwerks. Für jedes Paar an Stopps berechnen wir über die gegebenen Koordinaten ihre Luftliniendistanz und fügen einen Fußweg ein, falls diese kleiner als ein vorher festgelegter Wert ist. Wir setzen als Grenze dabei einen halben Kilometer und nehmen eine Gehgeschwindigkeit von 4km/h an, um die Dauer des Fußweges zu bestimmen. Um die Transitivität der Fußwege zu garantieren, iterieren wir so lange über die Fußwege und erstellen, falls benötigt neue, bis die Eigenschaft erfüllt ist. Wir speichern die Fußwege sortiert nach ihren Ausgangstopps und für jeden Stopp einen Zeiger auf seinen ersten Eintrag. Dies ermöglicht einen schnellen Zugriff zur Laufzeit.

3 Verwandte Arbeiten

Bevor wir das Treffpunkt-Problem in öffentlichen Verkehrsnetzen behandeln, vergleichen wir zunächst unterschiedliche Herangehensweisen, mit denen es in Straßennetzwerken gelöst wurde. Anschließend betrachten wir deren Heuristiken, die auch für die öffentlichen Personenverkehrsnetze relevant sind und die Unterschiede zwischen den Netzwerkvarianten. Abschließend gehen wir auf die Algorithmen zur Berechnung von minimalen Ankunftszeiten in öffentlichen Verkehrsnetzen ein.

3.1 Treffpunkt-Problem in Straßennetzwerken

Das Treffpunkt-Problem im Straßenverkehr wurde auf unterschiedliche Arten behandelt. Es ist dabei analog zu dem Problem im öffentlichen Personenverkehr, wie in Kapitel 2 beschrieben, definiert. Gegeben sind hierbei n Agenten, die sich zu Beginn jeweils auf einem Knoten des Straßenverkehrsnetzes befinden, das als Graph beschrieben ist. Gesucht ist der optimale Treffpunkt, der über unterschiedliche Kriterien definiert ist. Die Kosten der Reise eines der Agenten ist dabei über die Summe der Kantengewichte auf seinem Pfad gegeben.

Optimale Treffpunkte können naiv mit dem Dijkstra Algorithmus, der von Dijkstra (1959) in [Dij59] beschrieben wurde, berechnet werden. Dazu wird ausgehend von jedem der n Startpunkte ein Dijkstra Algorithmus durchgeführt, der die minimale Distanz zu allen Knoten des Straßennetzwerkes berechnet. Anschließend kann der optimale Treffpunkt mit einer Iteration über alle Stopps des Graphen ermittelt werden. Diese Methode liefert das optimale Ergebnis, aber ist ineffizient, da für jeden Agenten eine vollständige Berechnung der minimalen Distanzen durchgeführt werden muss.

Um die Berechnung der optimalen Treffpunkte effizienter zu gestalten, haben Lee et al. (2021) in [LP21] die G-Tree Methode eingeführt. Sie optimieren die Treffpunkte nach der minimalen Summe der Distanzen (MinSum) der Agenten und der minimalen maximalen Distanz (MinMax). Anschließend weicht die Definition der Kostenberechnung leicht von unserer für öffentliche Verkehrsnetze ab, da die Agenten nach dem Treffen an ihrem Treffpunkt gemeinsam zu einem gegebenen Knoten d im Netzwerk fahren. Dies lässt sich aber dennoch auf unsere Betrachtung der Problemdefinition anwenden. Die Berechnung der optimalen Treffpunkte führen Lee et al. über G-Trees durch. Diese Baumstruktur, die von Zhong et al. (2013) in [ZLTZ13] definiert wurde, basiert auf einer hierarchischen Partitionierung der Knoten des Graphen. In einer

einmaligen Vorberechnungsphase müssen zudem die Distanzen zwischen den Grenzknoten benachbarter Partitionen ermittelt und abgespeichert werden. Diese können bei der Berechnung der optimalen Treffpunkte zur Laufzeit genutzt werden. Dabei wird eine Branch-and-Bound Suche durchgeführt. Diese Bestensuche läuft über die Struktur des vorberechneten G-Trees und nutzt die Distanzen der Grenzknoten, um sie zu beschleunigen. Die einzelnen Distanzberechnungen während der Laufzeit des Algorithmus verwenden dynamisches Programmieren, um redundante Berechnungen zu vermeiden.

Atzmon et al. (2020) haben in [ALF+20] einen weiteren Algorithmus zur Berechnung der optimalen Treffpunkte vorgestellt. Ihr MM* Algorithmus ist ein multi-richtungsweisender heuristischer Suchalgorithmus, der den MM (Meet in the Middle) bidirektionalen Suchalgorithmus generalisiert. Der Algorithmus verwendet zwei unterschiedliche Kostenfunktionen, um analog zu unserer Definition Treffpunkte nach dem MinSum und MinMax Kriterium zu finden. Dabei wird für alle n Agenten eine Bestensuche durchgeführt, über die die Knoten des Straßennetzwerks expandiert werden. Die Suche erfolgt über eine gemeinsame Prioritätenwarteschlange für alle Agenten. Diese Warteschlange ist über die Prioritätenfunktion f sortiert, die eine untere Schranke für die Kostenfunktion des Optimalitätskriterium darstellt. Die Prioritätenwarteschlange ist dabei über zwei Listen OPEN und CLOSED definiert, in die die zu expandierenden und bereits expandierten Knoten jeweils zusammen mit einem Verweis auf ihren Agenten eingefügt werden. Die Kosten der aktuell besten Lösung werden während des Expandierens der Knoten aktualisiert, falls ein besserer Treffpunkt als der aktuell optimale gefunden wird. Der Algorithmus terminiert, sobald die untere Schranke f des kleinsten Wertes in der Prioritätenwarteschlange größer als die aktuell optimalen Kosten sind. Der Algorithmus garantiert, ein optimales Ergebnis zurückzugeben und ist hinsichtlich der Anzahl besuchter Knoten effizienter als die naive Herangehensweise über die Dijkstra Algorithmen, da die einzelnen Bestensuche zielgerichteter zum optimalen Treffpunkt führen.

3.2 Prioritätsfunktionen und Heuristiken

Zur Berechnung der optimalen Treffpunkte über den MM* Algorithmus von Atzmon et al. (2020) werden Prioritätsfunktionen und Heuristiken verwendet. Diese wurden ebenfalls in [ALF+20] beschrieben und ihre Korrektheit gezeigt.

Wir führen dazu die folgenden allgemeinen Notationen ein, die dem A* Algorithmus in [HNR68] entnommen sind. Die Heuristik $h(v)$ ist eine untere Grenze der optimalen Heuristik $h^*(v)$, die den optimalen Kosten von Knoten v zum Ziel t entspricht. Die optimalen Kosten sind dabei abhängig von der Optimalitätsfunktion. Für MinSum entsprechen sie der Summe der Fahrtzeiten zum optimalen Treffpunkt und für MinMax der maximalen Fahrtzeit der Agenten zu diesem. Seien $g(v)$ die bereits berechneten optimalen Kosten zum Knoten v . Dann ist die perfekte Prioritätsfunktion $f^*(v)$ über die Summe aus $g(v)$ und $h^*(v)$ definiert. Analog stellt $f(v) = g(v) + h(v)$ eine untere Schranke der Prioritätsfunktion $f^*(v)$ dar. Für die minimale Distanz $d(u, v)$ von Knoten u nach v definieren wir als untere Schranke die Heuristik $h_{\text{dur}}(u, v)$.

Im Kontext des öffentlichen Personenverkehrs gehen wir von Fahrtzeiten zwischen Stopps statt Distanzen aus.

Zur Berechnung der Prioritätenfunktion für das MinSum Optimalitätskriterium betrachten wir ein Paar (a_i, v) bestehend aus Agent a_i und Knoten v wie folgt. Die Kosten $f_{\text{MinSum}}^*(a_i, v)$ stellen die Kosten der optimalen Lösung dar, wobei der Agent a_i über v zu dem Treffpunkt t fährt. Die Kosten des Pfades von s_i zu v sind über $g(a_i, v)$ bereits berechnet. Die anderen Agenten a_j fahren in diesem Szenario von ihren Startpunkten s_j aus nach t . Da $f_{\text{MinSum}}^*(a_i, v)$ die Summe aus $g(a_i, v)$ und $h_{\text{MinSum}}^*(a_i, v)$ darstellt, ist $h_{\text{MinSum}}^*(a_i, v)$ die Summe der Kosten des Pfades von Agent a_i von v nach t und aller anderen Agenten a_j von s_j zu t . Dies ist somit wie folgt definiert:

$$h_{\text{MinSum}}^*(a_i, v) = \min_{t \in V} \left\{ d(v, t) + \sum_{a_j \in A \setminus \{a_i\}} d(s_j, t) \right\}.$$

Für eine untere Schranke der Heuristik $h_{\text{MinSum}}(a_i, v) \leq h_{\text{MinSum}}^*(a_i, v)$ ist die Prioritätenfunktion so definiert:

$$f_{\text{MinSum}}(a_i, v) = g(a_i, v) + h_{\text{MinSum}}(a_i, v).$$

Im Gegensatz dazu entspricht die Prioritätenfunktion für MinMax nicht der Summe der Fahrtkosten, sondern den Pfadkosten des Agenten mit der längsten Fahrt. Dabei stellt $f_{\text{MinSum}}^*(a_i, v)$ die optimalen Kosten dar, wenn der Agent a_i über v mit Kosten $g(a_i, v)$ zum Treffpunkt t fährt:

$$f_{\text{MinMax}}^*(a_i, v) = \min_{t \in V} \left[\max \left\{ \frac{g(a_i, v) + d(v, t)}{\max_{a_j \in A \setminus \{a_i\}} d(s_j, t)} \right\} \right].$$

Diese Prioritätenfunktion entspricht somit entweder der Pfadlänge des Agenten a_i über v oder der Distanz einer der anderen Agenten a_j zu t , falls diese größer ist. Für die untere Schranke der Prioritätenfunktion erhalten wir

$$f_{\text{MinMax}}(a_i, v) = \min_{t \in V} \left[\max \left\{ \frac{g(a_i, v)}{\frac{g(a_i, v) + h_{\text{MinSum}}(a_i, v)}{n}} \right\} \right]$$

mit n als Anzahl der Agenten. Die bereits berechnete Distanz $g(a_i, v)$ stellt eine untere Schranke von $f_{\text{MinMax}}^*(a_i, v)$ dar, da der Agent bereits über diese Strecke auf seinem Pfad zu t gereist ist. Einer der Agenten muss mindestens über die Distanz $f_{\text{MinSum}}^*(a_i, v)/n$ zum Treffpunkt t fahren. Da $f_{\text{MinSum}}(a_i, v)$ eine untere Schranke von $f_{\text{MinSum}}^*(a_i, v)$ darstellt, gilt $(g(a_i, v) + h_{\text{MinSum}}(a_i, v))/n \leq f_{\text{MinMax}}^*(a_i, v)$. Somit ist $f_{\text{MinMax}}(a_i, v)$ eine valide untere Schranke.

Die benötigten Heuristiken $h_{\text{MinSum}}(a_i, v)$ berechnen wir ebenfalls analog zu Atzmon et al. (2020) in [ALF+20] über die Clique Heuristik. Die anderen dort vorgestellten Heuristiken lassen sich nicht auf öffentliche Verkehrsnetze übertragen, da in diesen Graphen die Koordinaten der Stopps eine untergeordnete Rolle spielen. Dabei definieren wir $t^*(a_i, v)$ als den optimalen

Treffpunkt, wenn Agent a_i über den Knoten v fährt. Zur Einfachheit der Notation schreiben wir t^* statt $t^*(a_i, v)$. Wir definieren zudem die Menge der Startpunkte abgesehen des Ausgangsknotens von Agent a_i , der durch v ersetzt wurde, als $S_i(v) = (N \setminus \{s_i\}) \cup \{v\}$. Wir gehen davon aus, dass wir bereits valide untere Schranken $h_{\text{dur}}(v_1, v_2)$ für die minimalen Distanzen $d(v_1, v_2)$ zwischen allen Paaren (v_1, v_2) aus $S_i(v)$ berechnet haben.

Aus der Dreiecksungleichung folgt, dass für alle $v_1, v_2 \in S_i(v)$ (mit $v_1 \neq v_2$) folgendes gilt:

$$d(v_1, v_2) \leq d(v_1, t^*) + d(v_2, t^*).$$

Wir gehen dabei davon aus, dass der zugrunde liegende Graph G ungerichtet ist. Für zeitunabhängige Graphen des öffentlichen Verkehrsnetzes kann diese Eigenschaft gewährleistet werden.

Die Summe über alle diese Paare ergibt:

$$\sum_{\substack{\{v_1, v_2\} \in 2^{S_i(v)} \\ v_1 \neq v_2}} d(v_1, v_2) \leq [d(v_1, t^*) + d(v_2, t^*)].$$

Da jeder Knoten $v' \in S_i(v)$ mit $n - 1$ anderen Knoten in $S_i(v)$ gepaart wird, können wir die rechte Seite der Gleichung als $(n - 1) \cdot \sum_{v' \in S_i(v)} d(v', t^*)$ schreiben. Dies ergibt:

$$\sum_{\substack{\{v_1, v_2\} \in 2^{S_i(v)} \\ v_1 \neq v_2}} \frac{d(v_1, v_2)}{n - 1} \leq \sum_{v' \in S_i(v)} d(v', t^*) = h_{\text{MinSum}}^*(a_i, v).$$

Da $h_{\text{dur}}(v_1, v_2) \leq d(v_1, v_2)$ gilt, erhalten wir die folgende untere Schranke:

$$h_{\text{MinSum}}(a_i, v) = \sum_{\substack{\{v_1, v_2\} \in 2^{S_i(v)} \\ v_1 \neq v_2}} \frac{h_{\text{dur}}(v_1, v_2)}{n - 1} \leq h_{\text{MinSum}}^*(a_i, v).$$

Diese Heuristik kombiniert die Heuristiken der minimalen Distanzen zwischen allen Paaren der Ausgangspunkte und des aktuellen Knotens v von Agent a_i . Da abgesehen von den Distanzen, die diesen Knoten v betreffen, die gleichen Werte für alle Berechnungen der Heuristiken des Agenten a_i benötigt werden, können diese einmalig zu Beginn des Algorithmus aufsummiert und gespeichert werden.

3.3 Unterschiede zum öffentlichen Personenverkehr

Die Routenplanung in öffentlichen Verkehrsnetzen unterscheidet sich deutlich von der im Straßenverkehr, da die zugrunde liegenden Netzwerke verschiedene Eigenschaften besitzen.

Am entscheidendsten ist dabei die Zeitabhängigkeit der Verkehrsmittel im öffentlichen Personenverkehr, da diese an Zeitpläne gebunden sind. Diese legen die genauen Abfahrts- und Ankunftszeiten der Beförderungsmöglichkeiten an den Stopps fest. Somit ist die Verfügbarkeit und Fahrtdauer von Reisen im Netzwerk an den gegebenen Abfahrtszeitpunkt gebunden und kann sich je nach diesem unterscheiden. Dies macht grundsätzlich die Vorberechnung von Informationen, die später genutzt werden können, komplizierter, da alle möglichen Abfahrtszeiten berücksichtigt werden müssen. Die Fahrtzeiten sind zudem tagesabhängig, da sich die Zeitpläne von Verkehrsmitteln je nach Wochentag unterscheiden. Die Erstellung von zeitabhängigen Graphen resultiert daher in großen Graphen, da alle möglichen Trips berücksichtigt werden müssen. Über diese Graphen können klassische Suchalgorithmen wie der Dijkstra Algorithmus aus [Dij59] zur Berechnung von kürzesten Reisen angewandt werden.

Der Graph eines Straßennetzwerks ist annähernd planar, da hier nur wenige Knoten mit mehr als zwei ausgehenden Kanten existieren. Somit ist die Anzahl der Kanten des Graphen annähernd linear in der Anzahl der Knoten des Straßennetzwerks. Dies ist bei öffentlichen Verkehrsnetzwerken selbst für ihre zeitunabhängige Variante nicht gegeben. Der zeitunabhängige Graph bildet lediglich die Fußwerke des Netzwerkes und Verbindungen zwischen zwei benachbarten Stopps einer Route ab. Im Gegensatz zu dem Graphen eines Straßennetzwerkes existieren hier Knoten mit einer großen Anzahl an ausgehenden Kanten. Beispiele hierfür sind Hauptbahnhöfe von größeren Städten. Die Graphen sind zudem weniger strukturiert, da sich unter anderem Buslinien in Metropolregionen häufig kreuzen. Dies sorgt dafür, dass die zeitunabhängigen Graphen nicht annähernd planar sind, was die Effizienz der Algorithmen aus dem Straßenverkehr reduzieren kann. So haben Dibbelt et al. (2018) in [DPSW17] gezeigt, dass die Berechnung von frühesten Ankunftszeiten und damit verbundenen schnellsten Reisen über klassische Algorithmen für Graphen deutlich langsamer ist als mit extra für öffentliche Personennetzwerke entwickelten Algorithmen.

Ein weiterer Unterschied ist, dass im öffentlichen Personenverkehr alternative Optimalitätskriterien eine größere Rolle als im Straßenverkehr spielen. So wird hier nicht nur nach der kürzesten Fahrtdauer, sondern auch nach der Anzahl der Umstiege, Tarifzonen oder generell der Reisekosten optimiert.

Durch diese Unterschiede werden zur Berechnung von kürzesten Fahrtzeiten und optimalen Treffpunkten andere Algorithmen als im Straßenverkehr benötigt. Diese stellen wir in den folgenden Kapiteln vor.

3.4 Berechnung frühester Ankunftszeiten in öffentlichen Verkehrsnetzwerken

Zur Berechnung von frühesten Ankunftszeiten und den zugehörigen kürzesten Reisen mit öffentlichen Verkehrsmitteln wurden unterschiedliche Möglichkeiten gefunden.

Dies kann über die Umwandlung des öffentlichen Verkehrsnetzes in einen zeitabhängigen Graphen, der alle Abfahrts- und Ankunftszeiten der Trips enthält, und die anschließende Anwendung des Dijkstra Algorithmus, der in [Dij59] definiert ist, erfolgen. Berger et al. (2010) haben in [BGM10] gezeigt, dass diese Herangehensweise durch untere Schranken verbessert werden kann.

Die Berechnung von kürzesten Ankunftszeiten kann über Algorithmen beschleunigt werden, die nicht auf zeitabhängigen Graphen basieren. Ein Konzept sind dabei die Transfer Patterns, die von Bast et al. (2010) in [BCE+10] vorgestellt wurden. Dabei werden in einer Vorberechnungsphase die Fahrtauern von direkten Verbindungen ohne Umstiege und die kürzesten Reisen zu vorher ausgewählten zentralen Stopps vorberechnet. Das Abspeichern dieser Informationen erlaubt eine effiziente Berechnung zur Laufzeit. Dafür ist die Vorberechnung sehr zeitaufwändig und die Speicherung der Informationen benötigt viel Platz.

Der RAPTOR Algorithmus von Delling et al. (2012) in [DPW12] benötigt abgesehen von der Erstellung der zugehörigen Datenstrukturen wie in Kapitel 2.3 beschrieben keine Vorberechnungsphase und berechnet die kürzesten Ankunftszeiten rundenbasiert über Routen. Wir beschreiben seine Vorgehensweise genauer in dem nächsten Kapitel.

Der Connection Scan Algorithm (CSA) Algorithmus von Dibbelt et al. (2017) in [DPSW17] verwendet Verbindungen, die jeweils aus den Stoppzeiten zwischen zwei aufeinanderfolgenden Stopps eines Trips erstellt werden. Dabei reicht eine einmalige Iteration über alle Verbindungen des Verkehrsnetzwerkes sortiert nach ihrer Abfahrtszeit aus, um die frühesten Ankunftszeiten und zugehörige Reisen zu bestimmen. Es wird jeweils ermittelt, ob der Passagier in den Trip der Verbindung umsteigen oder sich bereits in diesem befinden kann. Falls dies der Fall ist, kann die früheste Ankunftszeit an dem Ankunftsstopp der Verbindung aktualisiert werden.

Nach unserem Wissensstand existiert keine Lösung des Treffpunktproblems in öffentlichen Verkehrsnetzwerken. Ahmadi et al. (2018) haben in [AN18] das Problem zwar hinsichtlich des Busverkehrs in Städten betrachtet, aber dabei die Zeitabhängigkeit dieser Verkehrsmittel nicht berücksichtigt. Diese stellt jedoch den entscheidenden Unterschied zu Straßennetzwerken und des Bedarfs nach unterschiedlichen Algorithmen für diese Szenarien dar. Somit besteht das wissenschaftliche Interesse, Algorithmen zur Berechnung der optimalen Treffpunkte im öffentlichen Personenverkehr zu entwerfen und zu analysieren.

4 Bestehende Algorithmen

Als Grundlage für unsere Algorithmen zur Berechnung von frühesten Ankunftszeiten und optimalen Treffpunkten benötigen wir einige bestehende Algorithmen. Als Basis für die Routenberechnung in öffentlichen Verkehrsnetzen dient uns das rundenbasierte Konzept des RAPTOR Algorithmus von Delling et al. (2015) in [DPW12]. Um für diesen Algorithmus im nächsten Kapitel eine zielgerichtete Version einführen zu können, benötigen wir eine Heuristik für die Fahrzeit zwischen zwei gegebenen Stopps des öffentlichen Verkehrsnetzes. Die erste Methode, um diese unteren Schranken für die minimale Fahrdauer zu erhalten, wurde von Bruns (2024) in [Bru24] beschrieben.

4.1 RAPTOR

Die Familie der RAPTOR Algorithmen wurde von Delling et al. (2015) in [DPW12] vorgestellt und ihre Algorithmen können genutzt werden, um früheste Ankunftszeiten in öffentlichen Personennetzwerken zu ermitteln. Sie suchen kürzeste Reisen rundenbasiert über die Routen des Verkehrsnetzwerkes. Dabei erweitern wir in jeder Runde des Algorithmus die bisher betrachteten Reisen um einen Streckenabschnitt und somit um einen Umstieg. So erhalten wir nach k Runden des Algorithmus die frühesten Ankunftszeiten, deren Reisen maximal aus k Streckenabschnitten bestehen und Passagiere somit höchstens $k - 1$ Mal umsteigen müssen.

Der Standard RAPTOR Algorithmus berechnet zur Lösung des EAT Problems die frühesten Ankunftszeiten an allen Stopps ausgehend vom Startstopp s und der minimalen Abfahrtszeit τ_s rundenbasiert.

Dazu speichern wir für jeden Stopp p des Netzwerks die frühesten Ankunftszeiten der aktuellen ($\tau_c(p)$) und vorangegangenen Runde ($\tau_p(p)$) in zwei Vektoren. Die minimale Ankunftszeit der aktuellen Runde stellt dabei auch die früheste bisher bekannte Ankunftszeit am Stopp p dar. Wir benutzen zudem zwei Vektoren, in denen wir die markierten Stopps der vorherigen und aktuellen Runde einfügen. Über Reisezeiger, die wir für jeden Stopp abspeichern, können später die Reisen der minimalen Ankunftszeiten wiederhergestellt werden.

Um den Algorithmus zu initialisieren, iterieren wir über alle ausgehenden Fußwege $F(s)$ vom Startstopp s . Für jeden Fußweg $f \in F(s)$ mit Zielstopp $f_{\text{arr_stop}}$ setzen wir als früheste Ankunftszeit $\tau_c(f_{\text{arr_stop}}) = \tau_s + f_{\text{dur}}$ mit Dauer f_{dur} des Fußweges. Zudem speichern wir den Stopp $f_{\text{arr_stop}}$ in den aktuell markierten Stopps. Für den Startstopp s setzen wir $\tau_c(s) = \tau_s$.

Zu Beginn jeder Runde ersetzen wir die Vektoren der vorherigen frühesten Ankunftszeiten und markierten Stopps durch die der frühesten Ankunftszeiten und markierten Stopps der vorangegangenen Runde. Zudem initialisieren wir den Vektor der aktuell markierten Stopps mit einem leeren Vektor.

Anschließend führen wir in jeder Runde zwei Schritte durch. Zunächst ermitteln wir aus den markierten Stopps der letzten Runde alle Routen, die in dieser Runde betrachtet werden müssen. Daraufhin iterieren wir über die Stopps dieser Routen und aktualisieren die frühesten Ankunftszeiten dieser.

Im ersten Schritt suchen wir alle relevanten Routen-Stopp-Paare der aktuellen Runde und speichern sie im Vektor Q . Für alle markierten Stopps p der vorherigen Runde ermitteln wir alle Routen r , in denen p enthalten ist. Wir speichern das zugehörige Routen-Stopp-Paar (r, p) in Q ab. Falls bereits ein Paar (r, p') in Q enthalten ist, überprüfen wir, ob p sich vor p' innerhalb der Route r befindet. Ist dies der Fall, ersetzen wir (r, p') durch (r, p) . Andernfalls wird (r, p) nicht abgespeichert. Nach der Iteration über alle markierten Stopps sind in Q alle Routen gespeichert, die in dieser Runde per Umstieg erreicht werden können. Der zugehörige Stopp der Routen-Stopp-Paare stellt den ersten Einstiegspunkt der Route dar.

Im zweiten Teil des Algorithmus betrachten wir alle Routen-Stopp-Paare (r, p) in Q . Dazu ermitteln wir zunächst den frühesten Trip t' der Route r , der an Stopp p nach der frühesten Ankunftszeit $\tau_p(p)$ der vorherigen Runde erreicht werden kann. Anschließend iterieren wir über alle folgenden Stopps der Route r und überprüfen, ob wir durch diesen Trip die Stopps früher als bisher bekannt erreichen können. Bei Betrachtung des Stopps p' aus der Route r ermitteln wir dazu alle ausgehenden Fußwege $F(p')$ von p' . Für alle Fußwege $f \in F(p')$ mit Zielstopp f_{arr_stop} bestimmen wir die Ankunftszeit $\tau_{arr}(f_{arr_stop}) = \tau_{arr}(t', p') + f_{dur}$ aus der Ankunftszeit des Trips und der Dauer des Fußweges. Falls $\tau_{arr}(f_{arr_stop}) < \tau_c(f_{arr_stop})$ speichern wir diese neue früheste Ankunftszeit in τ_c und markieren den Stopp f_{arr_stop} . Als Reisezeiger speichern wir dann für f_{arr_stop} den Einstiegsstopp des Trips und die Abfahrtszeit dessen an diesem Stopp. Falls der Zielstopp t des EAT Problems bekannt ist, müssen wir die Aktualisierung der frühesten Ankunftszeit f_{arr_stop} nur vornehmen, falls für seine früheste Ankunftszeit $\tau_{arr}(f_{arr_stop}) < \tau_c(t)$ gilt. Dieses Vorgehen wird als zielgerichtete Vereinfachung (target pruning) bezeichnet. Nachdem die Ankunftszeiten aller ausgehenden Fußwege aktualisiert wurden, überprüfen wir, ob ein früherer Trip t'' der Route r an Stopp p' erreicht werden kann. Dies ist nur möglich, falls der Stopp in der vorherigen Runde markiert wurde und die früheste Ankunftszeit der vorherigen Runde vor der Abfahrtszeit des Trips t' liegt. Der Stopp muss in der vorherigen Runde markiert worden sein, da wir ansonsten den Trip t'' auch schon betrachtet haben. Bei der Auswahl der frühesten erreichbaren Trips einer Route an seinen Stopps ist es ausreichend, einmal über die sortierten Trips der Route zu iterieren, da an späteren Stopps der Route lediglich frühere Trips ausgewählt werden können.

Am Ende jeder Runde überprüfen wir, ob in dieser mindestens ein Stopp markiert wurde. Ist dies nicht der Fall, terminiert der Algorithmus. Nach der Terminierung des Algorithmus stehen die frühesten Ankunftszeiten in τ_c . Die zugehörige Reise zu einem Zielstopp t kann über die Reisezeiger ausgehend von t ermittelt werden. Dazu wird der letzte Streckenabschnitt

zwischen dem Einstiegsstopp des Reisezeigers, der für t gespeichert wurde, und t erstellt. Ausgehend von diesem Einstiegsstopp wird daraufhin der nächste Streckenabschnitt der Reise extrahiert. Wir wiederholen dies so lange, bis der Startstopp s erreicht ist. Die Fußwege lassen sich jeweils über die Ausstiegs- und Einstiegsstopps der Streckenabschnitte ermitteln, sodass eine zusammenhängende Reise entsteht.

Mit dem RAPTOR Algorithmus lässt sich auch das LDT Problem lösen. Dazu muss der Algorithmus rückwärtsgewandt, ausgehend von dem Zielstopp und der spätesten Ankunftszeit τ_t , durchgeführt werden. Dabei werden die einzelnen Reisen in umgekehrter Reihenfolge ihrer Streckenabschnitte und Umstiege rundenbasiert ermittelt, um die zugehörigen spätesten Abfahrtszeiten an den Ausgangsstopps zu erhalten, damit die Passagiere vor τ_t an t ankommen.

4.2 Untere Schranken über zeitunabhängige Graphen

Um die Algorithmen zur Berechnung von frühesten Ankunftszeiten und optimalen Treffpunkten zu beschleunigen, verwenden wir Heuristiken $h_{\text{dur}}(u, v)$ für die Reisezeit. Diese stellt eine untere Schranke für die minimale Fahrtzeit zwischen den Stopps u und v zu jedem möglichen Startzeitpunkt dar. Mit diesen können überflüssige Berechnungen vermieden und die RAPTOR Algorithmen zielgerichteter ausgeführt werden. Die unteren Schranken berechnen wir auf zwei unterschiedlichen Arten. In der ersten Methode entfernen wir alle zeitabhängigen Informationen aus den Netzwerken und berechnen die minimalen Fahrtzeiten ohne Berücksichtigung dieser über Kontraktionshierarchien (Contraction Hierarchies) und den Parallel hardware-accelerated shortest path trees (PHAST) Algorithmus. Dabei gehen wir analog zu Bruns (2024) in [Bru24] vor. Die zweite Variante stellt eine neue Möglichkeit für öffentliche Verkehrsnetzwerke dar und findet sich dementsprechend in Kapitel 5.1 wieder.

Die erste Methode, um untere Schranken für die Fahrtdauern in öffentlichen Personennetzwerken zu berechnen, basiert auf zeitunabhängigen Graphen $G(V, E)$. Die Knotenmenge V des Graphen G besteht aus den Stopps des Netzwerkes. Um die Kantenmenge zu erstellen, iterieren wir über alle Trips und Fußwege des Netzwerkes. Wir erstellen eine Kante (u, v) zwischen zwei Stopps u und v , falls ein Fußweg zwischen ihnen existiert oder die Stopps innerhalb eines Trips direkt aufeinanderfolgen. Die minimale Dauer der Trips und Fußwege zwischen den Knoten u und v stellt das Kantengewicht von (u, v) dar. Um die Eigenschaft des ungerichteten Graphens der Clique-Heuristik, die auf der h_{dur} Heuristik basiert, zu gewährleisten, erstellen und aktualisieren wir benötigte Rückwärtskanten.

Die untere Schranke $h_{\text{dur}}(u, v)$ für die Fahrtdauer zwischen zwei Stopps u und v kann nun über Algorithmen zur Berechnung von kürzesten Entfernungen in Graphen berechnet werden. Diese Heuristik entspricht der minimalen Distanz $d_{\text{zu}}(u, v)$ zwischen den Knoten u und v in dem zeitunabhängigen Graphen. Sie kann über klassische Algorithmen zur Berechnung von kürzesten Wegen in Graphen wie dem Dijkstra Algorithmus in [Dij59] ermittelt werden. Zur Laufzeit der Algorithmen zur Lösung des Treffpunkt-Problems müssen wir die Distanzen von

einzelnen Stopps zu allen anderen Knoten effizient berechnen. Daher verwenden wir nicht den Dijkstra Algorithmus. Alternativ erstellen wir in einer einmaligen Vorberechnungsphase eine Kontraktionshierarchie für den Graphen wie in [GSSD08] beschrieben und können auf ihm zur Laufzeit den PHAST Algorithmus aus [DGNW11] durchführen. Die Distanzen der klassischen Algorithmen auf Graphen werden in diesem Fall durch die ermittelten minimalen Fahrtzeiten als Kantengewichte ersetzt. Für die Algorithmen zur Lösung des EAT Problems benötigen wir die unteren Schranken von allen Knoten zum Zielstopp t . Dies kann ebenfalls effizient über die Kontraktionshierarchie und den PHAST Algorithmus ermittelt werden.

Die Berechnung der Kontraktionshierarchie resultiert darin, dass jedem Knoten ein Level zugeordnet wird. Zudem erstellen wir Abkürzungen, die Kanten ersetzen, die durch das kontrahieren von Knoten entfernt wurden. Bei der Kontraktion eines Knotens wird dieser und alle seine adjazenten Kanten aus dem Graphen entfernt. Um kürzeste Pfade nicht zu verändern, werden Kanten als Abkürzungen zwischen den Nachbarknoten des kontrahierten Knotens eingefügt.

Das nutzen wir, um mit dem PHAST Algorithmus in zwei Schritten die minimalen Fahrtzeiten von gegebenen Ausgangsstopps N zu allen anderen Stopps zu berechnen. Dazu führen wir zunächst auf dem Graphen für jeden Stopp $s \in N$ einen Dijkstra auf dem Aufwärtsgraphen der Kontraktionshierarchie durch. Der Aufwärtsgraphen besteht dabei lediglich aus Kanten (x, y) , wobei x ein kleineres Level als y besitzt. Diese Berechnung kann parallel für alle Stopps in N durchgeführt werden. Ausgehend von dem Stopp mit dem höchsten Level iterieren wir über alle Knoten in absteigender Levelhöhe und aktualisieren die minimalen Fahrdauern der Stopps. Dabei ist eine gemeinsame Iteration für alle Ausgangsstopps ausreichend, da die Aktualisierung für die Berechnungen einzeln erfolgen kann.

Lemma 1

Für jedes Paar (u, v) an Stopps mit $u, v \in S$ ist die minimale Distanz $d_{zu}(u, v)$ im zeitunabhängigen Graphen kleiner gleich der minimalen Fahrdauer $d(u, v)$ zwischen u und v zu jedem möglichen Startzeitpunkt τ_s . Diese Distanz des kürzesten Weges im zeitunabhängigen Graphen wird als Heuristik $h_{dur}(u, v)$ gewählt.

Beweis. Sei $d(u, v)$ die minimale Fahrdauer zwischen u und v mit dem Startzeitpunkt τ_s . Angenommen, es gilt $d(u, v) < d_{zu}(u, v)$ mit $d_{zu}(u, v)$ als minimale Distanz im zeitunabhängigen Graphen. Dann existiert auf der kürzesten Reise zwischen u und v eine Verbindung zwischen zwei benachbarten Stopps x und y mit $d(x, y) < d_{zu}(x, y)$. Ausgehend von der Definition der zeitunabhängigen Graphen ist dies nicht möglich. Dieser Widerspruch resultiert in der Korrektheit des Lemmas. $\square$

Die Berechnung der unteren Schranken über die zeitunabhängigen Graphen besitzt zwei Nachteile. Durch das Entfernen aller zeitabhängigen Informationen werden auch mögliche Umstiegs- und Wartezeiten nicht mehr berücksichtigt. Da diese einen entscheidenden Anteil bei Fahrdauern von längeren Reisen darstellen, sorgt ihr Weglassen für eine Differenz

zwischen echten Fahrtzeiten und den unteren Schranken. Zudem muss die Berechnung der Distanzen über den PHAST Algorithmus zur Laufzeit erfolgen und sorgt somit für längere Ausführungszeiten.

5 Neue Algorithmen

Die bisher betrachtete Variante zur Berechnung von Heuristiken für die minimale Fahrtzeit zwischen zwei Stopps des öffentlichen Netzwerkes über die minimale Distanz im zeitunabhängigen Graphen besitzt den Nachteil, dass die Berechnung zur Laufzeit der Algorithmen erfolgen muss. Wir entwickeln daher eine neue Variante über Landmarken, bei der der Großteil der Berechnung in einer einmaligen Vorberechnungsphase erfolgt. Um das Treffpunkt-Problem lösen zu können, müssen wir zunächst den klassischen RAPTOR Algorithmus so anpassen, dass er die Heuristiken für die Fahrtzeiten nutzt, um über eine Prioritätenfunktion den Suchraum einzuschränken. Wir entwickeln dabei zwei neue Algorithmen zur Lösung des EAT Problems. Anschließend nutzen wir diese als Basis, um das Treffpunkt-Problem optimal zu lösen. Die beiden dazu entwickelten Algorithmen nutzen die Prioritätenfunktionen aus Kapitel 3.2, um die Menge an möglichen optimalen Treffpunkten im Vergleich zur naiven Suche einzuschränken. Abschließend entwickeln wir eine Methode, die für eine große Anzahl an Ausgangsstopps den optimalen Treffpunkt über die Auswahl einer Teilmenge der Ausgangsstopps und die genaue Berechnung dieser findet. Diesen Algorithmus entwickeln wir in unterschiedlichen Varianten so, dass er einerseits den optimalen Treffpunkt angenähert und andererseits exakt über Iterationen sucht.

5.1 Untere Schranken über Landmarken

In Kapitel 4.2 haben wir die Methode von Bruns (2024) in [Bru24] vorgestellt, um eine Heuristik h_{dur} für die minimale Fahrtzeit zwischen zwei Stopps über zeitunabhängige Graphen zu berechnen. Alternativ führen wir dazu nun die Möglichkeit ein, untere Schranken in öffentlichen Verkehrsnetzen über Landmarken zu bestimmen. Die Berechnung erfolgt ebenfalls auf dem zeitunabhängigen Graphen des Verkehrsnetzes. Das Vorgehen ist dabei analog zu Landmarken in Straßengraphen, wie von Goldberg et al. (2004) in [GH05] beschrieben.

Grundlage dieser Variante ist eine Menge $L \subset S$ an Stopps, die ausgewählt und als Landmarken bezeichnet werden. In einer einmaligen Vorberechnungsphase berechnen wir für jede Landmarke $l \in L$ und jeden Stopp p des Netzwerkes die minimalen Fahrtzeiten $d_{\text{zu}}(l, p)$ von l nach p und $d_{\text{zu}}(p, l)$ von p nach l im zeitunabhängigen Graphen. Dies kann über die Ausführung eines Vor- und Rückwärts-Dijkstra Algorithmus aus [Dij59] ausgehend von jeder Landmarke erfolgen. Aus der Definition des zeitunabhängigen Graphens folgt $d_{\text{zu}}(u, v) \leq d(u, v)$ für alle Stopps u, v und die minimale Fahrtzeit $d(u, v)$ im zeitabhängigen Kontext.

Für alle Stopps u, v und Landmarken $l \in L$ gelten die folgenden Dreiecksungleichungen im zeitunabhängigen Graphen:

$$\begin{aligned} d_{zu}(u, v) + d_{zu}(v, l) &\geq d_{zu}(u, l) \\ d_{zu}(l, u) + d_{zu}(u, v) &\geq d_{zu}(l, v) \end{aligned}$$

Daraus folgt für die minimale Fahrtdauer von u nach v :

$$d_{zu}(u, v) \geq \max_{l \in L} \max\{d_{zu}(u, l) - d_{zu}(v, l), d_{zu}(l, v) - d_{zu}(l, u)\}$$

Lemma 2

Für jedes Paar (u, v) an Stopps mit $u, v \in S$ ist $h_{dur}(u, v) = \max_{l \in L} \max\{d_{zu}(u, l) - d_{zu}(v, l), d_{zu}(l, v) - d_{zu}(l, u)\}$ kleiner gleich der minimalen Fahrtdauer zwischen u und v zu jedem möglichen Startzeitpunkt τ_s .

Beweis. Die Korrektheit des Lemmas folgt direkt aus den vorangegangenen Dreiecksungleichungen und $d_{zu}(u, v) \leq d(u, v)$ für alle $u, v \in S$. $\square$

Somit können wir die so berechnete Heuristik $h_{dur}(u, v)$ als untere Schranke für die Fahrtdauer zwischen den Stopps u und v zu jeder möglichen Abfahrtszeit verwenden.

Eine gute Heuristik besitzt eine möglichst kleine Differenz zu dem eigentlichen Wert der minimalen Fahrtdauer. Dies ist bei einer Fahrtdauer von u nach v und Landmarke l dann gegeben, falls die Reise der minimalen Fahrtdauer von u zu l über den Stopp v führt. Wir versuchen daher Landmarken auszuwählen, die möglichst weit am Rande des öffentlichen Verkehrsnetzes liegen. Dabei wenden wir die Farthest Strategie von Goldberg et al. (2004) in [GH05] an. Bei dieser Herangehensweise wird zu Beginn $L = \emptyset$ initialisiert und ein beliebiger Stopp v_1 ausgewählt. Mit dem Dijkstra Algorithmus wird der am weitesten entfernte Stopp von v_1 im zeitunabhängigen Graphen ermittelt und in L eingefügt. Anschließend werden iterativ die Distanzen von der zuletzt eingefügten Landmarke berechnet und der Stopps als nächste Landmarke ausgewählt, der von allen Landmarken in L am weitesten entfernt ist. Dies wird so lange wiederholt, bis L eine vorher definierte Anzahl an Stopps enthält.

Im Vergleich zu der Berechnung der unteren Schranken über die zeitunabhängigen Graphen besitzen die Landmarken den Vorteil, dass zur Laufzeit keine aufwändigen Berechnungen erfolgen müssen.

5.2 Früheste Ankunftszeit

Bevor wir das Treffpunkt-Problem lösen, müssen wir zunächst den klassischen RAPTOR Algorithmus so anpassen, dass er die Heuristiken für die minimale Fahrtzeit zwischen zwei Stopps nutzen kann. Wir nutzen daher nun diese Heuristiken, um den RAPTOR Algorithmus zur

Lösung des EAT Problems zielgerichteter auszuführen und Routen von der Betrachtung auszuschließen. Dabei stellen wir zwei unterschiedliche Algorithmen vor. Der Round-Based Public Transit Routing Bound (RAPTOR B)* Algorithmus ist eine Variante des Standard RAPTOR Algorithmus, die ebenfalls rundenbasiert arbeitet. Im Gegensatz dazu wird bei dem Round-Based Public Transit Routing Priority Queue (RAPTOR PQ)* Algorithmus das rundenbasierte Konzept durch eine Prioritätenwarteschlange ersetzt. Wir gehen dabei in Anlehnung an den A* Algorithmus für Graphen von Hart et al. (1968) in [HNR68] vor.

Prioritätenfunktion

Zur Lösung des EAT Problems suchen wir nach der minimalen Ankunftszeit am Zielstopp t ausgehend von Startstopp s und Startzeitpunkt τ_s . In beiden neu entwickelten Algorithmen verwenden wir die Prioritätenfunktion f , die eine untere Schranke für die früheste Ankunftszeit am Zielstopp darstellt und somit zur Einschränkung des Suchraums und der zielgerichteten Berechnung des optimalen Ergebnisses genutzt werden kann. Für einen Stopp $p \in S$ sei $g(p)$ die bereits berechnete früheste Ankunftszeit an p ausgehend von s und τ_s . Die Heuristik $h_{\text{dur}}(p, t)$ stellt eine untere Schranke für die minimale Fahrtzeit zwischen p und t dar und wird, wie in den Kapiteln 4.2 und 5.1 beschrieben, direkt über die minimale Distanz im zeitunabhängigen Graphen oder unter Verwendung von Landmarken berechnet. Die Prioritätenfunktion $f(p)$ ergibt sich aus der Summe der frühesten Ankunftszeit an p und der Heuristik bis zum Ziel:

$$f(p) = g(p) + h_{\text{dur}}(p, t).$$

Falls wir wie Bruns in [Bru24] als Heuristik die minimale Fahrtzeit im zeitunabhängigen Graphen verwenden, müssen wir in einer einmaligen Vorberechnungsphase lediglich die Kontraktionshierarchie für diesen erstellen. Zur Laufzeit der Algorithmen muss über eine einmalige Ausführung des PHAST Algorithmus die minimale Fahrtzeit ausgehend von allen Stopps des Netzwerkes hin zum Zielstopp t im zeitunabhängigen Graphen berechnet werden.

Im Fall der Berechnung der Heuristik über die Landmarken ermitteln wir in einer einmaligen Vorberechnung die minimalen Fahrtzeiten im zeitunabhängigen Graphen für alle Landmarken L . Zu Beginn der Ausführung der Algorithmen zur Lösung des EAT Problems wählen wir k Landmarken aus der Menge L aus, um während der Ausführung weniger Vergleiche durchführen zu müssen. Dabei wählen wir die k Landmarken mit der größten Heuristik $h_{\text{dur}}(s, t)$, da sie die beste untere Schranke für die Fahrtzeit zwischen Start- und Zielstopp liefern und somit auch für die anderen Stopps auf dieser Route am geeignetsten sind. Zur Laufzeit können die benötigten Heuristiken $h_{\text{dur}}(p, t)$ für Stopps p mit den vorberechneten Werten für die Landmarken und der Formel aus Kapitel 5.1 berechnet werden.

5.2.1 RAPTOR B*

Wir modifizieren den Standard RAPTOR Algorithmus so, dass er die Heuristik der Fahrtzeiten und die daraus resultierende untere Schranke für die früheste Ankunftszeit am Ziel nutzt, um überflüssige Routen nicht zu betrachten. Im Folgenden zeigen wir, wie diese Modifikation aussieht und beweisen die Korrektheit des Algorithmus.

Beschreibung

Der RAPTOR B* Algorithmus entspricht dem Standard RAPTOR Algorithmus, der bei der Auswahl der relevanten Routen zu Beginn jeder Runde angepasst wird. Dazu berechnen wir für jeden markierten Stopp p der vorherigen Runde die untere Schranke $f(p)$ der frühesten Ankunftszeit am Zielstopp t . Diese setzt sich wie zu Beginn des Kapitels beschrieben aus der Summe der frühesten Ankunftszeit an p und der Heuristik für die verbleibende Fahrtzeit zum Ziel zusammen:

$$f(p) = \tau_c(p) + h_{\text{dur}}(p, t)$$

Wir wählen nur die Routen der markierten Stopps p aus, deren Prioritätenfunktion $f(p)$ kleiner als das bisher beste Ergebnis $\tau_c(t)$ ist. Somit werden in dieser Runde keine Routen betrachtet, über deren Einstiegsstopps es nicht möglich ist, den Zielstopp zu einer früheren Ankunftszeit zu erreichen.

Der Vorteil des RAPTOR B* Algorithmus ist es, dass das rundenbasierte Vorgehen des RAPTOR Algorithmus bestehen bleibt. Dies bewirkt, dass die Lösung optimal hinsichtlich frühester Ankunftszeit und minimaler Anzahl an Umstiegen auf der Reise ist. Jedoch werden die Routen trotz des Ausschlusses einiger weiterhin Runde für Runde in alle Richtungen expandiert und es kommt zu keinem zielgerichteten Vorgehen.

Korrektheit

Um die Korrektheit des RAPTOR B* Algorithmus zu beweisen, verwenden wir die beiden folgenden Lemmata.

Lemma 3

(Vollständigkeit). Der RAPTOR B Algorithmus garantiert, falls eine Lösung existiert, diese und andernfalls ∞ zurückzugeben.*

Beweis. Der RAPTOR B* Algorithmus führt ausgehend von s über die Routen eine Breitensuche durch. Im schlimmsten Fall expandiert der Algorithmus dabei alle erreichbaren Routen und findet somit für jeden erreichbaren Stopp eine minimale Ankunftszeit mit zugehöriger Reise. Falls somit eine Lösung des EAT Problems für den gegebenen Start- s und Zielstopp t zur

Abfahrtszeit τ_s existiert, wird diese durch den RAPTOR B* Algorithmus gefunden. Sollte t von s aus nicht erreichbar sein, gibt der Algorithmus ∞ zurück. $\square$

Lemma 4

(Optimalität). Für eine gegebene zulässige Funktion f garantiert der RAPTOR B* Algorithmus die minimale Ankunftszeit an Zielstopp t und die zugehörige Reise ausgehend von s und τ_s zurückzugeben.

Beweis. Die Korrektheit des RAPTOR Algorithmus wurde durch Delling et al. in [DPW12] gezeigt. Aus der Zulässigkeit der Funktion f folgt, dass für jeden Umstiegsstopp p der kürzesten Reise $f(p)$ kleiner als die minimale Ankunftszeitpunkt an t ist. Somit werden die Routen der optimalen Reise nicht aus der Betrachtung des RAPTOR B* Algorithmus ausgeschlossen. Die Korrektheit des RAPTOR Algorithmus resultiert darin, dass der RAPTOR B* Algorithmus die optimale Ankunftszeit und deren Reise zurückgibt. $\square$

5.2.2 RAPTOR PQ*

Der RAPTOR PQ* Algorithmus nutzt ebenfalls die unteren Schranken, um die Anzahl der betrachteten Routen des Standard RAPTOR Algorithmus zu reduzieren. Dabei verwenden wir statt des rundenbasierten Vorgehens eine Prioritätenwarteschlange sortiert nach der definierten Prioritätenfunktion, um die Routen weniger in alle Richtungen und dafür zielgerichteter zu expandieren. Im Folgenden beschreiben wir den daraus resultierenden Algorithmus und zeigen dessen Korrektheit.

Beschreibung

Um die unteren Schranken und daraus resultierende Prioritätenfunktion noch mehr als beim RAPTOR B* Algorithmus zu nutzen, verwenden wir beim RAPTOR PQ* Algorithmus eine Prioritätenwarteschlange PQ . Diese speichert Paare bestehend aus dem Wert der Prioritätenfunktion und der zugehörigen Route ab. Die Prioritätenfunktion f stellt die Summe aus der bisher bekannten frühesten Ankunftszeit am Einstiegsstopp der Route und der Heuristik der Fahrtzeit ausgehend von diesem zum Ziel dar. Sie ist somit eine untere Schranke der minimalen Ankunftszeit am Zielstopp. Die Prioritätenwarteschlange ist nach dieser Funktion sortiert und sorgt daher für ein zielgerichtetes Expandieren der Routen. Im Vergleich zum Standard RAPTOR Algorithmus müssen keine Stopps mehr markiert werden. Dafür speichern wir in zwei Vektoren für jede Route r den aktuell kleinsten heuristischen Wert der Prioritätsfunktion ($sH[r]$) und den frühesten Einstiegsstopp ($eS[r]$) in diese Route. Das erlaubt es uns, Elemente in der Prioritätenwarteschlange nicht aktualisieren zu müssen. Durch Vergleiche mit den

gespeicherten Werten sichern wir dennoch, dass es nicht zu unnötigen mehrfachen Betrachtungen von Routen in der Warteschlange kommt. Da der Algorithmus nicht rundenbasiert arbeitet, speichern wir für jeden Stopp p lediglich die frühest mögliche Ankunftszeit in $\tau[p]$. Diese initialisieren wir für jeden Stopp mit ∞ (Zeile 2 – 4 in Algorithmus 5.1).

Während der Initialisierungsphase des Algorithmus fügen wir die Routen des Ausgangsstopps s und die der von ihm per Fußweg erreichbaren Stopps in die Prioritätenwarteschlange ein (Z. 9 – 17). Dazu berechnen wir den heuristischen Wert der Prioritätenfunktion der Stopps. Zudem aktualisieren wir die frühesten Ankunftszeiten dieser Stopps in τ .

Anschließend gehen wir wie folgt vor, bis die Prioritätenwarteschlange leer ist.

Zunächst entnehmen wir das erste Element der Prioritätenwarteschlange und überprüfen, ob sein Wert der Prioritätenfunktion f größer als das bisher beste Ergebnis ist (Z. 19 – 22). Ist dies der Fall, müssen wir die verbleibenden Elemente in der Warteschlange nicht mehr betrachten und der Algorithmus terminiert. Zudem vergleichen wir die Prioritätenfunktion des entnommenen Elements mit dem minimalen Wert der Route. Falls diese nicht übereinstimmen, betrachten wir nicht die zugehörige Route (Z. 23 – 25). Andernfalls starten wir am ersten Einstiegsstopp der Route und iterieren über dessen Stopps in der Reihenfolge, in der sie angefahren werden (Z. 27–39). Dabei ermitteln wir analog zum Standard RAPTOR Algorithmus den frühesten erreichbaren Trip (Z. 38) und aktualisieren die Ankunftszeiten der Stopps, falls wir über die Route früher als bisher bekannt an ihnen ankommen (Z. 29–36). Zudem überprüfen wir, ob es möglich ist, in frühere Trips der Route umzusteigen.

Wenn es möglich ist, die früheste Ankunftszeit eines Stopps p zu aktualisieren, fügen wir dessen erreichbaren Routen in die Warteschlange ein. Dazu berechnen wir für diesen Stopp p die Prioritätenfunktion $f(p)$ über die Summe aus frühester Ankunftszeit $\tau_c(p)$ und der Heuristik $h_{\text{dur}}(p, t)$ der Fahrtzeit zum Zielstopp t (Z. 47). Falls $f(p) > \tau_c(t)$ fügen wir keine neuen Routen für den Stopp p ein (Z. 48 – 50). Andernfalls iterieren wir über alle Routen r , in denen p enthalten ist (Z. 41 – 59). Dabei überprüfen wir, ob der heuristische Wert $f(p)$ kleiner als der minimale Wert dieser Route in der Warteschlange ist (Z. 52). Dies erfolgt effizient über die dazu gespeicherten Werte. Falls dies erfüllt ist, aktualisieren wir den Eintrag in dem Vektor und fügen die Route mit $f(p)$ in die Warteschlange ein (Z. 53). Wenn der Stopp p in der Route vor dem bisher ersten gespeicherten Einstiegsstopp angefahren wird, speichern wir ihn als neuen Umstiegsstopp ab (Z. 56 – 58).

Am Ende der Betrachtung der Route setzen wir den kleinsten heuristischen Wert und den ersten Einstiegsstopp der Route zurück (Z. 40 – 41).

Nach der Terminierung des Algorithmus kann die Reise der minimalen Ankunftszeit analog zum Standard RAPTOR Algorithmus über die Reisezeiger extrahiert werden.

Der RAPTOR PQ* Algorithmus besitzt den Vorteil, dass die Routen nicht rundenbasiert gleichmäßig in alle Richtungen expandiert werden, sondern priorisiert in die Richtung des Zielstopps. Jedoch erfolgt durch ihn nicht mehr die Minimierung nach frühester Ankunftszeit und minimaler Anzahl an Umstiegen.

Algorithmus 5.1 RAPTOR PQ*

```

1: procedure RAPTORPQ*( $s, t, \tau_s$ )
2:   for all  $i \in S$  do
3:      $\tau[i] \leftarrow \infty$ 
4:   end for
5:   for all  $r \in R$  do
6:      $eS[r] \leftarrow \infty$ 
7:      $sH[r] \leftarrow \infty$ 
8:   end for
9:   for all  $fp \in F(s)$  do
10:     $v \leftarrow fp_{\text{arr\_stop}}$ 
11:    if  $v = s$  then
12:       $\tau[v] \leftarrow \tau_s$ 
13:    else
14:       $\tau[v] \leftarrow \tau_s + fp_{\text{dur}}$ 
15:    end if
16:    INSERTROUTES( $v$ )
17:  end for
18:  while PQ is not empty do
19:    Extract  $(f(v), r)$  from PQ // with lowest  $f(v)$ 
20:    if  $f(v) > \tau[t]$  then
21:      return  $\tau[t]$ 
22:    end if
23:    if  $f(v) \neq sH[r]$  or  $eS[r] = \infty$  then
24:      continue
25:    end if
26:     $t' \leftarrow \infty$ 
27:    for all stops  $p_i$  of  $r$  beginning with  $eS[r]$  do
28:      if  $t' \neq \perp$  then
29:        for all  $fp \in F(p_i)$  do
30:           $v \leftarrow fp_{\text{arr\_stop}}$ 
31:           $arr \leftarrow \tau_{\text{arr}}(t', p_i) + fp_{\text{dur}}$ 
32:          if  $arr < \tau[v]$  then
33:             $\tau[v] \leftarrow arr$ 
34:            INSERTROUTES( $v$ )
35:          end if
36:        end for
37:      end if
38:       $t \leftarrow et(r, p_i)$  // earliest trip that can be caught at  $p_i$ 
39:    end for
40:     $eS[r] \leftarrow \infty$ 
41:     $sH[r] \leftarrow \infty$ 

```

```

42:   end while
43:   return  $\tau[t]$ 
44: end procedure
45:
46: procedure INSERTROUTES( $p$ )
47:    $f(p) \leftarrow \tau[p] + h_{\text{dur}}(p, t)$ 
48:   if  $f(p) > \tau[t]$  then
49:     return
50:   end if
51:   for all Routes  $r$  that contain  $p$  do
52:     if  $f(p) < sH[r]$  then
53:       Insert  $(f(p), r)$  into  $PQ$ 
54:        $sH[r] \leftarrow f(p)$ 
55:     end if
56:     if  $p$  is reached before  $eS[r]$  in  $r$  then
57:        $eS[r] \leftarrow p$ 
58:     end if
59:   end for
60: end procedure

```

Korrektheit

Die Korrektheit des RAPTOR PQ* Algorithmus zeigen wir über die beiden folgenden Lemmata.

Lemma 5

(Vollständigkeit). Der RAPTOR PQ* Algorithmus garantiert, falls eine Lösung existiert, diese und andernfalls ∞ zurückzugeben.

Beweis. Der RAPTOR PQ* Algorithmus führt ausgehend von s eine Bestensuche (best-first search) durch. Dabei ist die zugrunde liegende Prioritätenwarteschlange nach der Funktion f sortiert. Der Algorithmus stoppt im schlechtesten Fall, wenn die Warteschlange leer ist und somit alle erreichbaren Stopps ausgehend von s gefunden wurden. Wird so der Zielstopp t nicht erreicht, gibt der Algorithmus als minimale Ankunftszeit ∞ zurück. Andernfalls stoppt der Algorithmus, sobald der minimale Eintrag in der Prioritätenwarteschlange einen höheren Wert als die minimale Ankunftszeit besitzt. Dazu muss der Algorithmus t gefunden und ein Ergebnis ungleich ∞ zurückgegeben haben. $\square$

Lemma 6

(Optimalität). Für eine gegebene zulässige Funktion f garantiert der RAPTOR PQ* Algorithmus die früheste Ankunftszeit am Zielstopp t und die zugehörige Reise ausgehend von s und der Abfahrtszeit τ_s zurückzugeben.

Beweis. Angenommen der RAPTOR PQ* Algorithmus gibt ein nicht optimales Ergebnis $\tau_c(t)$ mit $\tau_c(t) > \tau_c^*(t)$ und $\tau_c^*(t)$ als frühestmöglicher Ankunftszeit an t zurück. Da der RAPTOR PQ* frühzeitig mit der Lösung $\tau_c(t)$ gestoppt ist, befindet sich die noch nicht expandierte Route r' mit Umstiegsstopp p' , die sich auf dem optimalen Pfad befindet, in der Prioritätenwarteschlange. Seien dabei alle Routen mit zugehörigen Umstiegsstopps, die auf der optimalen Reise vor r' und p' liegen schon expandiert. Daher wurde Eintrag von r' in der Prioritätenwarteschlange von der vorherigen Route auf der optimalen Reise mit dem Wert $f(p')$ generiert. Aus der Zulässigkeit der Funktion f folgt per Definition $f(p') \leq \tau_c^*(t)$. Da $\tau_c^*(t) < \tau_c(t)$ gilt, ist die Abbruchbedingung der Prioritätenwarteschlange durch $f(p') < \tau_c(t)$ verletzt. Dies resultiert in einem Widerspruch und somit in der Optimalität des Algorithmus. $\square$

5.3 Treffpunkt-Problem

Zur Lösung des Treffpunkt-Problems suchen wir nach dem optimalen Treffpunkt einer gegebenen Anzahl an Agenten, die sich zum Startzeitpunkt τ_s an einer Menge an Ausgangsstops N im Netzwerk befinden. Die Optimierung erfolgt dabei nach den Kriterien MinMax und MinSum. Wir haben unterschiedliche Algorithmen entwickelt, um die optimalen Treffpunkte exakt und angenähert zu finden. Zunächst lösen wir das Problem naiv und führen mit dem RAPTOR1 eine Methode ein, um den Treffpunkt nach der minimalen Anzahl an Umstiegen zu optimieren. Anschließend stellen wir zwei Methoden vor, um optimale Treffpunkte mithilfe von unteren Schranken zu finden. Der RAPTOR B Algorithmus arbeitet dabei rundenbasiert, der RAPTOR PQ Algorithmus mit einer Prioritätenwarteschlange. Abschließend lösen wir das Problem für große Anzahlen an Ausgangsstops über die Auswahl einer Teilmenge dieser.

Prioritätenfunktion

Um die Suche nach dem optimalen Treffpunkt im Vergleich zu der naiven Herangehensweise einzuschränken, stellen wir in diesem Kapitel zwei Algorithmen vor, die Prioritätenfunktionen als untere Schranken der optimalen Kosten verwenden. Die Prioritätenfunktionen und zugrunde liegenden Heuristiken definieren wir wie in Kapitel 3.2 beschrieben analog zu Atzmon et al. (2020) in [ALF+20].

Für einen gegebenen Agenten a_i und Stopp v an dem sich dieser Agent befindet, sind die Prioritätenfunktionen für die beiden Optimalitätskriterien MinSum und MinMax wie folgt definiert:

$$\begin{aligned} f_{\text{MinSum}}(a_i, v) &= g(a_i, v) + h_{\text{MinSum}}(a_i, v) \\ f_{\text{MinMax}}(a_i, v) &= \min_{t \in V} \left[\max \left\{ \frac{g(a_i, v) + h_{\text{MinSum}}(a_i, v)}{n} \right\} \right]. \end{aligned}$$

Die benötigte Heuristik $h_{\text{MinSum}}(a_i, v)$ definieren wir als Clique-Heuristik:

$$h_{\text{MinSum}}(a_i, v) = \sum_{\substack{\{v_1, v_2\} \in 2^{S_i(v)} \\ v_1 \neq v_2}} \frac{h_{\text{dur}}(v_1, v_2)}{n - 1}.$$

Dabei sei S_i die Menge der Ausgangsstops abgesehen von dem Ausgangsstopp von a_i vereinigt mit dem Stopp v : $S_i(v) = (N \setminus \{s_i\}) \cup \{v\}$.

Wir berechnen die dazu benötigte Heuristik $h_{\text{dur}}(v_1, v_2)$, die eine untere Schranke für die minimale Fahrtzeit zwischen zwei Stopps darstellt, wie in den Kapiteln 4.2 und 5.1 beschrieben, entweder direkt über die minimale Distanz im zeitunabhängigen Graphen oder unter Berücksichtigung von Landmarken. Wir erstellen den zeitunabhängigen Graphen dabei ungerichtet, um die Korrektheit der Clique-Heuristik zu gewährleisten. Falls wir die Heuristik analog zu Bruns in [Bru24] als die minimale Distanz im zeitunabhängigen Graphen definieren, müssen wir in einer einmaligen Vorberechnungsphase lediglich die Kontraktionshierarchie herstellen. Zu Beginn der Algorithmen kann für alle Ausgangsstops in N die Heuristik zu allen Stopps des Netzwerkes mit dem PHAST Algorithmus berechnet werden. Im Fall unseres Ansatzes über die Landmarken müssen in einer einmaligen Vorberechnungsphase alle Distanzen ausgehend von den Landmarken im zeitunabhängigen Graphen berechnet werden. Diese werden gespeichert und können zur Laufzeit dazu verwendet werden, die Heuristik der Fahrtzeit zwischen zwei Stopps zu berechnen. Um die Anzahl der Vergleiche zu verringern, wählen wir zu Beginn der Algorithmen für jeden Ausgangsstopp in N die Landmarke mit der kleinsten Distanz im zeitunabhängigen Graphen aus. Für die Berechnungen der Heuristiken für den zugehörigen Agenten verwenden wir lediglich diese Landmarke.

Während der Laufzeit der Algorithmen ändern sich an der Clique Heuristik eines Agenten a_i lediglich die Summanden, die den Knoten v enthalten. Die restlichen Werte der Heuristik h_{dur} bleiben konstant und können daher zu Beginn der Algorithmen bereits berechnet und aufsummiert werden.

5.3.1 Naiver Algorithmus

Wir führen zunächst eine naive Methode ein, um die optimalen Treffpunkte gemäß den Optimierungskriterien MinMax und MinSum zu berechnen.

Dazu initialisieren wir für jeden Ausgangsstopp $s \in N$ eines Agenten einen RAPTOR Algorithmus mit der Startzeit τ_s . Diese n RAPTOR Algorithmen führen wir anschließend parallel ohne Zielstopp t durch. So berechnen wir die frühesten Ankunftszeiten für jeden Agenten an jedem Stopp des öffentlichen Verkehrsnetzes ausgehend von τ_s . Wir iterieren nun über alle Stopps und berechnen je nach Optimalitätskriterium den zugehörigen Wert der Kostenfunktion. Für MinSum ist dies die Summe der Fahrtzeiten der einzelnen Agenten und für MinMax die maximale Fahrtzeit dieser. So können wir den Stopp mit dem kleinsten und somit optimalen Wert als Lösung unseres Problems zurückgeben.

Diese Herangehensweise hat den Nachteil, dass die Berechnung ineffizient abläuft, da ausgehend von jedem Ausgangsstopp eine vollständige Suche durchgeführt wird.

5.3.2 RAPTOR1

Der RAPTOR1 Algorithmus optimiert den ermittelten Treffpunkt nicht nach den MinSum oder MinMax Kriterien, sondern nach der maximalen Anzahl an Umstiegen, die einer der Agenten auf seiner Reise vornehmen muss. Für die anderen beiden Kriterien stellt das Ergebnis der RAPTOR1 Berechnung somit eine obere Grenze dar.

Wie bei der naiven Variante initialisieren wir für jeden Agenten einen Standard RAPTOR Algorithmus. Wir führen nun parallel jeden dieser Algorithmen für eine Runde aus und überprüfen anschließend, ob es bereits einen validen Treffpunkt gibt. Ist dies der Fall, terminiert der Algorithmus und gibt je nach Optimalitätskriterium den Treffpunkt mit der minimalen Summe oder der minimalen maximalen Fahrtdauer der letzten Runde zurück. Andernfalls berechnen wir so lange jeweils eine weitere Runde der RAPTOR Algorithmen, bis ein Treffpunkt gefunden wurde.

So ermitteln wir zwar nicht den optimalen Treffpunkt gemäß der MinSum oder MinMax Optimalitätskriterien, aber erhalten schnellstmöglich einen Treffpunkt. Dieser kann für die nachfolgenden Algorithmen als obere Schranke verwendet werden. In der Realität kann das Ergebnis des RAPTOR1 Algorithmus auch seine Anwendung finden, da es von Interesse sein kann, die Treffpunkte nach der minimalen maximalen Anzahl an Umstiegen zu optimieren. Dies resultiert in einfacheren und komfortableren Reisen für die Passagiere.

5.3.3 RAPTOR B

Der RAPTOR B Algorithmus berechnet die optimale Lösung des Treffpunktproblems unter Berücksichtigung der Heuristiken und Prioritätenfunktionen, die wir zu Beginn des Kapitels beschrieben haben. Wir beschreiben zunächst den Algorithmus und zeigen anschließend seine Korrektheit.

Beschreibung

Der RAPTOR B Algorithmus berechnet die optimalen Treffpunkte gemäß der Optimalitätskriterien MinSum und MinMax. Dazu verwenden wir die Prioritätenfunktionen für die Optimalitätskriterien, die die untere Schranken für die optimalen Kosten des Treffpunktproblems darstellen. Sie basieren auf der Clique-Heuristik und unseren Heuristiken für die minimalen Fahrtzeiten zwischen Stopps. Diese können anschließend genutzt werden, um Routen von der Betrachtung auszuschließen und somit den naiven Algorithmus zu beschleunigen.

Falls wir analog zu Bruns in [Bru24] die Heuristiken für die Fahrtzeiten zwischen zwei Stopps direkt über den zeitunabhängigen Graphen berechnen, führen wir zu Beginn des RAPTOR B Algorithmus den PHAST Algorithmus für alle Ausgangsstopps durch. So erhalten wir alle relevanten kürzesten Fahrtauern und müssen keine weiteren Berechnungen durchführen.

Anschließend initialisieren wir für jeden Agenten einen RAPTOR B* Algorithmus (Zeile 4 in Algorithmus 5.2). Diesen modifizieren wir leicht im Vergleich zu der Variante zur Lösung des EAT Problems. So haben wir in diesem Fall keinen Zielstopp, aber speichern uns die Kosten des aktuell besten Ergebnisses gemäß dem gewählten Optimierungskriterium in τ_o ab. Initial setzen wir $\tau_o = \infty$. Im Zuge der Initialisierung berechnen wir zudem den fixen Teil der Clique-Heuristik für jeden Agenten, sodass dies nur einmal erfolgen muss (Z. 5).

Wir führen die RAPTOR B* Algorithmen jeweils parallel für eine Runde durch (Z. 9 – 14). Am Ende jeder Runde iterieren wir über alle Stopps und ermitteln den aktuell besten Treffpunkt (Z. 15). Dessen Summe der Fahrtzeiten oder die maximale Fahrtauer speichern wir abhängig von dem gewählten Optimalitätskriterium anschließend für die RAPTOR B* Algorithmen in τ_o (Z. 11).

Während der Durchführung der RAPTOR B* Algorithmen berechnen wir zu Beginn jeder Runde für jeden seiner markierten Stopps p der vorherigen Runde die Prioritätenfunktion $f(a_i, p)$ mit zugehörigem Agenten a_i . Dazu verwenden wir die unteren Schranken für die Fahrtzeiten, die Clique Heuristik und die jeweiligen Formeln für die zwei Optimalitätskriterien wie zu Beginn des Kapitels beschrieben. Wir wählen dann nur Routen aus, für deren Einstiegsstopps $f(a_i, p) < \tau_o$ gilt.

Während der Iteration über die Stopps einer Route im zweiten Schritt des RAPTOR B* Algorithmus aktualisieren wir die früheste Ankunftszeit eines Stopps nur, falls die zugehörige Fahrtauer kleiner als das bisher beste Ergebnis τ_o ist. Für die MinMax Variante können wir so die neue Fahrtauer mit der minimalen maximalen Fahrtauer des optimalen Treffpunkts vergleichen, da mit ihr kein besseres Ergebnis erzielt werden kann. Dies schränkt die Anzahl der markierten Stopps in der Runde ein. Für MinSum kann dieser Vergleich nur zu der Summe der Fahrtauern hin zum optimalen Treffpunkt erfolgen, da ansonsten keine Aussage über das optimale Ergebnis möglich ist.

Im Vergleich zu dem naiven Algorithmus besitzt der RAPTOR B* Algorithmus den Vorteil, dass weniger Routen expandiert werden. Zudem wird das beste Ergebnis nach jeder Route aktualisiert. Dennoch werden die Routen weiterhin in alle Richtungen expandiert und es kommt zu keinem zielgerichteten Vorgehen.

Algorithmus 5.2 RAPTOR B

```

1: procedure RAPTORB( $source\_stops, \tau_s$ )
2:    $n \leftarrow |source\_stops|$ 
3:   for all  $i \in \{0, \dots, n - 1\}$  do // parallel execution
4:      $rB[i] \leftarrow \text{INITIALIZERAPTORBOUND}(source\_stops[i], \tau_s)$ 
5:      $rB[i].initializeHeuristic()$ 
6:   end for
7:    $best \leftarrow (\infty, \infty)$ 
8:   while not all  $rB[i]$  for  $i \in \{0, \dots, n - 1\}$  finished do
9:     for all  $i \in \{0, \dots, n - 1\}$  do // parallel execution
10:      if  $rB[i]$  is not finished then
11:         $rB[i].setBest(best)$ 
12:         $rB[i].processRaptorBRound()$ 
13:      end if
14:    end for
15:     $best \leftarrow \text{GETOPTIMALMEETINGPOINTANDCOSTS}(rB)$ 
16:  end while
17:  return  $best$ 
18: end procedure

```

Korrektheit

Wir zeigen die Korrektheit des Algorithmus über die beiden folgenden Lemmata.

Lemma 7

(Vollständigkeit). Der RAPTOR B Algorithmus garantiert, falls eine Lösung existiert, diese und andernfalls ∞ zurückzugeben.

Beweis. Der RAPTOR B Algorithmus führt ausgehend von den n Ausgangsstops für jeden Agenten eine Breitensuche über die Routen des öffentlichen Verkehrsnetzes durch. Im schlimmsten Fall expandiert der Algorithmus für jeden Agenten alle erreichbaren Stopps. Falls eine Lösung des Treffpunkt-Problems existiert, finden alle Agenten eine Reise zu dieser und der Algorithmus gibt einen möglichen Treffpunkt zurück. Falls keine Lösung existiert, führen alle Agenten eine maximale Breitensuche durch und finden alle erreichbaren Stopps. Der Algorithmus gibt dann ∞ zurück, da kein übereinstimmender Stopp in all diesen Mengen existiert. $\square$

Lemma 8

(Optimalität). Für eine gegebene zulässige Funktion f garantiert der RAPTOR B Algorithmus den optimalen Treffpunkt zurückzugeben.

Beweis. Der naive RAPTOR Algorithmus zur Treffpunktsuche gibt den optimalen Treffpunkt zurück, da ausgehend von jedem Ausgangsstopp der Agenten eine vollständige Breitensuche

durchgeführt wird. Anschließend wird der bestmögliche Treffpunkt durch eine Iteration über alle Stopps ermittelt. Aus der Zulässigkeit der Funktion f folgt, dass während des RAPTOR B Algorithmus im Vergleich zur naiven Variante nur Routen und zugehörige Stopps aus der Expansion ausgeschlossen werden, ausgehend von denen kein besseres Ergebnis als das bisherige optimale erzielt werden kann. Daraus folgt die Optimalität des Algorithmus. $\square$

5.3.4 RAPTOR PQ

Mit dem RAPTOR PQ Algorithmus lösen wir ebenfalls das Problem der Treffpunktsuche optimal. Dazu verwenden wir bei den einzelnen Suchen der Agenten Prioritätenwarteschlangen, die nach den Prioritätenfunktionen der beiden Optimalitätskriterien sortiert sind. Wie zu Beginn des Kapitels beschrieben verwenden wir dabei die Funktionen, die auf der Clique-Heuristik und unseren unteren Schranken für die Fahrtzeiten zwischen zwei Stopps basieren. Wir beschreiben zunächst den Algorithmus und beweisen anschließend seine Korrektheit.

Beschreibung

Der RAPTOR PQ Algorithmus hat das Ziel, das Treffpunkt-Problem zielgerichteter als mit dem RAPTOR B Algorithmus zu lösen. Dazu benötigen wir zunächst einen möglichen Treffpunkt und führen anschließend einzelne modifizierte RAPTOR PQ* Algorithmen parallel durch.

Analog zum RAPTOR B Algorithmus berechnen wir im Fall der direkten Berechnung über den zeitunabhängigen Graphen die Heuristik h_{dur} mit dem PHAST Algorithmus.

Um einen ersten Treffpunkt zu erhalten und diesen als obere Schranke für das optimale Ergebnis verwenden zu können, führen wir zunächst den RAPTOR1 Algorithmus für das gegebene Treffpunkt-Problem durch. Diese obere Schranke benötigen wir, da wir die einzelnen RAPTOR PQ* Algorithmen im Unterschied zu dem RAPTOR B Algorithmus nicht rundenweise ausführen können, da sie auf Prioritätenwarteschlangen basieren. Eine obere Schranke wird benötigt, um sie mit der unteren Schranke der Prioritätenfunktion f vergleichen zu können. Die schnellste Methode einen validen Treffpunkt und somit eine obere Schranke für die optimalen Kosten des Treffpunkt-Problems zu erhalten, ist die Berechnung dessen über den RAPTOR1 Algorithmus.

Wir modifizieren den RAPTOR PQ* Algorithmus zur Lösung des EAT Problems in der Hinsicht, dass wir keinen Zielstopp definieren. Stattdessen speichern wir je nach gegebenem Optimalitätskriterium die Summe der Fahrtzeiten oder die minimale maximale Fahrtdauer des Ergebnisses des RAPTOR1 Algorithmus als τ_o ab.

Um die n RAPTOR PQ* Algorithmen der n Ausgangsstopps zu initialisieren, berechnen wir zunächst den fixen Teil der Clique-Heuristik für alle Stopps. Danach wandeln wir jeden der n RAPTOR Algorithmen des RAPTOR1 Algorithmus in seinen äquivalenten RAPTOR

PQ* Algorithmus um. Dies erfolgt, indem wir die für die zuletzt markierten Stopps p und zugehörigen Agenten a_i den Wert der Prioritätenfunktion $f(a_i, p)$ berechnen und ihre Routen in die Prioritätenwarteschlange einfügen.

Analog zu den RAPTOR PQ* Algorithmen des EAT Problems nutzen die Algorithmen des Treffpunkt-Problems eine Prioritätenwarteschlange, die nach der Prioritätenfunktion f sortiert ist. Die Algorithmen werden so lange ausgeführt, bis die Warteschlange leer ist oder der f -Wert des ersten Eintrages größer als die obere Schranke ist. Wir fügen zudem nur Routen in die Prioritätenwarteschlange ein, für deren Umstiegsstopps p die Prioritätenfunktion kleiner als die obere Schranke ist. Da wir die modifizierten RAPTOR PQ* Algorithmen nicht rundenbasiert ausführen, aktualisieren wir die obere Schranke im Gegensatz zum RAPTOR B Algorithmus nicht während der Ausführungszeit. Wie beim RAPTOR B Algorithmus aktualisieren wir die früheste Ankunftszeit eines Stopps nur, falls sie kleiner als das bisher beste Ergebnis τ_o ist.

Der Vorteil des RAPTOR PQ Algorithmus ist, dass wir die Routen zielgerichteter in Richtung des vorberechneten ersten Treffpunkts expandieren. Dafür muss die Prioritätenwarteschlange während der Durchführung gefüllt und geleert werden, was bei vielen Einträgen ineffizient werden kann. Zudem wird die obere Schranke während der Ausführung nicht mehr aktualisiert.

Korrektheit

Wir zeigen die Korrektheit des Algorithmus über die beiden folgenden Lemmata.

Lemma 9

(Vollständigkeit). Der RAPTOR PQ Algorithmus garantiert, falls eine Lösung existiert, diese und andernfalls ∞ zurückzugeben.

Beweis. Zu Beginn des RAPTOR PQ Algorithmus zur Lösung des Treffpunktproblems wird der RAPTOR1 Algorithmus ausgehend von allen Ausgangsstopps ausgeführt. Im schlimmsten Fall wird somit für jeden Agenten eine maximale Breitensuche über alle erreichbaren Stopps durchgeführt. Falls keine Lösung für das Treffpunktproblem existiert, findet der RAPTOR1 keinen Treffpunkt und die modifizierten RAPTOR PQ* Algorithmen werden nicht ausgeführt. Der Algorithmus gibt somit ∞ zurück. Falls eine Lösung existiert, garantiert die Definition des RAPTOR1 Algorithmus, dass ein Treffpunkt gefunden wird. Die modifizierten RAPTOR PQ* Algorithmen werden mit diesem als obere Schranke gestartet und geben somit eine Lösung ungleich ∞ zurück. $\square$

Lemma 10

(Optimalität). Für eine gegebene zulässige Funktion f garantiert der RAPTOR PQ Algorithmus den optimalen Treffpunkt der n gegebenen Agenten ausgehend von der Startzeit τ_s zu finden.

Beweis. Angenommen der RAPTOR PQ Algorithmus gibt einen nicht optimalen Treffpunkt t mit Kosten $C(t) > C(t^*)$ für den optimalen Treffpunkt t^* zurück. Da der Algorithmus mit einem nicht optimalen Ergebnis terminiert ist, gilt für mindestens einen der modifizierten RAPTOR PQ* Algorithmen $f_{min} \geq C(t) > C(t^*)$ mit f_{min} als kleinstem f -Wert in der Prioritätenwarteschlange zum Zeitpunkt der Terminierung. Durch die vorzeitige Terminierung, befindet sich die noch nicht expandierte Route r' mit Umstiegsstopp p' , die sich auf dem optimalen Pfad befindet, in der Prioritätenwarteschlange dieses modifizierten RAPTOR PQ Algorithmus. Seien dabei alle Routen mit zugehörigen Umstiegsstopps, die auf der optimalen Reise vor r' und p' liegen, schon expandiert. Daher wurde Eintrag von r' in der Prioritätenwarteschlange von der vorherigen Route auf der optimalen Reise mit dem Wert $f(a_i, p')$ mit Agent a_i generiert. Aus der Zulässigkeit der Funktion f folgt per Definition $f(a_i, p') \leq C(t^*)$. Da $C(t) > C(t^*)$ und $f_{min} > C(t)$ gilt, ist die Abbruchbedingung der Prioritätenwarteschlange durch $f(a_i, p') < f_{min}$ verletzt. Dies resultiert in einem Widerspruch und somit in der Optimalität des Algorithmus. $\square$

5.3.5 RAPTOR SELECT

Das Ausschließen von Routen funktioniert besser für eine kleinere Anzahl an Ausgangsstopps, da dann weniger Möglichkeiten berücksichtigt werden müssen und die Berechnung zielgerichteter erfolgen kann. Da die Ausführungszeit für die bisher vorgestellten Algorithmen linear in der Anzahl der Ausgangsstopps ist und dementsprechend für eine große Anzahl an Agenten zu teuer ist, haben wir den RAPTOR SELECT Algorithmus entwickelt, der eine Teilmenge bestehend aus m Ausgangsstopps mit $m \ll n$ auswählt und lediglich für diese den optimalen Treffpunkt berechnet. Diesen Algorithmus haben wir in drei unterschiedlichen Varianten entwickelt. Die grundlegende Methode und die CANDIDATE-Variante berechnen den Treffpunkt ausschließlich basierend auf den m Ausgangsstopps und nähern den optimalen Treffpunkt somit nur an. So ist der durch sie gefundene Treffpunkt nicht zwangsläufig die optimale Lösung des Treffpunkt-Problems. Im Gegensatz dazu liefert die LOOP-Variante des RAPTOR SELECT Algorithmus über eine iterative Herangehensweise den optimalen Treffpunkt aller gegebenen n Treffpunkte. Sie kann jedoch lediglich für das MinMax Optimalitätskriterium angewandt werden. Nachfolgend beschreiben wir die Vorgehensweise des grundlegenden Algorithmus und seiner beiden Varianten.

Zu Beginn des RAPTOR SELECT Algorithmus wählen wir zunächst m der n Agenten mit $m \ll n$ aus, deren Ausgangsstopps möglichst weit voneinander entfernt liegen. Dabei gehen wir analog zu der Farthest Strategie von Goldmann et al. (2004) in [GH05] für Landmarken vor. Zunächst wird ein zufälliger Ausgangsstopp s_i gewählt und wir berechnen im zeitunabhängigen Graphen des öffentlichen Netzwerkes die minimale Distanz zu allen anderen Ausgangsstopps. Wir wählen den Ausgangsstopp, der am weitesten von s_i entfernt ist, als ersten der m Stopps aus. Anschließend gehen wir iterativ vor, bis wir die m Ausgangsstopps erhalten haben. Dabei berechnen wir jeweils die minimalen Distanzen des zuletzt gewählten Stopps und wählen als

Nächstes den Stopp aus, der von allen bisher gewählten Stopps am weitesten entfernt ist. Wir sortieren diese Fahrtzeiten und wählen die m Agenten mit der größten durchschnittlichen Distanz aus. Anschließend berechnen wir für diese den optimalen Treffpunkt hinsichtlich MinSum oder MinMax. Dieser Treffpunkt stellt eine Annäherung an den optimalen Treffpunkt aller n Agenten dar.

Im Fall des MinMax Optimalitätskriterium lässt sich der gefundene optimale Treffpunkte der gewählten m Ausgangsstopps über eine einmalige Ausführung des klassischen rückwärtsgewandten RAPTOR Algorithmus verifizieren oder falsifizieren. Dazu führen wir ausgehend von dem gefundenen Treffpunkt einen rückwärtsgewandten RAPTOR Algorithmus durch, mit dem wir die spätesten Abfahrtszeiten für alle Ausgangsstopps berechnen, um ausgehend von diesen den Treffpunkt rechtzeitig erreichen zu können. Falls die späteste Abfahrtszeit für mindestens einen Ausgangsstopp vor der minimalen Abfahrtszeit τ_s der Anfrage liegt, ist der gefundene Treffpunkt nicht der optimale. Andernfalls entspricht der optimale Treffpunkt der m Agenten dem optimalen Treffpunkt der n Ausgangsstopps, da ein besserer Treffpunkt auch schon durch die m Agenten gefunden würde. Für MinSum lässt sich das Ergebnis nicht verifizieren, da der Stopp mit der minimalen Summe an Fahrtzeiten für die m Agenten nicht zwangsläufig mit dem der n Agenten übereinstimmt. Hier lässt sich das Ergebnis nur falsifizieren, falls die späteste Abfahrtszeit an einem der Ausgangsstopps vor τ_s liegt.

Diese Eigenschaft für das MinMax Optimalitätskriterium haben wir genutzt, um den optimalen Treffpunkt für eine große Anzahl an Ausgangsstopps möglichst effizient zu berechnen. Der RAPTOR SELECT LOOP Algorithmus wählt analog zu dem RAPTOR SELECT Algorithmus initial m Ausgangsstopps mit der Farthest Strategie aus. Anschließend wiederholen wir die folgenden Schritte iterativ, bis wir den optimalen Treffpunkt gefunden haben. Für die zuletzt gewählten m Ausgangsstopps berechnen wir mit RAPTOR Algorithmen die frühesten Ankunftszeiten an allen Stopps und wählen den optimalen Treffpunkt aller bisher betrachteten Ausgangsstopps aus. Die Berechnung der frühesten Ankunftszeiten kann parallel für die m Agenten erfolgen. Anschließend verifizieren wir den gefundenen Treffpunkt mit einem rückwärtsgewandten RAPTOR Algorithmus. Falls der Treffpunkt valide ist, terminiert der Algorithmus. Andernfalls wählen wir bis zu m Stopps aus, deren späteste Abfahrtszeit, um den Treffpunkt zu erreichen, am weitesten vor der Abfahrtszeit τ_s des Treffpunkt-Problems liegt.

Die Annahme des RAPTOR SELECT LOOP Algorithmus ist, dass die Berechnung der frühesten Ankunftszeiten einer kleinen Menge an Ausgangsstopps und der Verifikation der Treffpunkte dieser ausreichend ist, um den Ausgangsstopp zu ermitteln, der für die längste Fahrtzeit des optimalen Treffpunkts gemäß MinMax verantwortlich ist.

In der CANDIDATE-Variante des RAPTOR SELECT Algorithmus berechnen wir den optimalen Treffpunkt lediglich angenähert. Dabei wählen wir aber nicht nur den einen optimalen Treffpunkt der m Agenten aus, sondern die besten k Punkte gemäß dem Optimalitätskriterium. Anschließend berechnen wir mit $n - m$ RAPTOR Algorithmen für die verbleibenden Agenten die minimalen Ankunftszeiten an den k möglichen Treffpunkten und wählen den besten unter diesen aus. Diese Methode umfasst mehr Berechnungen als der grundlegende RAPTOR SELECT

Algorithmus, aber liefert durch die Auswahl mehrerer Kandidaten eine bessere Annäherung an den optimalen Treffpunkt.

6 Experimente

Wir analysieren die Algorithmen, die wir zur Berechnung von frühesten Ankunftszeiten und optimalen Treffpunkten implementiert haben, in unterschiedlichen Experimenten. Dazu vergleichen wir zunächst die beiden beschriebenen Möglichkeiten untere Schranken für die minimale Fahrtzeit zwischen zwei Stopps zu berechnen, um die geeignetere der beiden auswählen zu können. Anschließend evaluieren wir die neu entwickelten Algorithmen zur Suche nach minimalen Ankunftszeiten im Vergleich zum klassischen RAPTOR Algorithmus. Abschließend messen wir die Ausführungszeiten der Algorithmen zur optimalen Berechnung von Treffpunkten und vergleichen die Ansätze zur approximierten Ermittlung dieser.

Als Grundlage der Experimente dient der komplette Datensatz des deutschen öffentlichen Personenverkehrs. Dieser umfasst 360.347 Stopps, 238.342 Routen, 1.643.799 zugehörige Trips und 2.083.635 Fußwege. Wir haben dabei Fußwege jeweils zwischen zwei Stopps erstellt, die in der Luftlinie 500 m auseinander liegen und anschließend die Transitivität dieser hergestellt. Der zeitunabhängige Graph, der aus diesem Netzwerk erstellt wird, besteht aus 360.347 Knoten und 3.231.711 Kanten.

Wir ermitteln die Parameter der Anfragen für die Experimente jeweils zufällig. Dabei wählen wir die Stopps zufällig aus den Abfahrts- und Ankunftszeiten der Stoppzeiten und nicht direkt aus den Stopps aus, um ihre Relevanz, gemessen durch die Anzahl an Abfahrten an diesem Stopp, zu berücksichtigen. Die Abfahrtszeiten der Anfragen wählen wir so aus, dass übliche Fahrtzeiten wahrscheinlicher sind. Dies resultiert darin, dass nicht überproportionale viele Abfahrtszeiten in der Nacht ausgewählt werden.

Die für die Experimente verwendete Hardware besitzt einen Intel Core i5 mit 6 Kernen, eine 500 GB NVMe Festplatte und 64 GB RAM. Auf ihr läuft Ubuntu 20.04.

6.1 Untere Schranken

Wir vergleichen die Berechnung der unteren Schranken $h_{\text{dur}}(u, v)$ für die Fahrtzeit zwischen zwei Stopps u und v des öffentlichen Verkehrsnetzes über Landmarken, die wir in Kapitel 5.1 vorgestellt haben, mit der Variante der direkten Berechnung über den zeitunabhängigen Graphen von Bruns in [Bru24]. Dazu führen wir ausgehend von 2.500 zufälligen Stopps und Abfahrtszeiten die Berechnung der frühesten Ankunftszeiten über den RAPTOR Algorithmus aus und vergleichen die minimalen Fahrtzeiten zu jeweils vier zufälligen Zielstopps mit den

unteren Schranken für die Fahrtzeiten. In einer einmaligen Vorberechnungsphase haben wir 100 Landmarken mit der Farthest Strategie von Goldberg et al. in [GH05] ausgewählt und die kürzesten Fahrt dauern zu allen Stopps berechnet.

In Tabelle 6.1 sind die absoluten Differenzen in Stunden zwischen den unteren Schranken und den eigentlichen minimalen Fahrtzeiten dargestellt. Wie man erkennen kann, sind die unteren Schranken sowohl im Durchschnitt als auch im Median für die Methode über Landmarken etwas schlechter als die direkte Berechnung über den zeitunabhängigen Graphen. Die absolute Differenz beträgt im Median zwischen 3 und 3,5 Stunden für beide Methoden. Relativ auf die minimale Fahrtzeit gesehen, entspricht diese Differenz etwas mehr als 50% im Median, wie in Tabelle 6.2 zu sehen ist.

Die Unterschiede der so berechneten Heuristik h_{dur} zu den minimalen Fahrtzeiten im zeitabhängigen Kontext lassen sich darüber erklären, dass bei der Berechnung über die zeitunabhängigen Graphen Faktoren wie unregelmäßige Abfahrtszeiten und Umsteigezeiten nicht berücksichtigt werden. Die Konnektivität der Graphen erhöht sich, wenn man dies nicht in die Berechnungen aufnimmt, da dann jederzeit alle Verbindungen und somit alle Kanten für die Berechnung von kürzesten Pfaden verwendet werden können. Dies ist in öffentlichen Verkehrsnetzen durch die Zeitabhängigkeit nicht der Fall. Der Unterschied zwischen den beiden von uns beschriebenen Methoden zur Berechnung der Heuristik lässt sich darüber erklären, dass die Berechnung über die Landmarken lediglich eine untere Schranke für die minimale Distanz in dem zeitunabhängigen Graphen darstellt. Da diese bei der Methode von Bruns in [Bru24] direkt als Heuristik gewählt wird, ist sie in jedem Fall größer gleich der Heuristik über die Landmarken.

Absolute Differenzen (in Stunden)	Zeitunabhängiger Graph	Landmarken
Durchschnitt	6,41	6,70
Median	3,15	3,46

Tabelle 6.1: Absolute Differenzen zwischen den unteren Schranken und den eigentlichen minimalen Fahrtzeiten

Relative Differenzen (in Prozent)	Zeitunabhängiger Graph	Landmarken
Durchschnitt	56,77	61,41
Median	52,10	54,54

Tabelle 6.2: Relative Differenzen zwischen den unteren Schranken und den eigentlichen minimalen Fahrtzeiten in Bezug auf die minimale Fahrt dauer

Ausgehend von diesen Ergebnissen, verwenden wir unsere Methode der Berechnung der unteren Schranken über Landmarken für die Algorithmen zur Ermittlung von frühesten Ankunftszeiten und optimalen Treffpunkten. Sie liefert zwar etwas schlechtere Heuristiken als die alternative Herangehensweise, aber die Differenz der beiden liegt bei lediglich ca. 20 Minuten im Durchschnitt. Im Gegensatz besitzt die Methode über die Landmarken den Vorteil, dass bei dieser Variante weniger Berechnungen zur Laufzeit der Algorithmen durchgeführt

werden, da für die genaue Berechnung im zeitunabhängigen Graphen dann Ausführungen des PHAST Algorithmus notwendig sind.

6.2 Früheste Ankunftszeit

Zur Berechnung der frühesten Ankunftszeiten und ihren zugehörigen kürzesten Reisen haben wir basierend auf dem RAPTOR Algorithmus von Delling et al. (2012) in [DPW12] zwei Algorithmen entwickelt, die über die Verwendung von unteren Schranken für die minimalen Fahrtzeiten gezielt Routen aus der Betrachtung ausschließen, die für keine Verbesserung des optimalen Ergebnisses sorgen. Dabei arbeitet der RAPTOR B* Algorithmus analog zu dem klassischen RAPTOR Algorithmus rundenbasiert. Der RAPTOR PQ* Algorithmus besitzt stattdessen eine Prioritätenwarteschlange. Wir vergleichen die Ausführungszeiten und Anzahl der expandierten Routen der Algorithmen für 2.500 zufällige Anfragen.

Die Ausführungszeiten der Algorithmen sind in der Tabelle 6.3 dargestellt. Die von uns neu entwickelten Algorithmen sind dabei sowohl im Durchschnitt als auch im Median schneller als der klassische RAPTOR Algorithmus. Dabei ist der RAPTOR PQ* Algorithmus mit 591 ms im Durchschnitt schneller als der RAPTOR B* Algorithmus mit 599 ms und der RAPTOR Algorithmus mit 732 ms.

Ausführungszeiten (in ms)	RAPTOR	RAPTOR B*	RAPTOR PQ*
Durchschnitt	732	599	591
Median	757	597	568

Tabelle 6.3: Die Ausführungszeiten der Algorithmen zur Berechnung der frühesten Ankunftszeiten

Die kürzeren Ausführungszeiten unserer Algorithmen lassen sich dadurch erklären, dass über die Verwendung der unteren Schranken für die minimalen Fahrtzeiten Routen ausgeschlossen werden können, über die die aktuell frühestmögliche Ankunftszeit nicht verbessert werden kann. Tabelle 6.4 zeigt, wie viele Routen während der Ausführung der einzelnen Algorithmen betrachtet und expandiert wurden. Wie man erkennen kann, sorgt die Berücksichtigung der unteren Schranken hier für einen deutlichen Vorteil. Besonders gut funktioniert das für den RAPTOR PQ* Algorithmus, bei dem die Berechnung der kürzesten Reise durch die Prioritätenwarteschlange gezielter in Richtung des Zielstopps erfolgt. Hier werden im Durchschnitt lediglich 183.573 statt 499.223 Routen expandiert. Über den RAPTOR B können zwar auch einige Routen ausgeschlossen werden, aber im Durchschnitt werden trotzdem 421.687 Routen betrachtet. Insgesamt besitzen diese beiden neu entwickelten Algorithmen trotzdem eine ähnliche Ausführungszeit, da bei dem RAPTOR PQ Algorithmus die Verwaltung der Prioritätenwarteschlange einen zusätzlichen Aufwand bedeutet.

Anzahl der expandierten Routen	RAPTOR	RAPTOR B*	RAPTOR PQ*
Durchschnitt	499.223	421.687	183.573
Median	516.336	427.089	177.012

Tabelle 6.4: Die Anzahl der expandierten Routen der Algorithmen zur Berechnung der frühesten Ankunftszeiten

Somit können wir mit den unteren Schranken und der Nutzung dieser in unseren neuen Algorithmen die Berechnung der frühesten Ankunftszeiten und zugehörigen Reisen im Vergleich zu dem klassischen RAPTOR Algorithmus deutlich beschleunigen.

6.3 Treffpunkt-Problem

Zur Berechnung von optimalen Treffpunkten gemäß der MinSum und MinMax Optimalitätskriterien im öffentlichen Personenverkehr haben wir unterschiedliche Algorithmen implementiert. Wir analysieren zunächst die Laufzeiten der Algorithmen zur Ermittlung des optimalen Ergebnisses für kleinere Anzahlen an Ausgangstopps. Anschließend evaluieren wir den RAPTOR SELECT Algorithmus, mit dem wir für deutlich mehr Ausgangstopps über eine Auswahl der Ausgangstopps, für die wir die Berechnung durchführen, angenäherte und optimale Treffpunkte berechnen. Wir vergleichen die Algorithmen dabei jeweils mit der naiven Berechnung der Lösung des Problems und führen sie in ihrer parallelen Version aus.

6.3.1 Vollständige Berechnung

Wir evaluieren die Ausführungszeiten der von uns neu entwickelten Algorithmen RAPTOR PQ und RAPTOR B, die alternativ zu dem naiven RAPTOR Algorithmus die optimalen Treffpunkte gemäß MinSum und MinMax berechnen. Wir führen sie dabei auf Anfragen für 2 bis 10 Ausgangstopps aus und vergleichen die Algorithmen zudem mit dem RAPTOR1 Algorithmus, der die Treffpunkte nach der maximalen Anzahl an Umstiegen der Agenten minimiert und somit schnellstmöglich einen Treffpunkt ermittelt. Dieser ist nach den MinSum und MinMax Optimalitätskriterien nicht optimal.

Für die Experimente wählen wir jeweils 1000 Anfragen des Treffpunktproblems zufällig aus und verwenden sie als Eingabe der Algorithmen.

Die Tabelle 6.5 vergleicht die durchschnittlichen Werte der Kostenfunktion für die optimalen Treffpunkte gemäß MinSum und MinMax. Für MinSum entspricht dies der Summe der Fahrtzeiten der Agenten zum optimalen Treffpunkt. Diese ist damit abhängig von der Anzahl der Ausgangstopps und steigt von 332 Minuten (etwa 5, 5 Stunden) für zwei Ausgangstopps bis zu 2.265 Minuten (etwa 37, 75 Stunden) für 10 Ausgangstopps an. Die Kostenfunktion des MinMax Kriteriums entspricht der maximalen Fahrtzeit der Agenten. Sie hat somit eine

kleinere Abhängigkeit von der Anzahl der Ausgangsstops und variiert lediglich zwischen 208 (etwa 3, 5 Stunden) und 436 Minuten (etwa 7, 3 Stunden).

Durchschnittliche Kostenfunktion (in min) für #Ausgangsstops	MinSum	MinMax
2	332	208
3	566	271
5	1.041	333
7	1.504	363
9	1.990	401
10	2.265	436

Tabelle 6.5: Der durchschnittliche Wert der Kostenfunktion des optimalen Treffpunkts gemäß MinSum und MinMax in Minuten. Für MinSum entspricht dies der Summe der Fahrtzeiten der Agenten zum optimalen Treffpunkt, für MinMax der maximalen Fahrtzeit der Agenten.

In Tabelle 6.6 sind die durchschnittlichen Ausführungszeiten (in ms) der vier Algorithmen zur Berechnung der optimalen Treffpunkte gemäß dem MinSum Optimalitätskriterium gezeigt. Für 2 bis 5 Ausgangsstops ist dabei die Ermittlung der Treffpunkte über die neu entwickelten Algorithmen schneller als die naive Berechnung. Besonders der RAPTOR B Algorithmus liefert dabei eine deutliche Beschleunigung gegenüber der naiven Ausführung, bei der für jeden Ausgangsstopp ein vollständiger RAPTOR Algorithmus durchgeführt wird. Für mehr als 5 Ausgangsstops benötigen die beiden neuen Algorithmen im Durchschnitt mehr Zeit zur Berechnung der optimalen Treffpunkte. Durch die höhere Anzahl der Ausgangspunkte sind diese potenziell weiter über das gesamte öffentliche Verkehrsnetz verteilt. Dies sorgt dafür, dass mehr Stopps als mögliche optimale Treffpunkte infrage kommen und somit das Ausschlussverfahren unserer Algorithmen über die unteren Schranken weniger Routen exkludiert. Bei dem MinSum Optimalitätskriterium steigen zudem die durchschnittlichen Kosten der optimalen Lösung für die größere Anzahl an Ausgangsstops, da die einzelnen Fahrtzeiten addiert werden. Dies resultiert in höheren Differenzen zu der Heuristik der RAPTOR B und RAPTOR PQ Algorithmen, da diese über die unteren Schranken der Fahrtzeiten ermittelt werden. Das führt auch dazu, dass weniger Routen ausgeschlossen werden und die Algorithmen für mehr Ausgangsstops ineffizienter werden. Ein weiterer Faktor, der für die längeren Ausführungszeiten mitverantwortlich ist, ist die quadratische Eigenschaft der Clique-Heuristik. So steigt die Anzahl der Berechnungen der Heuristiken zwischen den Ausgangsstops quadratisch in der Anzahl dieser.

In Tabelle 6.7 befinden sich analog dazu die durchschnittlichen Ausführungszeiten (in ms) der Algorithmen, die die optimalen Treffpunkte für das MinMax Optimalitätskriterium bestimmen. Dabei sind die beiden neu entwickelten Algorithmen für 2 bis 10 Ausgangsstops schneller als der naive Algorithmus. Besonders der RAPTOR B Algorithmus liefert eine deutliche Beschleunigung, die für $n = 3$ Ausgangsstops einen Faktor von 5 und für $n = 7$ immer noch einen

Durchschn. Ausführungszeiten (in ms) für #Ausgangsstopps	RAPTOR1	Naiver RAPTOR	RAPTOR PQ	RAPTOR B
2	10	1.484	774	381
3	24	1.636	1.285	895
5	63	1.874	1.731	1.740
7	112	2.901	3.365	2.841
9	156	3.406	3.591	3.468
10	177	3.571	3.730	3.673

Tabelle 6.6: Die durchschnittlichen Ausführungszeiten für die Algorithmen zur Berechnung der optimalen Treffpunkte gemäß dem MinSum Optimalitätskriterium für unterschiedliche Anzahlen an Ausgangsstopps

Faktor von 3 hat. Der RAPTOR PQ Algorithmus ist zwar schneller als der naive Algorithmus, aber langsamer als der RAPTOR B Algorithmus. Dies lässt sich damit erklären, dass er die Routen zwar zielgerichteter zu potenziellen optimalen Treffpunkten expandiert, aber zusätzlich die Prioritätenwarteschlangen verwalten muss. Das zielgerichtete Expandieren liefert auch einen kleineren Vorteil als bei der Berechnung der frühesten Ankunftszeit über den RAPTOR PQ* Algorithmus, da das eigentliche Ziel nicht bekannt ist. Als Zielvorgabe wird hier lediglich der mögliche Treffpunkt verwendet, der aus dem RAPTOR1 Algorithmus resultiert. Dieser muss ausgehend von den Ausgangsstopps nicht zwangsläufig die gleiche Richtung wie der optimale Treffpunkt des MinMax Kriteriums besitzen.

Durchschn. Ausführungszeiten (in ms) für #Ausgangsstopps	RAPTOR1	Naiver RAPTOR	RAPTOR PQ	RAPTOR B
2	10	1.484	571	176
3	24	1.636	868	344
5	63	1.874	1.181	638
7	112	2.901	2.066	1.057
9	156	3.406	2.365	1.495
10	177	3.571	2.495	1.656

Tabelle 6.7: Die durchschnittlichen Ausführungszeiten für die Algorithmen zur Berechnung der optimalen Treffpunkte gemäß dem MinMax Optimalitätskriterium für unterschiedliche Anzahlen an Ausgangsstopps

Die Abbildung 6.1 zeigt die durchschnittlichen Ausführungszeiten der Algorithmen für beide Optimalitätskriterien im Vergleich. Dabei kann man erkennen, dass die Berechnung der optimalen Treffpunkte gemäß MinMax für mehr als 2 Ausgangsstopps schneller als für MinSum ist. Dies lässt sich einerseits damit begründen, dass die Kosten der optimalen Lösung bei MinSum deutlich höher sind, da diese aufsummiert werden. Dies resultiert in höheren Differenzen zu der Heuristik der Algorithmen und somit in weniger ausgeschlossenen Routen. Zudem kann

nur bei dem MinMax Kriterium die zielgerichtete Vereinfachung (target pruning) effektiv eingesetzt werden. Dabei werden früheste Ankunftszeiten nicht aktualisiert und ihre zugehörigen Routen nicht mehr berücksichtigt, sobald sie größer ist als die maximale Ankunftszeit des aktuell optimalen Ergebnisses. Dies ist bei MinSum nicht möglich, da wir hier nicht nach der maximalen Ankunftszeit, sondern der Summe der Fahrtzeiten den Treffpunkt optimieren.

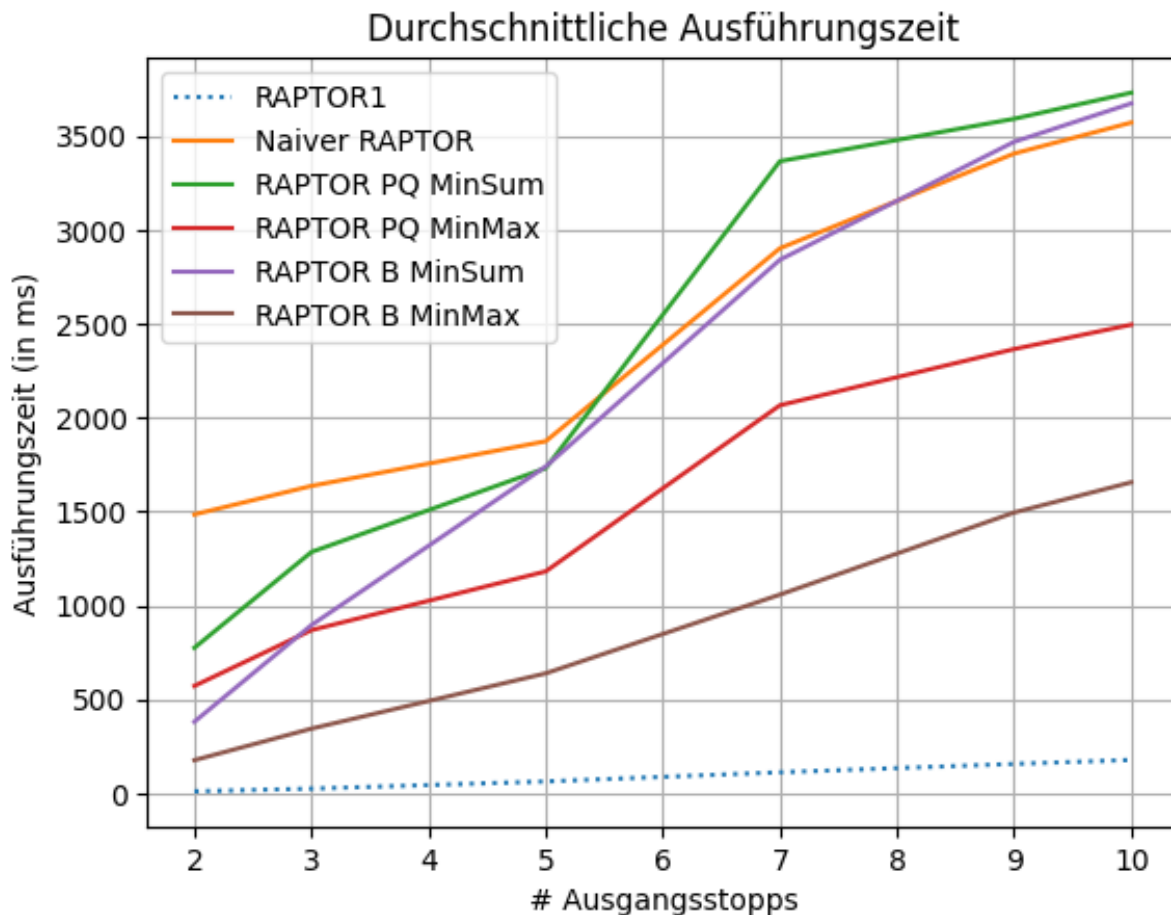


Abbildung 6.1: Die durchschnittlichen Ausführungszeiten der RAPTOR Algorithmen zur Berechnung der optimalen Treffpunkte gemäß MinSum und MinMax.

Die durchschnittliche Anzahl der expandierten Routen pro Ausgangsstopp der Anfrage zur Berechnung der optimalen Treffpunkte unterstützt diese Analyse. In Tabelle 6.8 sind die Anzahlen der expandierten Routen für das MinSum und in Tabelle 6.9 für das MinMax Optimalitätskriterium dargestellt. Abbildung 6.2 zeigt die Werte im Vergleich.

Wie man erkennen kann, ist die Anzahl der expandierten Routen bei der naiven Berechnung der optimalen Treffpunkte nicht von der Anzahl der Ausgangsstops abhängig, da wir hier keine Routen über eine Heuristik exkludieren. Dies erfolgt lediglich bei den neu entwickelten RAPTOR PQ und RAPTOR B Algorithmen. Für kleinere Werte von n können auch bei dem Min-

Sum Optimalitätskriterium über den RAPTOR B Algorithmus so viele Routen ausgeschlossen werden, dass dieser der effizienteste Algorithmus ist. Durch die deutlich ansteigenden Kosten der Optimalitätsfunktion für mehr Ausgangsstops nähert sich die Anzahl der expandierten Routen der naiven Lösung an. Dies resultiert in deutlich langsameren Ausführungszeiten. Im Vergleich dazu können über das zielgerichtete Vorgehen des RAPTOR PQ Algorithmus mehr Routen ausgeschlossen werden. Dies zeigt sich besonders für das MinMax Optimalitätskriterium, bei der die Anzahl der expandierten Routen $\frac{1}{4}$ der expandierten Routen der naiven Herangehensweise entspricht. Dies resultiert in einer schnelleren Ausführungszeit, da wir während der Durchführung der einzelnen RAPTOR Algorithmen weniger Berechnungen durchführen müssen. Auch hier werden für kleinere n zunächst weniger Routen während des RAPTOR B Algorithmus expandiert und für größere dominiert der RAPTOR PQ Algorithmus.

Insgesamt zeigt sich, dass die Herangehensweise über das Exkludieren von Routen erfolgreich die Ausführungszeiten im Vergleich zu dem naiven Algorithmus reduziert. Durch die weitere Verteilung der Ausgangsstops im öffentlichen Verkehrsnetz und die wachsenden Kostenfunktionen, ist es jedoch besonders effizient für kleinere Werte von n und die Optimierung gemäß MinMax.

Durchschn. #Expandierte Routen für #Ausgangsstopps	RAPTOR1	Naiver RAPTOR	RAPTOR PQ	RAPTOR B
2	2.992	852.520	214.418	191.524
3	7.424	850.638	331.326	498.503
5	18.040	855.135	414.448	725.175
7	25.910	855.804	425.649	800.588
9	31.508	854.617	425.605	816.530
10	32.964	853.039	426.862	821.973

Tabelle 6.8: Die durchschnittlichen Anzahlen an expandierten Routen pro Nummer an Ausgangsstops während den Algorithmen zur Berechnung der optimalen Treffpunkte gemäß dem MinSum Optimalitätskriterium für unterschiedliche Anzahlen an Ausgangsstops

Der RAPTOR1 Algorithmus führt für jeden Ausgangsstopp parallel jeweils einen klassischen RAPTOR Algorithmus durch. Nach jeder Runde der einzelnen Algorithmen wird überprüft, ob bereits ein Treffpunkt der Agenten gefunden wurde. Sobald dies erreicht ist, terminiert der Algorithmus und berechnet somit schnellstmöglich einen Treffpunkt, der zudem gemäß der maximalen Anzahl an Umstiegen der Agenten optimal ist. Die durchschnittlichen Ausführungszeiten liegen zwischen 9 ms und 173 ms für 2 bis 10 Ausgangsstops und sind somit deutlich geringer als die der Algorithmen zur Berechnung der optimalen Treffpunkte gemäß MinSum und MinMax. Es ist daher möglich, den RAPTOR1 Algorithmus zu Beginn des RAPTOR PQ Algorithmus zu verwenden, da über ihn effizient ein valider Treffpunkt gefunden werden kann. Dieser kann anschließend bei dem zielgerichteten Expandieren der Routen als obere Schranke berücksichtigt werden.

Durchschn. #Expandierte Routen für #Ausgangsstopps	RAPTOR1	Naiver RAPTOR	RAPTOR PQ	RAPTOR B
2	2.992	852.520	154.092	82.705
3	7.424	850.638	219.602	146.377
5	18.040	855.135	265.178	233.783
7	25.910	855.804	273.776	291.536
9	31.508	854.617	276.454	336.373
10	32.964	853.039	279.265	349.863

Tabelle 6.9: Die durchschnittlichen Anzahlen an expandierten Routen pro Nummer an Ausgangsstopps während den Algorithmen zur Berechnung der optimalen Treffpunkte gemäß dem MinMax Optimalitätskriterium für unterschiedliche Anzahlen an Ausgangsstopps

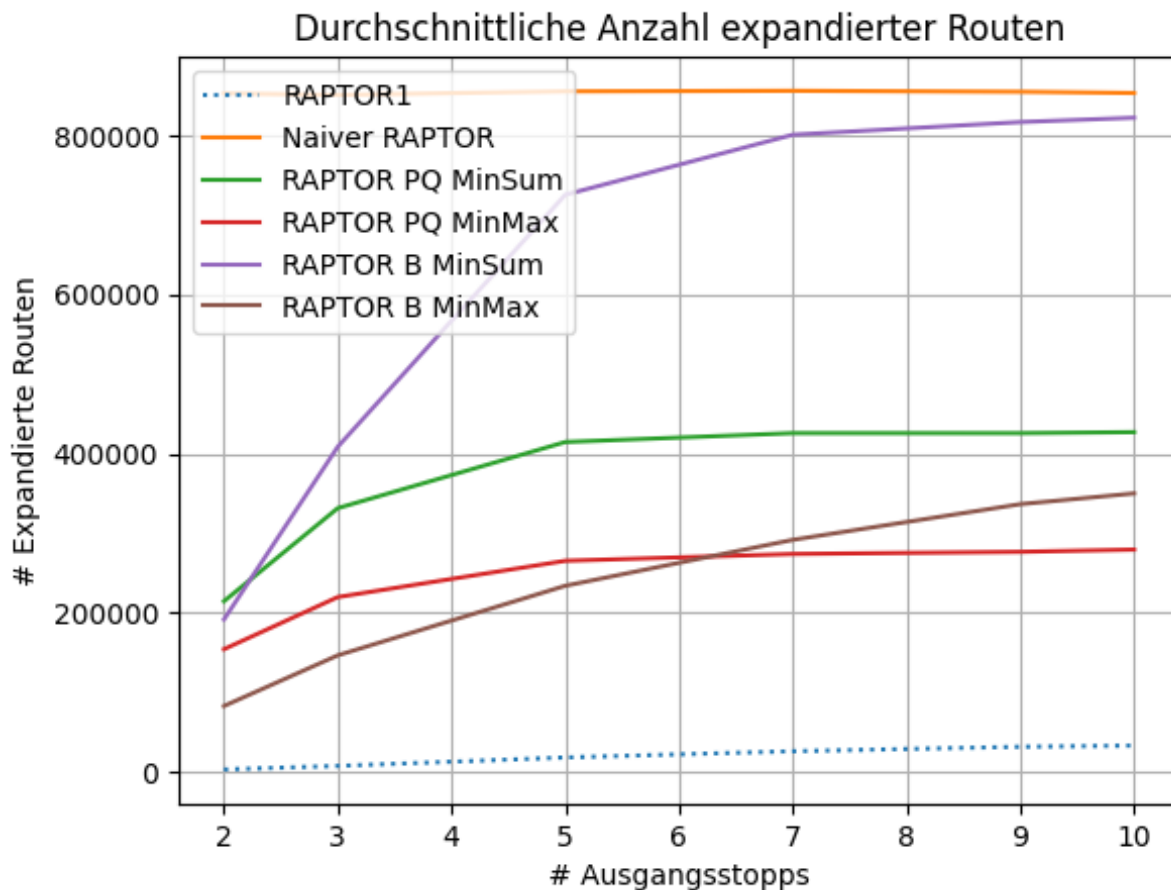


Abbildung 6.2: Die durchschnittlichen Anzahlen der expandierten Routen pro Ausgangsstopps der RAPTOR Algorithmen zur Berechnung der optimalen Treffpunkte gemäß MinSum und MinMax.

Um die Qualität der Ergebnisse des RAPTOR1 Algorithmus zu evaluieren, bestimmen wir zunächst ihre Genauigkeit. Diese gibt in Prozent den Anteil der Anfragen an, bei der die Kostenfunktion des Treffpunkts des RAPTOR1 Algorithmus mit der optimalen Lösung gemäß MinSum oder MinMax übereinstimmt. Tabelle 6.10 zeigt die Resultate dieser Analyse. Wie man erkennen kann, sinkt die Genauigkeit für eine größere Anzahl an Ausgangsstops, da durch die weitere Verteilung der Ausgangsstops längere und komplexere Reisen zum optimalen Treffpunkt führen. Diese werden selten durch eine minimale Anzahl an Umstiegen erreicht. Insgesamt liegt die Genauigkeit bei MinSum maximal bei 7,3% und für MinMax bei 13,2%. Somit können die Ergebnisse als ersten Anhaltspunkt für den RAPTOR PQ Algorithmus verwendet werden, aber sollten nicht direkt als Treffpunkt gewählt werden, obwohl sie in einfacheren Reisen für die Agenten resultieren.

Genauigkeit (in Prozent) für #Ausgangsstops	MinSum	MinMax
2	5,7	13,2
3	7,3	7,6
5	5,2	6,6
7	3,6	6,6
9	1,5	5,2
10	1,5	4,9

Tabelle 6.10: Die Genauigkeit des RAPTOR1 Algorithmus. Diese definiert sich über den Anteil der Anfragen, bei denen der Treffpunkt des RAPTOR1 Algorithmus mit dem optimalen gemäß MinSum und MinMax übereinstimmt.

Tabelle 6.11 zeigt den Prozentsatz der Ergebnisse des RAPTOR1 Algorithmus, bei dem die Kosten der Lösung relativ gesehen um maximal 10% von den Kosten des optimalen Treffpunkts differiert. Somit werden Treffpunkte toleriert, die lediglich eine kleine Abweichung zu der optimalen Lösung darstellen. Für MinSum steigt dabei die Genauigkeit auf bis zu 57,3% an. Im Fall des MinMax Optimalitätskriterium kann maximal eine Genauigkeit von 30,4% erzielt werden.

6.3.2 Selektive Berechnung

Wie die Experimente zur Berechnung der optimalen Treffpunkte mit unseren neu entwickelten Algorithmen gezeigt haben, ist diese lediglich für kleinere Anzahlen an Ausgangsstops effizient. Dies lässt sich damit erklären, dass die Laufzeit der Algorithmen linear mit der Anzahl der Ausgangsstops wächst. Dies resultiert daraus, dass wir für jeden Ausgangsstopp eine Variante des RAPTOR Algorithmus durchführen. Für Anfragen mit mehr als 10 Ausgangsstops haben wir daher den RAPTOR SELECT Algorithmus in unterschiedlichen Versionen entwickelt. Die Idee dabei ist lediglich für eine Auswahl an Ausgangsstops, die sich möglichst verteilt im öffentlichen Verkehrsnetz befinden, den optimalen Treffpunkt zu berechnen. Wir wählen dabei

Genauigkeit (in Prozent) mit Abweichung $\leq 10\%$ für #Ausgangsstopps	MinSum	MinMax
2	34,9	25,7
3	28,7	20,6
5	37,5	26,3
7	45,7	27,7
9	53,2	27,3
10	57,3	30,4

Tabelle 6.11: Der Anteil an Anfragen des RAPTOR1 Algorithmus, bei denen die Differenz der Kosten zum optimalen Treffpunkt kleiner gleich 10% der Fahrzeit dessen ist.

jeweils $m = 6$ Ausgangsstopps über die Farthest Strategie im zeitunabhängigen Graphen aus. Wir beschränken uns dabei auf 6 Ausgangsstopps, da die Berechnung des optimalen Treffpunkts dieser Stopps parallel auf den 6 Kernen unserer Hardware ausgeführt werden kann. Der über die Berechnung der frühesten Ankunftszeiten dieser m Ausgangsstopps ermittelte Treffpunkt kann im Fall des MinMax Optimalitätskriterium für alle Ausgangsstopps validiert und iterativ in der LOOP-Variante des Algorithmus der optimale Treffpunkt gefunden werden. Die CANDIDATE-Variante (CAND) wählt die besten $k = 25$ Treffpunkte für die Auswahl an Ausgangsstopps aus und ermittelt über die Berechnung der frühesten Ankunftszeiten an diesen ausgehend von allen anderen Stopps den besten Treffpunkt. Die minimalen Ankunftszeiten werden dabei mit dem RAPTOR PQ* Algorithmus berechnet. Da wir über diese Herangehensweise nicht garantiert optimale Ergebnisse erhalten, handelt es sich bei den Resultaten des RAPTOR SELECT Algorithmus lediglich um eine Annäherung dessen. Eine Ausnahme stellt dabei die LOOP-Variante dar, da wir diese iterativ so lange ausführen, bis wir das optimale Ergebnis erhalten. Im Folgenden analysieren wir die Ausführungszeiten des Algorithmus und die Genauigkeit der ermittelten Treffpunkte für 25 bis 150 Ausgangsstopps.

Für die Experimente wählen wir jeweils 100 Anfragen des Treffpunktproblems zufällig aus und verwenden sie als Eingabe der Algorithmen.

Die Tabelle 6.12 enthält die Ausführungszeiten des RAPTOR SELECT Algorithmus und seiner CANDIDATE-Variante im Vergleich zu der naiven Berechnung des optimalen Treffpunkts gemäß dem MinSum Optimalitätskriterium. Mit dem RAPTOR SELECT Algorithmus werden Treffpunkte für maximal 150 Ausgangsstopps in 4 bis 6 Sekunden berechnet. Dies stellt im Vergleich zu der Ausführung des naiven Algorithmus, der im Durchschnitt bis zu 45 Sekunden für 150 Ausgangsstopps benötigt, eine deutliche Beschleunigung dar. Die CANDIDATE-Variante des Algorithmus benötigt durch die zusätzlichen Berechnungen der frühesten Ankunftszeiten für die verbleibenden Ausgangsstopps mehr Zeit. So steigen die Ausführungszeiten hier von 6 Sekunden für 25 Ausgangsstopps auf bis zu 25 Sekunden im Durchschnitt für 150 Stopps.

Die Tabelle 6.13 zeigt die Ausführungszeiten für den RAPTOR SELECT Algorithmus und seiner Varianten gemäß dem MinMax Optimalitätskriterium. Der grundlegende RAPTOR

Durchschn. Ausführungszeiten (in ms) für # Ausgangsstops	Naive RAPTOR	RAPTOR SELECT	RAPTOR SELECT CAND
25	8.088	4.393	6.430
50	15.292	4.788	9.280
75	22.729	5.181	14.054
100	29.901	5.372	17.980
150	44.058	5.679	25.674

Tabelle 6.12: Die Ausführungszeiten im Durchschnitt für die Algorithmen zur selektiven Berechnung der optimalen Treffpunkte im Durchschnitt gemäß dem MinSum Optimalitätskriterium für unterschiedliche Anzahlen an Ausgangsstops im Vergleich zur naiven Berechnung der Treffpunkte.

SELECT Algorithmus sorgt auch hier mit einer Ausführungszeit von 3 bis 5 Sekunden für eine deutliche Beschleunigung mit Faktor bis zu 10 im Vergleich zu der Ausführung des naiven Algorithmus. Die CANDIDATE-Variante benötigt im Durchschnitt bis zu 28 Sekunden für 150 Ausgangsstops. Da wir für das MinMax Optimalitätskriterium über eine einzelne Ausführung des klassischen rückwärtsgerichteten RAPTOR Algorithmus einen gefundenen Treffpunkt hinsichtlich seiner Optimalität validieren können, können wir iterativ immer mehr Ausgangsstops berücksichtigen, um den Optimalen zu finden. Dazu wählen wir in der LOOP-Variante nach jeder Runde bis zu 6 neue Stops für die genaue Berechnung des optimalen Treffpunkts aus, bis wir den optimalen Treffpunkt gefunden haben. Im Durchschnitt beträgt die Ausführungszeit des LOOP Algorithmus etwa 7 Sekunden für 25 Ausgangsstops und lediglich 10, 5 s für 150 Ausführungsstops. Im Vergleich zu der naiven Suche nach dem optimalen Treffpunkt liefert sie dabei ab 50 Ausgangsstops eine Beschleunigung der Ausführungszeit mindestens um den Faktor 2. Für 150 Ausgangsstops ist die Ausführungszeit des naiven Algorithmus viermal so lange wie die des LOOP-Algorithmus.

Die Abbildung 6.3 zeigt die Ausführungszeit der approximierten Algorithmen für beide Optimalitätskriterien im Vergleich. Dabei lässt sich erkennen, dass die Berechnung der Treffpunkte über die Auswahl von Ausgangsstops einen deutlichen Vorteil im Vergleich zu dem naiven Algorithmus hinsichtlich der Ausführungszeiten liefert. Die Ausführungszeiten der CANDIDATE-Variante steigen ebenfalls linear in Abhängigkeit der Anzahl der Ausführungsstops an. Im Gegensatz dazu sind die Ausführungszeiten der LOOP-Variante im Durchschnitt beschränkt und sind deutlich geringer als die Werte der naiven Suche. Die Berechnung der optimalen Treffpunkte über den RAPTOR SELECT LOOP Algorithmus liefert somit einen deutlichen Vorteil im Vergleich zu der naiven Suche.

Um die Qualität der approximierten Ergebnisse des RAPTOR SELECT Algorithmus und seiner CANDIDATE-Variante zu analysieren, betrachten wir zunächst die Genauigkeit der Treffpunkte des MinSum Optimalitätskriterium. Diese gibt in Prozent den Anteil der Anfragen an, für die die Kostenfunktion der approximierten Lösung mit der Optimalen übereinstimmt. Für die klassische Variante beträgt diese für 25 bis 150 Ausgangsstops 12 – 21% und lässt sich somit

Durchschn. Ausführungszeiten (in ms) für # Ausgangsstops	Naive RAPTOR	RAPTOR SELECT	RAPTOR SELECT CAND	RAPTOR SELECT LOOP
25	8.088	3.278	5.230	6.879
50	15.292	3.821	8.504	8.336
75	22.729	4.107	13.873	8.698
100	29.901	4.459	18.460	9.514
150	44.058	4.712	27.695	10.594

Tabelle 6.13: Die Ausführungszeiten im Durchschnitt für die Algorithmen zur selektiven Berechnung der optimalen Treffpunkte gemäß dem MinMax Optimalitätskriterium für unterschiedliche Anzahlen an Ausgangsstops im Vergleich zur naiven Berechnung der Treffpunkte.

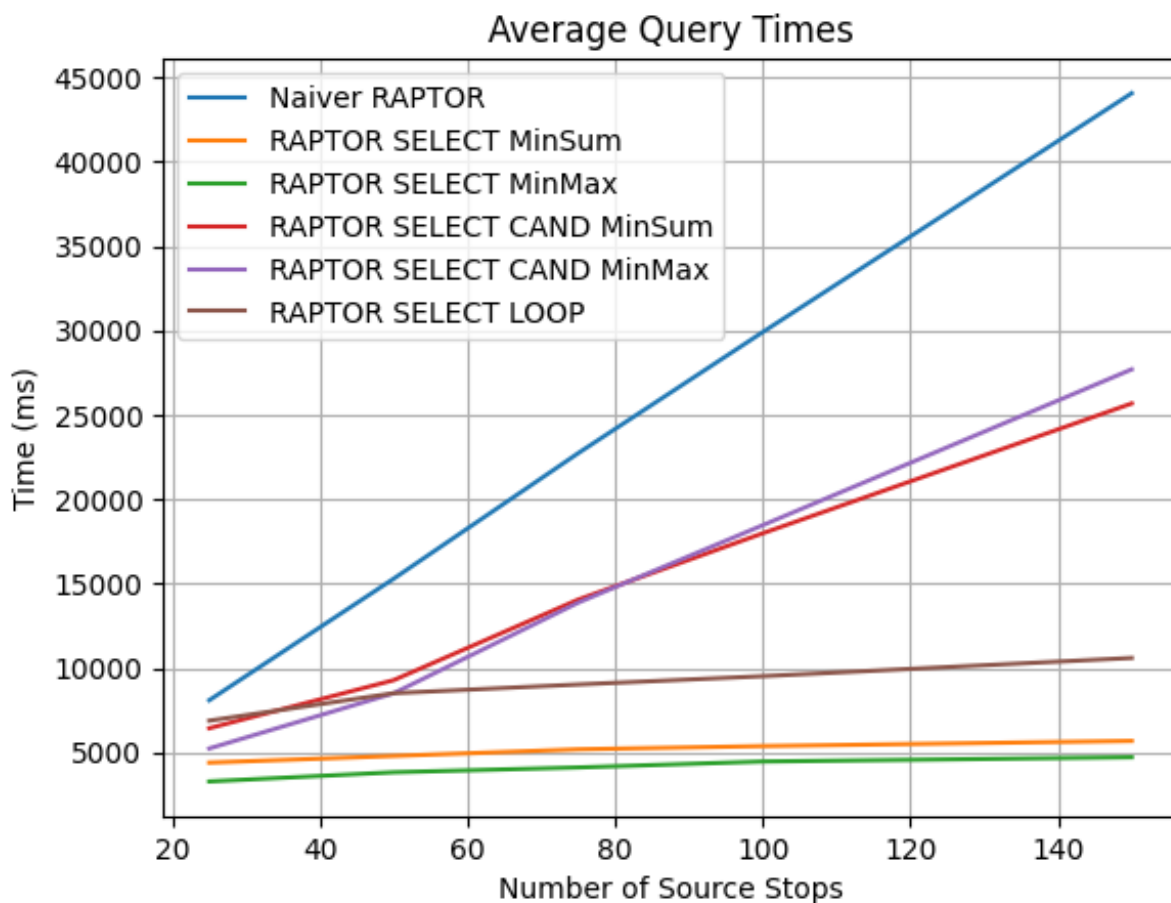


Abbildung 6.3: Die Ausführungszeiten der RAPTOR SELECT Algorithmen im Durchschnitt zur Berechnung der optimalen Treffpunkte gemäß MinSum und MinMax.

nicht zur Berechnung von optimalen Treffpunkten verwenden. Selbst mit der CANDIDATE-Option wird lediglich eine Genauigkeit von bis zu 32% für 100 Ausgangsstops erreicht. Dies lässt sich damit erklären, dass für die Berechnung des optimalen Treffpunkts gemäß dem MinSum Kriterium mehr als eine kleine Auswahl an Ausgangsstops von Bedeutung ist. Das resultiert daraus, dass sich die Kostenfunktion aus der Summe der Reisezeit aller Agenten zusammensetzt.

Genauigkeit (in Prozent) für #Ausgangsstops	RAPTOR SELECT	RAPTOR SELECT CAND
25	21	30
50	12	28
75	12	27
100	12	32
150	14	27

Tabelle 6.14: Die Genauigkeit der RAPTOR SELECT Algorithmen für das MinSum Optimalitätskriterium. Sie definiert sich über den Anteil der Anfragen, bei denen die Kosten des gefundenen Treffpunkts mit den Optimalen übereinstimmen.

Im Vergleich dazu, ist die Kostenfunktion des MinMax Kriteriums von der maximalen Fahrtzeit eines Agenten abhängig. Somit sorgt die genaue Berechnung der Treffpunkte für die Agenten, die sich am weitesten auseinander befinden, dafür, dass der optimale Treffpunkt gefunden werden kann, indem nur eine kleine Auswahl der Ausgangsstops betrachtet wird. Über die einmalige Auswahl der m Ausgangsstops im RAPTOR SELECT Algorithmus und seiner CANDIDATE-Variante lässt sich dennoch lediglich eine Genauigkeit von bis zu 34% erzielen, wie in Tabelle 6.15 zu sehen.

Genauigkeit (in Prozent) für #Ausgangsstops	RAPTOR SELECT	RAPTOR SELECT CAND
25	33	34
50	25	29
75	25	25
100	20	23
150	14	15

Tabelle 6.15: Die Genauigkeit der RAPTOR SELECT Algorithmen für das MinMax Optimalitätskriterium. Sie definiert sich über den Anteil der Anfragen, bei denen die Kosten des gefundenen Treffpunkts mit den Optimalen übereinstimmen.

Die entscheidende Eigenschaft, die wir im Fall des MinMax Kriteriums für große Anzahlen an Ausgangsstops, nutzen können, liegt in der einfachen Verifizierbarkeit des gefundenen

Treffpunkts über eine einmalige Ausführung des rückwärtsgerichteten RAPTOR Algorithmus. Die Tabelle 6.16 zeigt die Anzahl der Ausgangsstopps, für die eine genaue Berechnung des optimalen Treffpunkts durchgeführt wird. Im Durchschnitt werden dabei für 25 bis 150 Ausgangsstopps lediglich 8 bis 10, 5 Ausgangsstopps benötigt, für die wir die frühesten Ankunftszeiten an allen Stopps berechnen müssen, bis wir den optimalen Treffpunkt gefunden haben. Selbst im schlechtesten Fall werden maximal die Berechnungen für 17 Ausgangsstopps benötigt, um den Treffpunkt zu finden. Dies ist von besonderer Bedeutung, da dies zeigt, dass wir für jede Anfrage den optimalen Treffpunkt selbst für eine große Anzahl an Ausgangsstopps effizient berechnen können, da eine Auswahl von wenigen Ausgangsstopps ausreicht. Für $m = 6$ können alle benötigten frühesten Ankunftszeiten in drei Runden des Algorithmus durch jeweilige parallele Ausführungen des RAPTOR Algorithmus berechnet werden. Im Gegensatz zu den anderen Algorithmen zur Lösung des Treffpunkt-Problems, weist der RAPTOR SELECT LOOP Algorithmus keine lineare Abhängigkeit von der Anzahl der Ausgangsstopps auf. Dies resultiert darin, dass er auch für sehr große n effizient ausgeführt werden kann.

# Genaue Berechnungen für # Ausgangsstopps	Durchschnitt	Maximum
25	8, 2	14
50	9, 5	17
75	9, 4	16
100	9, 8	17
150	10, 5	17

Tabelle 6.16: Die Anzahl der Ausgangsstopps, für die während des RAPTOR SELECT LOOP Algorithmus die frühesten Ankunftszeiten berechnet und somit bei der Berechnung des optimalen Treffpunkts gemäß dem MinMax Kriterium berücksichtigt werden.

Um diese Eigenschaft noch weiter zu analysieren, haben wir abschließend den RAPTOR SELECT LOOP Algorithmus mit 500 bis 1000 Ausgangsstopps getestet, um seine Grenzen zu evaluieren. Die Tabelle 6.17 enthält die Ausführungszeiten des Algorithmus im Durchschnitt. Im Vergleich sind die geschätzten Werte (mit * markiert) des naiven RAPTOR Algorithmus dargestellt. Eine Ausführung des Algorithmus benötigt zu lange, aber durch die lineare Eigenschaft lässt sich seine Laufzeit gut abschätzen. Der iterative RAPTOR SELECT LOOP Algorithmus liefert selbst für diese Anzahlen an Ausgangsstopps verhältnismäßig kurze Ausführungszeiten von 18 bis 26 Sekunden im Durchschnitt. Diese stellen eine Beschleunigung der Ausführungszeit des naiven Algorithmus um den Faktor von bis zu 10 dar.

Dies lässt sich erneut mit der kleinen Anzahl an Ausgangsstopps, die zur Berechnung der optimalen Treffpunkte entscheidend sind, erklären. Die Tabelle 6.18 zeigt im Durchschnitt und Maximum die Anzahl Ausgangsstopps, für die eine genaue Berechnung der frühesten Ankunftszeiten durchgeführt wird. Selbst für 1000 Ausgangsstopps werden im Durchschnitt lediglich 11, 8 Ausgangsstopps in zwei Runden des iterativen Algorithmus benötigt, um den

Ausführungszeiten (in ms) für #Ausgangsstopps	Naive RAPTOR	RAPTOR SELECT LOOP
500	150.000*	18.133
1000	300.00*	26.147

Tabelle 6.17: Die Ausführungszeiten des RAPTOR SELECT LOOP Algorithmus im Durchschnitt für das MinMax Optimalitätskriterium und große Anzahlen an Ausgangsstopps. Im Vergleich sind die geschätzten Werte (mit * markiert) des naiven RAPTOR Algorithmus dargestellt.

optimalen Treffpunkt zu finden. Im Maximum liegt die Anzahl der benötigten Treffpunkte bei 18 bzw. 19 und somit bei maximal vier Runden des iterativen Algorithmus für diese Anzahl an Ausgangsstopps. Der RAPTOR SELECT LOOP Algorithmus kann somit zur effizienten Berechnung von optimalen Treffpunkten gemäß dem MinMax Optimalitätskriterium für sehr große Anzahlen an Ausgangsstopps angewandt werden.

#Genaue Berechnungen für #Ausgangsstopps	Durchschnitt	Maximum
500	11,4	18
1000	11,8	19

Tabelle 6.18: Die Anzahl der Ausgangsstopps, für die während des RAPTOR SELECT LOOP Algorithmus die frühesten Ankunftszeiten berechnet und somit bei der Berechnung des optimalen Treffpunkts gemäß dem MinMax Kriterium berücksichtigt werden.

7 Fazit

Um das Treffpunkt-Problem in öffentlichen Verkehrsnetzen zu lösen, haben wir unterschiedliche Algorithmen entwickelt. Da unsere neu entwickelten Algorithmen auf Prioritätenfunktionen basieren, die Heuristiken für die Fahrtzeit zwischen zwei Stopps benötigen, haben wir uns zunächst mit der Berechnung dieser beschäftigt. Dazu haben wir die Methode der minimalen Distanzen in zeitunabhängigen Graphen von Bruns in [Bru24] betrachtet. Zusätzlich haben wir eine Variante entwickelt, bei der die unteren Schranken über vorberechnete Fahrtzeiten im zeitunabhängigen Graphen zu Landmarken bestimmt werden können. Unsere Experimente haben dabei gezeigt, dass die Heuristiken, die aus der Berechnung über Landmarken resultieren, zwar etwas größere Unterschiede zu den eigentlichen minimalen Fahrtzeiten aufweisen, dafür aber zur Laufzeit der Algorithmen, in denen sie eingesetzt werden können, weniger Berechnungszeit benötigen.

Um diese Heuristiken und daraus resultierenden Prioritätenfunktionen in der Routenplanung in öffentlichen Verkehrsnetzen anwenden zu können, haben wir anschließend ausgehend von dem klassischen RAPTOR Algorithmus zwei neue Algorithmen entwickelt, mit denen die frühesten Ankunftszeiten und somit die kürzesten Reisen zwischen zwei Stopps ermittelt werden können. Dabei verwenden der RAPTOR B* und RAPTOR PQ* Algorithmus jeweils die unteren Schranken, um Routen nicht zu betrachten, die nicht zum optimalen Ergebnis führen können. Der RAPTOR PQ* Algorithmus bricht dabei das rundenbasierte Konzept des RAPTOR Algorithmus und expandiert die Routen zielgerichtet über eine Prioritätenwarteschlange. Mit dieser Optimierung erzielen wir eine Beschleunigung der Ausführungszeit des RAPTOR Algorithmus, um eine Lösung für das EAT Problem zu berechnen.

Zur Lösung des Treffpunkt-Problems haben wir die gesuchten Treffpunkte nach zwei Kriterien optimiert und unsere neu entwickelten Algorithmen mit einer naiven Herangehensweise verglichen. Mit den neu entwickelten RAPTOR B und RAPTOR PQ Algorithmen lassen sich die optimalen Treffpunkte effizienter berechnen, da wir auch hier eine zielgerichtetere Berechnung durch unsere Schranken erzielen. Für weniger als 10 Ausgangsstopps hat insbesondere der RAPTOR B Algorithmus für das MinMax Optimalitätskriterium deutlich geringere Ausführungszeiten als die naive Herangehensweise. Für größere Anzahlen an Ausgangsstopps haben wir den RAPTOR SELECT LOOP Algorithmus entwickelt. Dieser wählt eine kleine Anzahl der Ausgangsstopps aus und berechnet lediglich für diese den optimalen Treffpunkt. Für das MinMax Kriterium lässt sich die Optimalität des so gefundenen Treffpunkts für alle Ausgangsstopps effizient verifizieren. Dies haben wir genutzt, um mit einem iterativen Algorithmus die optimalen Treffpunkte für bis zu 1000 Ausgangsstopps effizient zu berechnen. Dabei werden

selbst für diese Anzahl an Ausgangstopps im Maximum lediglich 19 genaue Berechnungen benötigt. Da diese zudem parallelisiert werden können, reichen maximal vier Runden des iterativen Algorithmus aus, um den optimalen Treffpunkt zu bestimmen. Dies liefert eine Beschleunigung der Ausführungszeit des naiven Algorithmus um einen Faktor von über 10 im Durchschnitt.

Der vollständige Programmcode aller beschriebenen Algorithmen und zugehöriger Experimente befindet sich in [GitHub MeetingPoints Repository](#).

Ausblick

Die Ausführungszeit unserer neuen Algorithmen für das EAT- und das Treffpunkt-Problem ist durch die Anzahl der expandierten Routen bestimmt. Um möglichst viele Routen aus der Berechnung auszuschließen, benötigen wir Heuristiken, die möglichst kleine Differenzen zu den minimalen Fahrtzeiten zwischen zwei Stopps aufweisen. Eine weitere Verbesserung dieser unteren Schranken könnte somit in kürzeren Ausführungszeiten resultieren. Da unsere Ansätze lediglich auf zeitunabhängigen Graphen basieren, wäre es interessant zu sehen, inwiefern zeitabhängige Bedingungen wie Umsteige- oder Abfahrtszeiten in die Berechnung von Heuristiken der minimalen Fahrtzeit eingebunden werden können.

Unsere Experimente haben gezeigt, dass selbst für kleine Anzahlen an Ausgangstopps die neu entwickelten RAPTOR B und RAPTOR PQ Algorithmen lediglich für das MinMax Optimalitätskriterium einen Zeitvorteil gegenüber dem naiven Algorithmus bieten. Für größere n kann der iterative RAPTOR SELECT LOOP Algorithmus nur für MinMax angewandt werden, da die Verifikation der Optimalität eines Treffpunkts lediglich für dieses Kriterium effizient erfolgen kann. Daher ist es von Interesse zu erforschen, inwieweit ebenfalls effiziente Algorithmen für das MinSum Kriterium sowohl für kleine als auch große n entwickelt werden könnten.

Literaturverzeichnis

- [ALF+20] D. Atzmon, J. Li, A. Felner, E. Nachmani, S. Shperberg, N. Sturtevant, S. Koenig. „Multi-Directional Heuristic Search“. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. Hrsg. von C. Bessiere. Main track. International Joint Conferences on Artificial Intelligence Organization, Juli 2020, S. 4062–4068. DOI: [10.24963/ijcai.2020/562](https://doi.org/10.24963/ijcai.2020/562). URL: <https://doi.org/10.24963/ijcai.2020/562> (zitiert auf S. 21, 26, 27, 45).
- [AN18] E. Ahmadi, M. Nascimento. „Optimal Meeting Points for Public Transit Users“. In: Juni 2018, S. 15–23. DOI: [10.1109/MDM.2018.00017](https://doi.org/10.1109/MDM.2018.00017) (zitiert auf S. 30).
- [BCE+10] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, F. Viger. „Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns“. In: Bd. 6346. Sep. 2010, S. 290–301. ISBN: 978-3-642-15774-5. DOI: [10.1007/978-3-642-15775-2_25](https://doi.org/10.1007/978-3-642-15775-2_25) (zitiert auf S. 30).
- [BGM10] A. Berger, M. Grimmer, M. Müller-Hannemann. „Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks“. In: Mai 2010, S. 35–46. ISBN: 978-3-642-13192-9. DOI: [10.1007/978-3-642-13193-6_4](https://doi.org/10.1007/978-3-642-13193-6_4) (zitiert auf S. 30).
- [Bru24] D. Bruns. „Optimal Routing in Public Transportation Networks Using PHAST“. In: 2024. URL: <http://dx.doi.org/10.18419/opus-14220> (zitiert auf S. 31, 33, 37, 39, 46, 48, 55, 56, 71).
- [DGNW11] D. Delling, A. V. Goldberg, A. Nowatzyk, R. F. Werneck. „PHAST: Hardware-Accelerated Shortest Path Trees“. In: *2011 IEEE International Parallel Distributed Processing Symposium*. 2011, S. 921–931. DOI: [10.1109/IPDPS.2011.89](https://doi.org/10.1109/IPDPS.2011.89) (zitiert auf S. 34).
- [Dij59] E. W. Dijkstra. „A Note on Two Problems in Connexion with Graphs.“ In: *Numerische Mathematik 1* (1959), S. 269–271 (zitiert auf S. 25, 29, 30, 33, 37).
- [DPSW17] J. Dibbelt, T. Pajor, B. Strasser, D. Wagner. *Connection Scan Algorithm*. 2017. arXiv: [1703.05997](https://arxiv.org/abs/1703.05997) [cs.DS] (zitiert auf S. 29, 30).
- [DPW12] D. Delling, T. Pajor, R. Werneck. „Round-Based Public Transit Routing“. In: *Transportation Science* 49 (Jan. 2012). DOI: [10.1287/trsc.2014.0534](https://doi.org/10.1287/trsc.2014.0534) (zitiert auf S. 3, 17, 20, 30, 31, 41, 57).

- [GH05] A. V. Goldberg, C. Harrelson. „Computing the shortest path: A search meets graph theory“. In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '05. Vancouver, British Columbia: Society for Industrial und Applied Mathematics, 2005, S. 156–165. ISBN: 0898715857 (zitiert auf S. 37, 38, 52, 56).
- [GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. „Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks“. In: *Experimental Algorithms*. Hrsg. von C. C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 319–333. ISBN: 978-3-540-68552-4 (zitiert auf S. 34).
- [HNR68] P. E. Hart, N. J. Nilsson, B. Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), S. 100–107. DOI: [10.1109/TSSC.1968.300136](https://doi.org/10.1109/TSSC.1968.300136) (zitiert auf S. 26, 39).
- [LP21] J. Lee, S. Park. „Efficient methods for finding an optimal network location for travel planning“. In: *The Journal of Supercomputing* 77.11 (Nov. 2021), S. 12561–12580. ISSN: 1573-0484. DOI: [10.1007/s11227-021-03776-7](https://doi.org/10.1007/s11227-021-03776-7). URL: <https://doi.org/10.1007/s11227-021-03776-7> (zitiert auf S. 25).
- [San24] J. Sander. „Berechnung optimaler Wege im öffentlichen Verkehr“. In: 2024. URL: <http://dx.doi.org/10.18419/opus-12540> (zitiert auf S. 17, 21).
- [ZLTZ13] R. Zhong, G. Li, K.-L. Tan, L. Zhou. „G-tree: an efficient index for KNN search on road networks“. In: *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. CIKM '13. San Francisco, California, USA: Association for Computing Machinery, 2013, S. 39–48. ISBN: 9781450322638. DOI: [10.1145/2505515.2505749](https://doi.org/10.1145/2505515.2505749). URL: <https://doi.org/10.1145/2505515.2505749> (zitiert auf S. 25).

Alle URLs wurden zuletzt am 31. 07. 2024 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift