Karlsruher Institut für Technologie

Heuristische Optimierung der globalen Zielfunktion im dynamischen Ridesharing

Bachelorarbeit von

Kevin Frisen

an der Fakultät für Informatik
Institut für Theoretische Informatik, Algorithm Engineering

Erstgutachter:	Prof. Dr. Peter Sanders
Zweitgutachter:	Prof. Dr. Thomas Bläsius
Betreuender Mitarbeiter:	Moritz Laupichler, M.Sc.

01. Mai 2023 – 31. August 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst, und weder ganz oder in Teilen als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet.

PLACE, DATE

. . . . . .

(Kevin Frisen)

Zusammenfassung

Deutsche Zusammenfassung

LOUD ist ein Ridesharing-Algorithmus, welcher Fahrtanfragen zu dem Zeitpunkt, an dem sie eintreffen, in die Route eines Fahrzeuges aus einer begrenzten Menge an Fahrzeugen einfügt. Dafür betrachtet LOUD den Umweg, den ein Fahrzeug machen muss, um die Fahrtanfrage abzuarbeiten. Dem Fahrzeug, welches den kleinsten Umweg für die Fahrtanfrage machen muss, weist LOUD die Fahrtanfrage zu. Dadurch soll eine globale Zielfunktion, die durchschnittliche Gesamtfahrzeit eines Fahrzeuges, minimiert werden. LOUD berücksichtigt dabei nicht, ob ein Fahrzeug von Bedeutung für zukünftige Fahrtanfragen ist, aber durch die Zuweisung der aktuellen Fahrtanfrage an das Fahrzeug für die zukünftige Fahrtanfrage nicht mehr zur Verfügung steht. In dieser Arbeit befassen wir uns mit der strategischen Planung für das Ridesharing-Problem. Dabei bestimmen wir heuristisch die Fahrzeuge, die von Bedeutung für zukünftige Fahrtanfragen sind und benachteiligen diese bei der Auswahl eines Fahrzeuges für eine Fahrtanfrage. Dadurch sollen die Fahrzeuge derart positioniert werden, dass sich der Umweg, welches ein Fahrzeug für eine zukünftige Fahrtanfrage machen muss verringert und damit die globale Zielfunktion weiter minimiert werden kann. Wir konnten zeigen, dass mit diesem Ansatz die durchschnittliche Gesamtfahrzeit, sowie die Reisezeit der Fahrgäste im 95%-Quantil reduziert werden kann.

Abstract

LOUD is a ridesharing algorithm that inserts ride requests into the route of a vehicle from a limited set of vehicles at the time they arrive. To do this, LOUD considers the detour that a vehicle has to make in order to process the ride request. LOUD assigns the ride request to the vehicle with the smallest detour for the ride request. This aims to minimize a global objective function. Therefor we use the average total trip time of a vehicle. LOUD does not take into account whether a vehicle is important for future requests, but is no longer available for future ride requests due to the assignment of the current ride request to the vehicle. In this thesis, we address strategic planning for the ridesharing problem. We heuristically determine the vehicles that are important for future ride requests and penalize them when selecting a vehicle for a ride request. This should position the vehicles in such a way that the detour that a vehicle has to make for a future ride request is reduced and the global objective function can thus be further minimized. We were able to show that this approach can reduce the average total trip time as well as the trip time of the passengers in the 95%-quantile.

Inhaltsverzeichnis

Zusammenfassung **i**

1 Einleitung **1**
1.1 Verwandte Arbeiten 2
1.2 Eigener Betrag 2

2 Grundlagen **3**
2.1 Ridesharing-Modell 3
2.2 Dijkstras Algorithmus 4
2.3 Contraction Hierarchies 5
2.4 LOUD 7
2.5 Graph-Voronoi-Diagramm 9

3 LOUD Voronoi **13**
3.1 Kostenfunktion 13
3.2 Metrik für Flexibilitätsbedarf 14
3.3 Graph-Voronoi-Diagramm für LOUD 17

4 Evaluation **23**
4.1 Aufbau der Experimente 23
4.2 Evaluation der Laufzeit 24
4.3 Evaluation der Qualitätsmetriken 27

5 Schlussfolgerung **31**
5.1 Zukünftige Arbeit 31

Literatur **33**

1 Einleitung

Schon heute ist Ridesharing von Bedeutung in innerstädtischen Raum und wird zukünftig an Bedeutung gewinnen. Es existieren bereits Plattformen für Ridesharing, wie Uber oder BlaBlaCar. Diese unterschieden sich allerdings in der Definition von Ridesharing. Wir verstehen unter Ridesharing, dass nicht nur ein Pool von Fahrzeugen einer Menge von Personen zur Verfügung steht, die eine Fahrt ordern können für sich alleine, wie es bei einem Taxi-Unternehmen der Fall wäre. Zusätzlich dazu teilen sich die Fahrgäste die Fahrten. So kann die Auslastung der Fahrzeuge gesteigert und damit die Anzahl an Fahrzeugen im Verkehr reduziert werden. Das hat wiederum positive Auswirkungen auf die Verkehrslage, die ein hohe Verkehrsbelastung aufweisen, wie z.B. in Ballungsgebieten. Da die Fahrgäste sich die Fahrten teilen, kann die Anzahl an Fahrten reduziert werden. Das bietet den Vorteil, dass die Kosten für eine Fahrt pro Person sich reduzieren, da die Fahrgäste sich die Kosten für die Fahrt teilen. Weiter kann durch die Reduktion der Fahrten die Gesamtdistanz verringert werden, die die Fahrzeuge zurücklegen und damit auch der Ausstoß von Schadstoffen. Die Fahrgäste müssen allerdings hinnehmen, dass sich die Fahrzeiten erhöhen. Das liegt daran, dass sich die Routen zweier Personen in der Regel nicht die gleiche Route zurücklegen wollen, sondern die Routen sich nur Teilweise überlappen. Der entsprechende Umweg, welcher durch weitere Fahrgäste im Fahrzeug verursacht wird, bewirkt eine Verlängerung der Fahrzeit.

Es existieren bereits Dispatching-Algorithmen, wie z.B. LOUD, die eine Fahrtanfrage einer Person einem Fahrzeug zuweisen. LOUD ist Online-Algorithmen, welcher versuchen eine globale Zielfunktion zu minimieren. Dafür verwendet LOUD als globale Zielfunktion die Summe der Fahrzeiten aller Fahrzeuge, die benötigt wird um alle Fahrtanfragen abzuarbeiten. Um das zu erreichen wählt LOUD äquivalent dazu das Fahrzeug aus, welches den kleinsten Umweg machen muss, um die Fahrtanfrage zu bedienen und damit auch den geringsten Anstieg der Fahrzeit verursacht. Diese Herangehensweise liefert allerdings nicht zwangsläufig optimalen Ergebnisse, da es sich hierbei lediglich um eine Heuristik für die globale Zielfunktion handelt. Diese Heuristik ist nicht zwingend die beste Heuristik, da dieser Ansatz den Systemzustand nicht berücksichtigt. So liefern Zuweisungen von Fahrtanfragen zu Fahrzeugen, welche zwar nicht den kleinsten Umweg besitzen, aber immer noch eine gute Zuweisung darstellen, ähnlich gute Ergebnisse. Dieser Spielraum soll ausgenutzt werden, um für die Zukunft zu planen und damit gute Wahlmöglichkeiten zu erzeugen. Es soll bei der Zuweisung also berücksichtigt werden, welchen Einfluss die Wahl des Fahrzeuges für eine Fahrtanfrage auf den Systemzustand hat und damit auch wie gut das System zukünftige Fahrtanfragen beantworten kann.

1.1 Verwandte Arbeiten

Dial-A-Ride-Problem Das Ridesharing-Problem ist verwandt mit dem Dial-A-Ride-Problem (DARP) [2]. Das Problem besteht darin für eine begrenzte Menge an Fahrzeugen die Touren zu planen, sodass alle Fahrtanfragen beantwortet werden können. Die meisten Untersuchungen befassen sich mit statischen Variante des Problems, wobei alle Fahrtanfragen im Voraus bekannt sind. Das statische DARP ist bekanntermaßen NP-vollständig. Wir befassen uns mit dynamischem Ridesharing. Hierbei erhält der Dispatcher die Fahrtanfragen erst zu dem Zeitpunkt an dem die Anfrage gestellt wird und muss diese einem Fahrzeug zuweisen ohne die Fahrtanfragen zu kennen, die in der Zukunft liegen. Für die Zuweisung einer Fahrtanfrage an ein Fahrzeug verwenden wir Heuristiken, um lokal entscheiden zu können, wie die Fahrtanfrage in die Tour eines Fahrzeuges eingefügt werden soll in der Erwartung, dass damit eine globale Zielfunktion minimiert wird.

Rebalancieren von Fahrzeugen Ridesharing-Algorithmen fokussieren sich primär darauf Fahrtanfragen möglichst optimal in die Routen einer Menge von Fahrzeugen einzufügen. Dabei kann es vorkommen, dass Fahrzeuge in einer Region in den Leerlauf übergehen, in der nur wenige Fahrtanfragen erwartet werden. Rebalancierung [8] von Fahrzeugen versucht Fahrzeuge, die sich im Leerlauf befinden, umzuverteilen, sodass diese in einer Region für Fahrtanfragen zur Verfügung stehen, die ein hohes Aufkommen an Fahrtanfragen aufweist. Durch die Umverteilung von Fahrzeugen im Leerlauf soll eine globale Zielfunktion minimiert werden.

1.2 Eigener Betrag

Bei dieser Arbeit handelt es sich um eine Erweiterung des LOUD Algorithmus. Verändert wird hierbei die Berechnung der Kosten für eine Zuweisung einer Fahrtanfrage zu einem Fahrzeug. Es soll die Bedeutung eines Fahrzeuges für zukünftige Fahrtanfragen geschätzt werden und in der Kostenberechnung berücksichtigt werden, sodass das System in einen besseren Systemzustand kommt, als durch die Wahl des Fahrzeuges mit dem geringsten Umweg und dadurch die globale Zielfunktion weiter minimiert werden.

Die Arbeit führt zunächst im Kapitel 2 in die für diese Arbeit benötigten Grundlagen ein. Dazu gehören Begriffe, sowie Algorithmen, die verwendet werden. Anschließend wird im Kapitel 3 detailliert beschrieben, wie die Bedeutung eines Fahrzeuges für zukünftige Fahrtanfragen berechnet wird und wie diese in die Kosten für eine Zuweisung einfließen. Zuletzt wird die Arbeit im Kapitel 4 evaluiert.

2 Grundlagen

2.1 Ridesharing-Modell

Zunächst führen wir grundlegende Begriffe ein, die im weiteren Verlauf dieser Arbeit verwendet werden.

Straßennetzwerk Ein *Straßennetzwerk* wird als gerichteter, gewichteter Graphen $G = (V, E)$ dargestellt, wobei V die Menge aller Kreuzungen und E die Menge aller Straßenabschnitten beschreibt. Alle $e = (u, v) \in E$ haben nicht-negative Kantengewichte. Diese entsprechen der Reisezeit, die benötigt wird, um diesen Straßenabschnitt zu passieren.

Fahrzeug Ein *Fahrzeug* ist ein 4-Tupel $v = (l_i, c, t^{min}_{serv}, t^{max}_{serv})$ bestehend aus einer Startposition l_i, der Sitzkapazität des Fahrzeuges c mit $c \geq 1$ und dessen Servicezeit $[t^{min}_{serv}, t^{max}_{serv})$, in der das Fahrzeug aktiv ist.

Stopp Ein *Stopp* $v \in V$ ist ein Knoten im Straßennetzwerk, an dem ein Fahrgast abgesetzt und/oder abgeholt wird. Um einen Stopp durchzuführen, hält das Fahrzeug für eine Zeitdauer von t_{stop}. Es wird hierbei angenommen, dass Stopps nur an Kreuzungen stattfinden. In der Realität finden diese auch entlang eines Straßenabschnittes statt. Zur Vereinfachung wird davon abstrahiert.

Route Eine *Route* $R(v) = \langle s_0, ..., s_k \rangle$ ist eine Folge von Stopps eines Fahrzeuges v an den Positionen $l(s_j) \in V$ für $j \in 0...k$. Dabei beschreibt $l(s_0) \in V$ den letzten abgeschlossenen Stopp des Fahrzeuges. Hält ein Fahrzeug zum aktuellen Zeitpunkt, dann entspricht $l(s_0) \in V$ der aktuellen Position. Hat ein Fahrzeug seine Route abgearbeitet und seinen letzten Stopp erreicht, so befindet es sich bei $s_k = s_0$ im Leerlauf und bedient keine Fahrgäste. Wir bezeichnen derartige Fahrzeuge fortan als idle.

Request Ein *Request* $r = (p, d, t^{min}_{dep})$ ist eine Mitfahranfrage eines Fahrgastes, bestehend aus $p \in V$ der Pickup-Position, an dem der Fahrgast abgeholt werden soll, $d \in V$ der Dropoff-Position, an dem der Fahrgast abgesetzt werden soll und t^{min}_{dep} dem frühesten Abholzeitpunkt des Fahrgastes. In diesem Modell entspricht t^{min}_{dep} dem Zeitpunkt, an dem der Request gestellt wird. Es ist also kein Vorbuchen möglich.

Insertion Eine *Insertion* $\iota = (v, r, i, j)$ ist ein 4-Tupel, welches beschreibt, dass der Request r dem Fahrzeug v zugewiesen wird und von diesem abgearbeitet wird. Der Stopp des Fahrzeuges an der Pickup-Position p wird an i-ter Stelle in der Route $R(v) = \langle s_0, ..., s_k \rangle$, $i < j <= k$ eingefügt und an j-ter Stelle die Dropoff-Position d, also $s_i = p$ und $s_j = d$. Es existieren dabei verschiedenen Insertion-Typen. Für diese Arbeit sind Pickup-After-Last-Stop (PALS) Insertions von Bedeutung. Sei $r = (p, d, t^{min}_{dep})$ ein Request und $R(v) = \langle s_0, ..., s_k \rangle$ eine Route. Eine PALS-Insertion ist eine Insertion $\iota = (v, r, i, j)$, sodass $i = j = k$. Pickup-Position p und Dropoff-Position d werden also hinten an die bestehende Route R eingefügt.

Detour Eine *Detour* δ_ι bezeichnet den Umweg, welcher durch eine Insertion $\iota = (v, r, i, j)$ für ein Fahrzeug verursacht wird. Eine Detour entspricht also der Erhöhung der Zeit, die das Fahrzeug v benötigt um am letzten Stopp der Route anzukommen. Eine Detour δ_ι setzt sich dabei aus der Strecke zur Pickup-Position $dist(s_{i-1}, p)$, der Strecke von der Pickup-Position zurück $dist(p, s_{i+1})$, der Strecke zur Dropoff-Position $dist(s_{j-1}, d)$ und der Strecke von der Dropoff-Position zurück $dist(d, s_{j+1})$ zusammen. Es ergibt sich die Formel:

$$\delta_\iota = dist(s_{i-1}, p) + dist(p, s_{i+1}) + dist(s_{j-1}, d) + dist(d, s_{j+1}) - dist(s_i, s_{i+1}) - dist(s_j, s_{i+1})$$

In diesem Modell seien das Straßennetzwerk, eine Menge an Fahrzeugen, sowie die Routen der Fahrzeuge bekannt. Beachte, dass die Routen der Fahrzeuge initial nur aus der aktuellen Position bestehen. Requests können zu einem beliebigen Zeitpunkt auftreten. Es soll nun eine Insertion gefunden werden, sodass die Detour δ minimiert wird unter Berücksichtigung von Einschränkungen. Dazu gehört zum einen die Sitzplatzkapazität c eines Fahrzeuges. Weiter darf die maximale Wartezeit und die maximale Reisezeit der Requests r', die einem Fahrzeug bereits zugewiesen wurden, nicht überschritten werden.

Die maximale Wartezeit t_{wait}^{max} ist ein Schwellwert, durch welchen festgelegt wird, wann ein Fahrgast spätestens abgeholt werden muss. Der späteste Abholzeitpunkt $t_{dep}^{max}(r') = t_{dep}^{min}(r') + t_{wait}^{max}$ eines Requests r' ist definiert als die Zeitspanne zwischen dem Stellen des Requests und dem Ankommen des zugewiesenen Fahrzeuges an der Pickup-Position $p(r')$. Hierbei ist t_{wait}^{max} ein Systemparameter. Die maximale Reisezeit $t_{trip}^{max}(r') = \alpha * dist(p(r'), d(r')) + \beta$ eines Requests r' entspricht der Zeit, die nach dem Stellen eines Requests maximal vergehen darf bis der Fahrgast an seiner Dropoff-Position $d(r')$ ankommt. Es muss also gelten, dass die Detour, die eine Insertion verursacht keine Überschreitung der maximalen Ankunkftszeit $t_{arr}^{max}(r') = t_{dep}^{min}(r') + t_{trip}^{max}(r')$ der Requests r' bewirkt, die dem Fahrzeug bereits zugewiesen sind. α und β sind Systemparameter.

Diese Einschränkung sind harte Einschränkungen für die Requests, die Bereits einem Fahrzeug zugewiesen wurden. Für den aktuellen Request sind die maximale Wartezeit und Reisezeit weiche Einschränkungen. Diese dürfen verletzt werden. Die Verletzung der Einschränkungen wird in der Kostenfunktion berücksichtigt mit

$$f(\iota) = \delta_\iota + \gamma_{wait} * max\{t_{dep}(p(r)) - t_{dep}^{max}(r), 0\} + \gamma_{trip} * max\{t_{arr}(d(r)) - t_{arr}^{max}(r), 0\},$$

wobei γ_{wait} und γ_{trip} Systemparameter sind, $t_{dep}(p(r))$ die berechnete Abfahrtzeit von der Pickup-Position und $t_{arr}(d(r))$ die berechnete Ankunftzeit an der Dropoff-Position.

Ein Request wird abgelehnt, wenn keine gültige Insertion gefunden werden kann, was aufgrund der weichen Einschränkungen nur dann vorkommen kann, wenn kein Fahrzeug existiert, dass sich aktuell in seiner Servicezeit befindet. Wenn man von autonomen Fahrzeugen ausgeht, ist die Servicezeit unbeschränkt. In diesem Fall wird also kein Request abgelehnt.

2.2 Dijkstras Algorithmus

Beim Dijkstras Algorithmus [3] handelt es sich um einen one-to-all Kürzeste-Wege-Algorithmus auf Graphen. Sei $G = (V, E)$ ein Graph mit nicht-negativen Distanzen $l(e) \geq 0$

für alle $e \in E$ und $s \in V$ der Startknoten der Suche. Dijkstras Algorithmus findet kürzeste Wege durch das Aufbauen eines Kürzeste-Wege-Baumes mit Wurzel s.

Dafür hält Dijkstras Algorithmus ein Distanz-Array d, ein Parent-Array p und eine Priority-Queue (PQ). Das Distanz-Array d enthält alle vorläufig kürzesten Distanz $d[v]$ von s nach v für alle $v \in V$. Das Parent-Array p enthält für einen Knoten $v \in V$ den Elternknoten im Kürzeste-Wege-Baum. Initial enthält die PQ nur den Startknoten s mit der Distanz $d[s] = 0$ und $p[s] = s$. Für die Distanzen gilt, dass $d[v] = \infty$. Dijkstras Algorithmus entfernt nun solange den Knoten mit der kleinsten Distanz aus der PQ bis die PQ leer ist. Sei $u \in V$ der Knoten mit kleinster Distanz in der PQ, der bei einer Iteration entfernt wurde, dann werden alle Nachbarknoten von u traversiert und es wird überprüft, ob ein neuer kürzester Weg zu dem Nachbarn gefunden wurde. In diesem Fall werden d, p und die PQ aktualisiert. Wir sagen, dass der Knoten u gesettlt wurde. Wenn ein Knoten aus der PQ entfernt wird, wurde der kürzeste Weg für diesen gefunden.

Algorithmus 1 : Dijkstras Algorithmus

Input : $G = (V, E)$ mit nicht-negativen Kantengewichten $l(e) \geq 0$ für $\forall e \in E$
$s \in V$ Startknoten
Data : Q Priority-Queue
Result : Distanzen d[v] und Elternknoten p[v] für alle $v \in V$

```
// Initialisiere Distanzen d und Elternknoten p
```
1 $d[v] \leftarrow \infty \; \forall v \in V$;
2 $p[v] \leftarrow \perp \; \forall v \in V$;
3 $d[s] \leftarrow 0$;
4 $Q.insert(s, 0)$;

```
// Baue den Kürzeste-Wege-Baum auf
```
5 **while** *Q is not empty* **do**
6 $u \leftarrow Q.deleteMin()$;
7 **for** $(u, w) \in E$ **do**
8 **if** $d[w] > d[u] + l(u, w)$ **then**
9 $d[w] \leftarrow d[u] + l(u, w)$;
10 $p[w] \leftarrow u$;
11 **if** $Q.contains(w)$ **then**
12 $Q.decreaseKey(w, d[w])$
13 **else**
14 $Q.insert(w, d[w])$;

2.3 Contraction Hierarchies

Contraction Hierachies (CH) [4] ist ein Beschleunigungstechnik für die Kürzeste-Wege-Suche in Straßennetzwerken. Die Technik besteht dabei aus zwei Phasen. In der ersten Phase, der Preprocessing Phase, werden zwei Suchgraphen aufgebaut. Die Suchgraphen

werden in der Query Phase verwendet, um kürzeste Wege effizient zu berechnen. Die Technik nutzt dabei die Beobachtung aus, dass Straßennetze hierarchisch aufgebaut sind. Straßennetze bestehen also aus einigen wichtigen Straßen und Kreuzungen, die von größerer Bedeutung für den Verkehr sind, und aus vielen unwichtigen, die für den Lokalverkehr benötigt werden. Die Suche wird dabei in die Richtung wichtiger Kanten beschränkt und damit die Kürzeste-Wege-Suche beschleunigt.

Preprocessing Phase

Sei $G = (V, E)$ ein Straßennetzwerk mit nicht-negativen Kantengewichten $l(e) \geq 0$ mit $e \in E$ gegeben. In der Preprocessing Phase werden alle Knoten $v \in V$ gemäß einer Heuristik nach ihrer Wichtigkeit sortiert und anschließend nach aufsteigender Wichtigkeit kontrahiert. Bei der Kontraktion wird ein Knoten $v \in V$ temporär aus dem Graphen G gelöscht und für jeden Pfad $p = (u, v, w)$ überprüft, ob v auf dem kürzesten Weg von u nach w liegt. Ist das der Fall wird ein Shortcut $e' = (u, w)$ mit der Distanz $l(e') = l(u, v) + l(v, w)$ hinzugefügt. Ein Shortcut wird allerdings nur dann hinzugefügt, wenn es sich bei p um einen eindeutigen kürzesten Pfad handelt. Um das zu überprüfen wird eine lokale Dijkstra-Suche als Witness Search eingesetzt. Es handelt sich dabei um die Suche nach einem kürzesten Pfad von u nach v. Falls diese keinen anderer Witness findet, wird der Shortcut eingefügt.

Query Phase

In dieser Phase werden kürzeste Wege von $s \in V$ nach $t \in V$ berechnet. Hierfür wird eine Form des bidirektionalen Dijkstras Algorithmus verwendet, wobei eine Vorwärtsuche ausgehend von s und eine Rückwärtssuche ausgehend von t gestartet wird. Wir bezeichnen die Vorwärtssuche als Vorwärts-CH-Suche und die Rückwärtssuche als Rückwärts-CH-Suche. Es werden dabei nur die Kanten relaxiert, die zu einem Knoten höheren Rangs führen, also eine höhere Wichtigkeit haben. Die Suche wird beendet sobald eine der beiden Suchen eine größere Distanz erreicht hat als die kürzeste bisher gefundene s-t-Distanz. In diesem Fall ist diese s-t-Distanz die Kürzeste-Wege-Distanz gefunden.

Bucket Contraction Hierachies

Bei Bucket Contraction Hierachies (BCH) [6] handelt es sich um eine Erweiterung von CHs, welche verwendet wird um das one-to-many Kürzeste-Wege-Problem zu lösen. Sei also $s \in V$ ein Startknoten und $T \subseteq V$ eine Menge von Zielen, dann berechnet BCH die kürzesten Wege von s zu allen $t \in T$.

Für die Kürzeste-Wege-Berechnung hält der Algorithmus für alle $v \in V$ einen initial leeren Bucket $B(v)$ und vorläufige Distanzen $D_s(t)$ von s zu allen $t \in T$, die mit ∞ initialisiert werden. Zuerst wird eine Rückwärts-CH-Suche für alle $t \in T$ durchgeführt. Dabei wird für jeden Knoten $v \in V$ der Eintrag $(t, dist(v, t))$ dem Bucket $B(v)$ hinzugefügt, wenn der Knoten v gesettlet wird. Ein Bucketeintrag kann hierbei als Shortcut von v nach t angesehen werden. Danach wird eine Vorwärts-CH-Suche von s ausgehend gestartet. Wird ein Knoten $u \in V$ gesettlet, wird über die Einträge $(t, dist(u, t)) \in B(u)$ des Buckets

iteriert. Im Fall von $dist(s,u)+dist(u,t) < D_s(t)$, wird $D_s(t) = dist(s,u)+dist(u,t)$ gesetzt. Endet der Algorithmus enthält D_s die Kürzeste-Wege-Distanzen zu allen $t \in T$. BCHs sind sehr effektiv, wenn das selbe T für mehrere Queries genutzt werden soll oder one-to-many Kürzeste-Wege berechnet werden sollen. Im ersten Fall muss die Rückwärts-CH-Suche nur einmal berechnet werden, da die Bucketeinträge nicht von s abhängen, sondern nur von T. Im zweiten Fall muss der Suchraum von s nur einmal traversiert werden, anstatt diesen für jede Query neu zu traversieren, wie das bei CH der Fall wäre.

Analog kann das many-to-one Kürzeste-Wege-Problem von jedem $s \in S \subseteq V$ zu einem Zielknoten $u \in V$ berechnet werden. $B(u)$ speichert in diesem Fall alle Shortcuts von mehreren s zu u.

2.4 LOUD

LOUD (LOcal Bucket Dispatching) [1] ist ein dynamischer Ridesharing-Algorithmus, welcher eine Menge von Requests auf eine Menge von Fahrzeugen abbildet. Der Algorithmus basiert auf dem im Kapitel 2.1 vorgestellten Ridesharing-Modells. Dabei liefert LOUD nachweislich exakte Lösungen.

Die grundlegende Aufgabe, die LOUD löst, ist die Berechnung von gültigen Detours δ_ι. Hierfür verwendet LOUD BCHs. Es werden dafür für jeden Knoten $v \in V$ des Straßennetzwerkes ein Source-Bucket $B_s(v)$ und ein Target-Bucket $B_t(v)$ gehalten. Der Source-Bucket wird verwendet, um das one-to-many Problem zu lösen ausgehend von einem Stopp s. Der Target-Bucket wird verwendet, um das many-to-one Problem von einem beliebigen Knoten $v \in V$ zu einem Stopp s zu lösen. Initial sind alle Buckets leer. Wird ein Stopp s einer Route hinzugefügt, wird eine Vorwärts- und Rückwärts-CH-Suche von s gestartet. Wenn ein Knoten $u \in V$ während der Suche gesettlet wird, wird der Eintrag $(s, dist(s,u))$ dem Source-Bucket $B_s(u)$ bzw. $(s, dist(u,s))$ dem Target-Bucket $B_t(u)$ hinzugefügt. Die Bucket-Einträge für jeden Stopp erlauben also, mithilfe von BCH-Suchen die kürzesten Wege von einem beliebigen Knoten im Straßennetzwerk zu allen Stopps (und andersherum) zu finden.

Um die Detour δ_ι für alle möglichen Insertions ι eines Requests $r = (p, d, t_{dep}^{min})$ zu berechnen, werden zwei Vorwärts-CH-Suchen ausgehend von p und d gestartet, die die Target-Buckets B_t scannen und zwei Rückwärts-CH-Suchen ausgehend von p und d, die die Source-Buckets B_s scannen. Die Bucket-Einträge für jeden Stopp erlauben also, mithilfe von BCH-Suchen die kürzesten Wege von einem beliebigen Knoten im Straßennetzwerk zu allen Stopps (und andersherum) zu finden. Wir bezeichnen die Menge aller Fahrzeuge, die während der Suche gefunden wurden und damit mögliche Insertions besitzen als C.

Sei $R(v) = \langle s_0, ..., s_k \rangle$ die Route des Fahrzeuges $v \in C$ und $o(s)$ die Anzahl an belegten Sitzplätzen, nachdem v den Stopp s verlassen hat. Um die beste Insertion zu finden iteriert LOUD über alle möglichen Pickup-Positionen s_i mit $0 < i < k$. Wenn $o(s_i) < c(v)$, also die Sitzplatzkapazität des Fahrzeuges nicht überschritten wird, iteriert LOUD für ein festes s_i über alle möglichen Dropoff-Positionen s_j mit $i \leq j < k$. Sei $\iota = (v, r, i, j)$ eine solche gefundene Insertion, dann überprüft LOUD, ob ι alle Einschränkungen erfüllt. In diesen Fall merkt sich der Algorithmus die Insertion als beste Insertion, wenn sie die kleinste bisher gesehene Detour besitzt. Das wiederholt LOUD für alle Fahrzeuge $v \in C$.

Einschränkungen überprüfen

Das Überprüfen der Einschränkungen für eine Insertion ist ein zeitkritischer Vorgang, da diese Operation sehr häufig aufgerufen wird. LOUD kann die Einschränkungen in konstanter Zeit überprüfen unabhängig von der Anzahl an Stopps und Requests, die einem Fahrzeug zugewiesen sind.

Wir betrachten die Einschränkungen maximale Wartezeit t_{wait}^{max} und maximale Reisezeit t_{trip}^{max}. Die maximale Wartezeit beschreibt, wie lange ein Fahrgast maximal warten darf bis er von einem Fahrzeug abgeholt wird. Die maximale Reisezeit beschreibt den Zeitpunkt, an dem ein Fahrgast bei seinem Ziel spätestens ankommen muss. Für einen Stopp s merkt sich der Algorithmus die früheste Abfahrtzeit $t_{dep}^{min}(s)$ und die späteste Ankunftszeit $t_{arr}^{max}(s)$. Die späteste Ankunftszeit $t_{arr}^{max}(s)$ definiert den spätesten Zeitpunkt, an dem das Fahrzeug den Stopp s verlassen muss, um die Einschränkungen eines Fahrgastes nicht zu verletzen. Wenn v später als $t_{arr}^{max}(s)$ bei s ankommt, kommt es zu Verspätungen bei folgenden Stopps. Sei $R(v) = \langle s_0, ..., s_k \rangle$ die Route des Fahrzeuges v und $\iota = (r, v, i, j)$. Wir definieren die Pickup-Detour δ_p und die Dropoff-Detour δ_d:

$$\delta_p = dist(s_i, p) + t_{stop} + dist(p, s_{i+1}) - dist(s_i, s_{i+1})$$
$$\delta_d = dist(s_j, d) + t_{stop} + dist(d, s_{j+1}) - dist(s_j, s_{j+1})$$

Man beachte, dass die Distanzen $dist(s_i, s_{i+1})$ und $dist(s_j, s_{j+1})$ nicht explizit gespeichert werden müssen. Sie ergeben sich aus der t_{dep}^{min}, nämlich $dist(s_i, s_{i+1}) = t_{dep}^{min}(s_{i+1}) - t_{dep}^{min}(s_i) - t_{stop}$. Analog lässt sich $dist(s_j, s_{j+1})$ berechnen. Die Einschränkungen sind genau dann erfüllt, wenn durch das Abholen des Fahrgastes keine Verspätung bei den bestehenden Fahrgästen entsteht, d.h.

$$t_{dep}^{min}(s_{i+1}) - t_{stop} + \delta_p \leq t_{arr}^{max}(s_{i+1}),$$

durch das Absetzen des Fahrgastes keine Verspätung bei bestehenden Fahrgästen entsteht, d.h.

$$t_{dep}^{min}(s_{j+1}) - t_{stop} + \delta_p + \delta_d \leq t_{arr}^{max}(s_{j+1})$$

und durch die Insertion die Servicezeit des Fahrzeuges nicht überschreiten wird, d.h.

$$t_{dep}^{min}(s_k) + \delta_p + \delta_d \leq t_{serv}^{max}(v).$$

Man beachte, dass jeder Term in konstanter Zeit bestimmt werden kann und damit die Auswertung konstante Zeit benötigt.

Elliptic Pruning

Für einen Request r besitzen die meisten Fahrzeuge keine gültige Insertion, weil die minimale Detour des Fahrzeuges zu groß ist, sodass die Einschränkungen nicht erfüllt werden können. Eine der grundlegenden Ideen von LOUD beruht auf der Beobachtung, dass der Spielraum λ für die Länge einer Detour zwischen zwei aufeinander folgenden Stopps s_i und s_{i+1} einer Route R durch die harten Einschränkungen der bestehenden Fahrgäste beschränkt ist. Daraus folgt, dass jeder weitere Stopp, welcher in die Route zwischen s_i

und s_{i+1} eingefügt werden soll, sich innerhalb einer Kürzesten-Wege-Ellipse befinden muss. Die Ellipse ist definiert als die Menge aller Knoten $v \in V$, für welche die Ungleichung $dist(s_i, v) + dist(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})$ gilt. Wobei $\lambda(s_i, s_{i+1}) = t_{arr}^{max}(s_{i+1}) - t_{dep}^{min}(s_i) - t_{stop}$ die maximale Zeitspanne beschreibt, die das Fahrzeug zwischen der Abfahrt bei Stopp s_i und der Ankunft bei Stopp s_{i+1} benötigen darf. Die Spielräume $\lambda(s_i, s_{i+1})$ werden verwendet, um die Anzahl der benötigten Bucket-Einträge für jeden Stopp zu verringern. Der Eintrag $(s_i, dist(s_i, v))$ für den Bucket $B_s(v)$, $v \in V$ wird nur dann generiert, wenn sich der Knoten innerhalb der Ellipse von s_i und s_{i+1} befindet. Analog gilt das für den Target-Bucket $B_t(v)$.

Wird während einer BCH-Suche für einen Request r ein Stopp gefunden, merkt sich LOUD das zugehörige Fahrzeug zu dem Stopp in einer Menge C. Fahrzeuge, die nicht in der Menge C liegen, besitzen keine gültige Insertion und werden von der Suche ausgeschlossen. LOUD muss anschließend nur alle möglichen Insertions von C überprüfen.

Elliptic Pruning beschleunigt die BCH-Suche und die Suche nach der besten Insertion, da die Buckets kleiner werden und die Anzahl der zu überprüfenden Fahrzeuge reduziert wird.

LOUD ist 30 mal schneller als state-of-the-art Algorithmen, die derzeit Verwendung finden in Industrie und Wissenschaft.

2.5 Graph-Voronoi-Diagramm

Bei Graph-Voronoi-Diagrammen [7] handelt es sich um eine Adaption von Voronoi-Diagrammen [5] auf Graphen. Wir führen deshalb zunächst Voronoi-Diagramme im euklidischen Raum ein.

Voronoi Diagramm

Sei $S \subseteq \mathbb{R}^2$ eine endliche Menge im euklidischen Raum und $d(v, w)$ mit $v, w \in \mathbb{R}^2$ eine Distanzfunktion. Für einen Punkt $s \in S$ ist die Voronoi-Region

$$VR(s) := \{x \in \mathbb{R}^2 : d(x, s) \leq d(x, s') \; \forall s' \in S\}$$

definiert als die Menge aller Punkte in $\mathbb{R}^2$, für die die Distanz zu s minimal ist für alle $s' \in S$. Man beachte, dass Voronoi-Regionen nicht eindeutig sind. Existieren für einen Punkt $x \in \mathbb{R}^2$ mehrere $s' \in S$ mit minimaler Distanz, wird x genau einer der möglichen Voronoi-Region beliebig hinzugefügt.

Die Menge aller Voronoi-Regionen

$$VD(S) = \{VR(s) : s \in S\}$$

ist dann das Voronoi-Diagramm für die Punktmenge S. VD(S) ist eine disjunkte Zerlegung von $\mathbb{R}^2$.

Graph-Voronoi-Diagramm

Wir übertragen nun Voronoi-Diagramme vom euklidischen Raum $\mathbb{R}^2$ auf Graphen $G = (V, E)$ mit Kantengewicht $l(e) \geq 0$ für $e \in E$.

Sei $S \subseteq V$ eine Knotenmenge. Die Voronoi-Region VR(s) für $s \in S$ definieren wir als

$$VR(s) = \{v \in V : dist(s, v) \leq dist(s', v)\ \forall s' \in S\}$$

die Menge aller Knoten v, dessen kürzester Weg von s nach v minimal ist für alle $s' \in S$. Analog zu den Voronoi-Diagrammen ist die Zuordnung eines Knoten $v \in V$ zu einer Voronoi-Region nicht eindeutig. Im mehrdeutigen Fall wird der Knoten v einer möglichen Voronoi-Region beliebig zugewiesen. Die Menge der Voronoi-Regionen VD(S) zerlegt den Graphen in disjunkte Voronoi-Regionen. VD(S) ist analog zur Definition für Voronoi-Diagramme.

In dieser Arbeit werden wir Voronoi-Regionen $VR(s)$ auch als Voronoi-Zellen mit Zentrum s bezeichnen.

Rand einer Voronoi-Zelle Sei $VR(s)$ die Voronoi-Zelle mit Zentrum s. Der Rand R_s ist definiert als die Menge aller Knoten $w \in V$, die nicht zur Voronoi-Zelle $VR(s)$ gehören und eine Kante $(w, v) \in E$ besitzen mit $v \in VR(s)$, die in $VR(s)$ hineinragt.

$$R_s = \{w \in V\ :\ \exists (w, v) \in E\ mit\ w \notin VR(s),\ v \in VR(s)\}$$

Der Rand einer Voronoi-Zelle kann mit einer Breitensuche berechnet werden. Die Breitensuche starten vom Zentrum der Zelle auf dem Graphen $G = (V, E)$. Sei $w \in V$ ein Knoten, welcher duch die Suche gefunden wurde, dann werden alle eingehenden Kanten $(u, w) \in E$ von w betrachtet. Wird ein Knoten $u \notin VR(s)$ gefunden, der nicht zur Voronoi-Zelle gehört, wird der Knoten u dem Rand R_s hinzugefügt. Für die Überprüfung wird der Rückwärtsgraph $\overleftarrow{G} = (V, \overleftarrow{E})$ verwendet. Wobei $\overleftarrow{E} = \{(w, v) : (v, w) \in E\}$ alle Kanten von E enthält in entgegengesetzter Richtung.

Werden während der Suche Knoten gefunden, die nicht zur Voronoi-Zelle $VR(s)$ gehören oder bereits besucht wurden, dann wird von diesem Knoten ausgehend nicht weiter gesucht (siehe Algorithmus **??**).

Algorithmus 2 : calcBorder

Input : Graph $G = (V, E)$ und dessen Rückwartsgraphen $\overleftarrow{G} = (V, \overleftarrow{E})$
Voronoi-Zelle $VR(s)$
Data : Q : Queue, explored : Bool-Array
Result : Rand der Zelle R_s

```
1  explored[v] ← false ∀v ∈ V;
2  Q.push(s);
3  while Q is not empty do
4      v ∈ V ← Q.pop();
5      for (v, u) ∈ ←E do
6          if u ∉ VR(s) und explored[u] = false then
7              explored[u] ← true;
8              R_s.push(u)
9      for (v, w) ∈ E do
10         if w ∈ VR(s) und explored[w] = false then
11             explored[w] ← true;
12             Q.push(w)
```

3 LOUD Voronoi

In diesem Kapitel wird der Beitrag dieser Arbeit vorgestellt. Bei LOUD Voronoi handelt es sich um eine Erweiterung des LOUD Algorithmus, wobei durch das Verändern der Kostenfunktion eine Verbesserung der Gesamtqualität von LOUD erreicht werden soll. Der Algorithmus versucht dafür abzuschätzen, wie wichtig jedes Fahrzeug für zukünftige Requests sein wird. Das nennen wir den Bedarf an Flexibilität für ein Fahrzeug. Abhängig von dieser Abschätzung versieht der Algorithmus bei der Verarbeitung eines neuen Requests die Kosten von Insertions der Fahrzeuge mit hohem Bedarf an Flexibilität mit einem Malus. So versucht LOUD Voronoi zu vermeiden, Fahrzeuge einzusetzen, welche in naher Zukunft für andere Requests die beste Option darstellen und damit die Disponibilität des Fahrzeuges zu gewährleisten. Neben der Disponibilität eines Fahrzeuges wollen wir weitere Metriken bei der Kostenberechnung für eine Insertion berücksichtigen. So soll die Reisezeit jedes Fahrgastes möglichst kurz sein. Beachte, dass LOUD die Detour einer Insertion als Kostenfunktion verwendet.

3.1 Kostenfunktion

In diesem Abschnitt beschäftigen wir uns damit, wie wir die Disponibilität für ein Fahrzeug in der Kostenfunktion berücksichtigen können. Wir erinnern uns, dass sich die Kostenfunktion $f_{LOUD}(\iota)$ für LOUD aus der Detour δ_ι und Zusatzkosten für das Überschreiten der maximalen Wartezeit und der maximalen Reisezeit für eine Insertion ι zusammensetzt.

Wir wollen die Kostenfunktion nun erweitern, sodass die Disponibilität eines Fahrzeuges für zukünftige Requests gewährleistet wird und die Reisezeit der Fahrgäste minimiert wird. Dafür führen wir zunächst ein paar Begriffe ein.

Reisezeit Die *Reisezeit* $t_{trip}(\iota)$ entspricht der Zeitspanne, die zwischen dem Eingehen eines Requests r und dem Ankommen des Fahrgastes an seinem Reiseziel vergeht für die Insertion $\iota = (v, r, i, j)$.

$$t_{trip}(\iota) = t_{arr}(d(r)) - t_{dep}^{min}(r)$$

Reisezeitveränderung Die *Reisezeitveränderung* $t_{\Delta trip}(\iota)$ beschreibt die Summe der Veränderung der Reisezeit für bestehende Fahrgäste eines Fahrzeuges v, die durch die Insertion $\iota = (v, r, i, j)$ verursacht wird. Die Reisezeit eines Fahrgastes verändert sich, wenn dessen Dropoff-Position sich hinter i oder j befindet. Sei n_{pickup} die Anzahl an Dropoff-Positionen in der Route des Fahrzeuges v, die sich hinter der Pickup-Position des Requests r befinden. Und sei $n_{dropoff}$ die Anzahl an Dropoff-Positionen in der Route des Fahrzeuges v, die sich hinter der Dropoff-Position des Requests r befinden. Dann entspricht die Reisezeitveränderung

$$t_{\Delta trip}(\iota) = n_{pickup} * \delta_p + n_{dropoff} * \delta_d,$$

wobei δ_p die Pickup-Detour und δ_d die Dropoff-Detour definiert.

Disponibilität Die Disponibilitätsfunktion gibt die Bedeutung des Fahrzeuges v für zukünftige Requests an. Die Disponibilität $disp(\iota)$ entspricht dabei dem Malus für die Kosten einer Insertion $\iota = (v, r, i, j)$. Die Insertion soll dabei benachteiligt werden, wenn das Fahrzeug wichtig ist für zukünftige Requests und damit die Disponibilität des Fahrzeuges für diese Requests sichergestellt werden soll. Die Disponibilität setzt sich dabei zusammen aus den zwei Modell-Parametern γ und m, sowie aus einer Disponibilitätsfunktion $dispCost(v)$

$$disp(\iota) = \gamma * m * dispCost(v).$$

Die in LOUD Voronoi verwendete Kostenfunktion ergibt sich aus der Summe der Kostenfunktion von LOUD $f_{LOUD}(\iota)$, der Reisezeit, der Reisezeitveränderung und der Disponibilität. Die Kostenfunktion lässt sich also berechnen mit

$$f_{voronoi}(\iota) = f_{LOUD}(\iota) + t_{trip}(\iota) + t_{\Delta trip}(\iota) + disp(\iota).$$

3.2 Metrik für Flexibilitätsbedarf

Nun wollen wir uns damit beschäftigen, wie die Disponibilitätsfunktion $disp(\iota)$ mithilfe eines Voronoi-Diagramms berechnet werden kann. Zunächst definieren wir den Begriff der Flexibilität eines Fahrzeuges. Dafür betrachten wir die Route $R(v) = \langle s_0, ..., s_k \rangle$ eines Fahrzeuges v. Wir sagen, dass ein Fahrzeug v flexibel ist an Position s_l, $0 \leq l \leq k$ für zukünftige Insertions $\iota = (v, r, i, j)$ mit $i > l$, wenn v nicht durch einen nachfolgenden Stopp s_{l+1} gebunden ist. Daraus folgt, dass ein Fahrzeug nur an seinem letzten Stopp flexibel sein kann. Wenn im folgenden über die Flexibilität eines Fahrzeuges gesprochen wird, nehmen wir Bezug auf die Flexibilität des Fahrzeuges beim letzten Stopp seiner Route. Weiter sprechen wir von einem Bedarf an Flexibilität für ein Fahrzeug v, wenn in naher Zukunft Requests erwartet werden, für die gute Insertions in die Route des Fahrzeuges v existieren. Beachte, dass ein Fahrzeug nur an einem letzten Stopp flexibel ist und daher nur PALS-Insertions betrachtet werden. Der Bedarf an Flexibilität kann sich dabei für verschiedene Fahrzeuge oder auch für das selbe Fahrzeug zu verschiedenen Zeitpunkten unterscheiden. Sei f_v eine Metrik, die den Bedarf an Flexibilität für ein Fahrzeug v angibt. Dann kann f_v interpretiert werden als die Wichtigkeit des Fahrzeuges für zukünftige PALS-Insertions zum aktuellen Zeitpunkt. Die Hauptaufgabe der Arbeit besteht darin den Bedarf an Flexibilität f_v für alle Fahrzeuge zu bestimmen.

Voronoi Region als Flexibilitätsmaß

Für die Bestimmung des Bedarfs an Flexibilität f_v wird für jedes Fahrzeug eine Voronoi-Zelle $VR(v)$ berechnet. Die grundlegende Idee für die Berechnung des Bedarfs an Flexibilität f_v beruht auf der Beobachtung, dass der Bedarf an Flexibilität lokal räumlich begrenzt ist. Betrachte zwei Fahrzeuge v und v' und zwei mögliche PALS-Insertions $\iota = (v, r, s_k(v), s_k(v))$ und $\iota' = (v', r, s_k(v'), s_k(v'))$ für einen zukünftigen Request r. Beachte, dass bei PALS-Insertions die Detour der Distanz vom letzten Stopp des Fahrzeuges zur

Pickup-Position p und der Distanz von p zur Dropoff-Position d entspricht und, dass die Fahrzeuge nach dem letzten Stopp keine Fahrgäste führen. LOUD wählt also die Insertion mit der geringeren Detour aus unter Berücksichtigung der Einschränkungen. Demnach ist nur das Fahrzeug mit der geringeren Distanz vom letzten Stopp des Fahrzeuges zur Pickup-Position eine gute Wahl für LOUD, solange die Ankunftszeit des Fahrzeuges bei der Pickup-Position p nicht die maximale Wartezeit des Fahrgastes überschreitet. Da LOUD Voronoi zusätzlich die Reisezeit und damit auch die Wartezeit in der Kostenfunktion berücksichtigt, ist nicht nur die Detour entscheidend für die Wahl einer PALS-Insertion, sondern auch die Wartezeit des Fahrgastes, die vom Stellen des Requests bis zum Eintreffen des Fahrzeuges an der Pickup-Position vergeht. Das entspricht aber der Ankunftszeit $d_v(p)$ eines Fahrzeuges v bei der Pickup-Position p. Die Ankunftszeit setzt sich aus der frühesten Abfahrtzeit des Fahrzeuges von seinem letzten Stopp $t_{dep}^{min}(s_k(v))$ und der Distanz vom letzten Stopp des Fahrzeuges bis zur Pickup-Position p des Requests zusammen. Es gilt also:

$$d_v(p) = t_{dep}^{min}(s_k(v)) + dist(s_k(v), p)$$

Sei v das Fahrzeug mit der frühesten Ankunftszeit $d_v(p)$ bei p unter allen Fahrzeugen. Dann ist v das Fahrzeug, dessen PALS-Insertion die geringste Wartezeit für den Request r aufweist. Wir können einen Knoten $v \in V$ also dem Fahrzeug v zuordnen, wenn die Ankunftszeit von v bei v minimal ist für alle Fahrzeuge. Daraus ergibt sich eine Knotenmenge M_v aller Knoten, für die v die minimale Ankunftszeit aller Fahrzeuge erreicht. Wir sagen, dass alle Knoten von M_v zum Bedarf an Flexibilität für das Fahrzeug v beitragen. Die Knotenmengen M_v entspricht aber einer Voronoi-Zerlegung mit der Distanzmetrik $d_v(v)$ für alle $v \in V$ und Zentren $s_k(v)$ für alle v in der Flotte. Die Voronoi-Zelle ist dabei definiert durch die Knotenmenge $VR(v) = M_v$.

Betrachten wir noch einmal die Distanzfunktion $d_v(v)$ mit $v \in VR(v)$. Wenn sich das Fahrzeug v seinem letzten Stopp nähert, reduziert sich die Zeitspanne $\Delta(v, v)$ die das Fahrzeug benötigt, um bei v anzukommen. Beachte, dass $\Delta(v, v)$ nur solange schrumpft, bis v bei seinem letzten Stopp ankommt und damit idle wird. Dann entspricht $\Delta(v, v) = dist(s_k(v), v)$ der Distanz vom letzten Stopp des Fahrzeuges zu v. Sei v' ein weiteres Fahrzeug, sodass die Distanz von $s_k(v')$ zu v kleiner ist. Es gilt also, dass $dist(s_k(v'), v) < dist(s_k(v), v)$ und $d_{v'}(v) > d_v(v)$. Daraus folgt, dass v seinen letzten Stopp früher erreicht, als v' seinen letzten Stopp. Weil $\Delta(v, v)$ mit voranschreitender Zeit gegen $dist(s_k(v), v)$ konvergiert, existiert ein Zeitpunkt t, sodass $\Delta(v', v) < \Delta(v, v)$ und damit $d_{v'}(v) < d_v(v)$ die Ankunftszeit von v' bei v kleiner ist, als die Ankunftszeit von v bei v. Daraus folgt, dass sich die Zugehörigkeit des Knotens v zu seiner Voronoi-Zelle mit voranschreitender Zeit verändern kann.

Es gelte für v', dass v' nicht idle ist, da nur in diesem Fall der Knoten v zukünftig in der Voronoi-Zelle $VR(v')$ von v' liegen kann. Da $dist(s_k(v'), v) < dist(s_k(v), v)$, läge v anderenfalls bereits in der Zelle $VR(v')$. Sei $l[v]$ eine nach der Ankunftszeit aufsteigend sortierte Liste aller Fahrzeuge, die zukünftig die früheste Ankunftszeit bei v haben werden. Die Liste repräsentiert die Pareto-Front im Bezug auf die Ankunftszeit $d_v(v)$ bei v und Distanz $dist(s_k(v), v)$ vom letzten Stopp von v zu v. Wir betrachten zwei aufeinander folgende Fahrzeuge in der Liste $v = l[v]_i$ und $v' = l[v]_{i+1}$. Dann lässt sich der Zeitpunkt,

an dem die Zugehörigkeit des Knotens v von ν zu ν' wechselt, berechnen mit der Formel

$$switchTime(\nu, \nu', v) = t_{dep}^{min}(s_k(\nu')) + dist(s_k(\nu'), v) - dist(s_k(\nu), v)$$

wobei $s_k(\nu')$, $s_k(\nu)$ die letzten Stopps von ν, ν' sind.

Die Formel kann von der Gleichung

$$t + dist(s_k(\nu), v) \;=\; d_{\nu'}(v) \;=\; t_{dep}^{min}(s_k(\nu')) + dist(s_k(\nu'), v)$$

abgeleitet werden. Es gilt Gleichheit genau dann, wenn zum Zeitpunkt t die Ankunftszeit von ν gleich der Ankunftszeit von ν' ist. Ab dem Zeitpunkt t gilt, dass $d_{\nu'}(v) < d_\nu(v)$, da d_ν eine monoton fallend ist.

Damit können wir also berechnen, wann die Zugehörigkeit eines Knotens v zu seiner Voronoi-Zelle $VR(\nu)$ wechselt und in welcher Voronoi-Zelle v zukünftig liegen wird.

Das Voronoi-Diagramm mit Distanzmetrik $d_\nu(v)$ und Zentren bei den letzten Stopps der Fahrzeuge ist also zeitabhängig, da die Distanzmetrik $d_\nu(v)$ zeitabhängig ist. Beschrieben wird das Voronoi-Diagramm durch die Listen $l[v]$ für alle $v \in V$, da der Listenkopf $l[v]_1$ das Fahrzeug bezeichnet, welches zum aktuellen Zeitpunkt die frühste Ankunftszeit bei v besitzt und damit für v gilt, dass $v \in VR(l[v]_1)$. Das Voronoi-Diagramm dient als Grundlage für die Berechnung der Disponibilitätsfunktion.

Beachte, dass sich das zeitabhängige Voronoi-Diagramm ändert, wenn die Routen der Fahrzeuge sich durch das Einfügen einer Insertion ändert.

Disponibilitätsfunktion

Wir verwenden nun die Voronoi-Zellen $VR(\nu)$, um den Bedarf an Flexibilität eines Fahrzeuges zu berechnen. Die Disponibilitätsfunktion $dispCost(\nu)$ interpretiert die Voronoi-Zelle $VR(\nu)$ als das aktuelle Einzugsgebiet von ν. Die Funktion verwendet Metriken, wie die Größe der Zelle oder die Anzahl in naher Zukunft erwarteter Requests in der Zelle als Basis für eine Einschätzung des Bedarfs an Flexibilität von ν.

Wir gehen hierbei von einem Offline-Modell aus. Der Disponibilitätsfunktion $dispCost(\nu)$ ist also bekannt, welche Requests innerhalb von t_Δ eingehen werden und dessen Pickup-Position. Wir definieren zunächst ein paar Methoden.

req(v) Die Funktion $req(\nu)$ gebe die Anzahl an Requests r an, die innerhalb einer Zeitspanne t_Δ in der Voronoi-Zelle $VR(\nu)$ auftreten werden mit $p(r) = v \in VR(\nu)$.

relativeDemand(ν) Die Funktion $relativeDemand(\nu)$ gebe den Bedarf der Voronoi-Zelle $VR(\nu)$ relativ zum durchschnittlichen Bedarf aller Voronoi-Zellen im Voronoi-Diagramm an . Dafür setzt die Funktion die durchschnittliche Anzahl an Requests pro Knoten innerhalb der Zelle in Relation zur durchschnittlichen Anzahl an Requests pro Knoten im gesamten Graphen.

$$relativeDemand(\nu) = \frac{req(\nu)}{|VR(\nu)|} \;/\; \frac{\sum_{\nu'} req(\nu')}{|V|}$$

Wir verwenden vier mögliche Disponibilitätsfunktionen:

$$dispCost_1(v) = c + \frac{relativeDemand(v)}{100} \tag{3.1}$$

$$dispCost_2(v) = c + \frac{req(v)}{|VR(v)|} \tag{3.2}$$

$$dispCost_3(v) = \frac{relativeDemand(v)}{100} \tag{3.3}$$

$$dispCost_4(v) = \frac{req(v)}{|VR(v)|} \tag{3.4}$$

Die Kostenfunktion f_i bezeichnet im weiteren Verlauf der Arbeit die Kostenfunktion mit der Disponibilitätsfunktion $dispCost_i$

3.3 Graph-Voronoi-Diagramm für LOUD

In diesem Kapitel wird beschrieben, wie wir das dynamische Voronoi-Diagramm berechnen. Die Berechnung beruht dabei auf Dijkstras Algorithmus. Zunächst betrachten wir aber, wie die Zugehörigkeit eines Knotens zu seiner Voronoi-Zelle abgebildet wird.

Assignments

Wir erinnern uns, dass Dijkstras Algorithmus ein Parent-Array p und ein Distanz-Array d verwendet, um kürzeste Wege für einen Graphen $G = (V, E)$ mit nicht-negativem Kantengewicht zu berechnen. Wir nennen die Zugehörigkeit eines Knotens zu einer Voronoi-Zelle Assignment. Da sich die Zugehörigkeit eines Knoten mit der Zeit verändern kann, hält $p[v]$ eine sortierte Liste, die alle zukünftigen Assignments enthält, die entstehen werden gemäß der aktuellen Routen aller Fahrzeuge. Die Liste ist aufsteigend sortiert nach der switchTime. Eine Assignment für einen Knoten $v \in V$ ist also ein Tripel $(v, dist(s_k(v), v), switchTime)$ bestehend aus einem Fahrzeug, der Distanz vom letzten Stopp $s_k(v)$ des Fahrzeuges zu v und der $switchTime$. Der Listenkopf $(p[v])_1$ enthält die aktuelle Assignment. Aus der Sortierung ergibt sich, dass die Liste ebenfalls absteigend sortiert ist nach den Distanzen der Assignments. Das resultiert daraus, dass ein Konten $v \in V$ nur dann zukünftig zu einer Voronoi-Zelle $VR(v)$ gehören wird, wenn ein Zeitpunkt existiert, an dem v die frühste Ankunftszeit für v besitzt unter allen Fahrzeugen. Wir erinnern uns, dass die Ankunftszeit bei v sich aus der Ankunftszeit des Fahrzeuges bei seinem letzten Stopp und der Distanz vom letzten Stopp des Fahrzeuges zu v zusammensetzt.

Die switchTime von $p[v]_i$ zu $p[v]_{i+1}$ ist immer später als die Ankunft von $p[v]_i$ bei seinem letzten Stopp. Sei $j > i$ und t der Zeitpunkt, an dem das Assignment von v zu $p[v]_j$ wechselt. Dann ist $p[v]_i$ zum Zeitpunkt t bereits idle.

Nehme an, dass $dist(s_k(p[v]_i), v) < dist(s_k(p[v]_j), v)$ gilt. Dann würde zum Zeitpunkt t auch gelten, dass

$$\begin{aligned} d_{p[v]_i}(v) &= t + dist(s_k(p[v]_i), v) \\ &< t + dist(s_k(p[v]_j), v) \\ &\leq t_{dep}^{min}(s_k(p[v]_j) + dist(s_k(p[v]_j), v) \\ &= d_{p[v]_j}(v). \end{aligned}$$

Damit wäre $p[v]_j$ nicht pareto-optimal und nicht in $p[v]$ enthalten. Also muss $p[v]$ absteigend nach den Distanzen sortiert sein.

Initiales Aufbauen des Diagramms

Initial sind alle Fahrzeuge idle. Wir erinnern uns, dass ein Knoten $v \in V$ nur dann zukünftig einer anderen Voronoi-Zelle angehören kann, wenn ein Fahrzeug existiert, welches nicht idle ist. Für einen Knoten v existiert also nur initial genau eine Assignment Es gilt also, dass $|p[v]| = 1$. Daher verhält sich das Voronoi-Diagramm bei Systemstart statisch.

Im Unterschied zu Dijkstras Algorithmus wird die Kürzeste-Wege-Suche nicht nur von einem Startknoten ausgehend gestartet, sondern von einer Menge von Startknoten S. Die Menge von Startknoten S entspricht der Menge der Zentren der Voronoi-Zellen. Ausgehend von S wird eine Kürzeste-Wege-Suche durchgeführt, die simultan bei allen Zentren anfängt. Das bedeutet, dass während der Initialisierung des Dijkstras Algorithmus alle Startknoten $s \in S$ der PQ hinzugefügt werden mit Distanz 0.

Nach der Initialisierung wird der PQ-Eintrag mit der kleinsten Priorität aus der PQ entfernt und gesettlt. Als Distanz für die PQ dient hier die Ankunftszeit. Weil alle Fahrzeuge initial idle sind, hängt die Ankunftszeit eines Fahrzeuges v bei einem Knoten $v \in V$ nur von der Distanz $dist(s_k(v), v)$ vom letzten Stopp des Fahrzeuges zum Knoten ab. Sei $v \in V$ der Knoten, welcher die frühste Ankunftszeit $d_v(v)$ in der PQ hat mit dem Fahrzeug v und $u \in V$ ein Nachbar von v. Dann ergibt sich die Ankunftszeit bei u von v über v aus $d_v(u) = d_v(v) + l(v, u)$. Wenn $d_v(u)$ die frühste bisher gefundene Ankunftszeit ist, wird u der Voronoi-Zelle $VR(v)$ zugewiesen.

Dijkstras Algorithmus liefert einen Kürzeste-Wege-Wald, dessen Wurzelknoten die Zentren der Voronoi-Zellen sind, anstatt eines Kürzeste-Wege-Baumes. Ein Baum im Kürzeste-Wege-Wald entspricht dabei einer Voronoi-Zelle. Da wir allerdings nicht an den Pfaden eines kürzesten Weges interessiert sind, sondern an der Zugehörigkeit eines Knoten zu seiner Voronoi-Zelle, merken wir uns im Parent-Array nicht den Parent-Knoten eines Knoten $v \in V$ im Kürzeste-Wege-Baum, sondern das Fahrzeug v. Dieser dient als Identifikator für die Voronoi-Zelle $VR(v)$ Damit enthält das Parent-Array für jeden Knoten $v \in V$ die Zugehörigkeit von v zu seiner Voronoi-Zelle.

Einfluss einer Insertion auf das Diagramm

Wir wollen uns nun den Einfluss eine Insertion auf das Voronoi-Diagramm anschauen. Wir betrachten hierbei zwei Fälle. Zuerst betrachten wir den Fall, dass sich der letzte Stopp des Fahrzeuges durch die Insertion verändert. Im zweiten Fall verändert sich der

letzte Stopp nicht. Sei $\iota' = (v, r, i, j)$ eine Insertion, die LOUD Voronoi in die Route des Fahrzeuges v einfügt. Wir erinnern uns, dass die Ankunftszeit $d_v(v)$ von v bei v von der Distanz vom letzten Knoten zu v und von der frühsten Abfahrtzeit von v bei seinem letzten Stopp abhängt.

1. Fall Wir nehmen an, dass sich der letzte Stopp von v durch die Insertion ι verändert. Daraus ergibt sich, dass sich die Distanz vom letzten Stopp zu allen Knoten $v \in V$ und durch die Detour für ι auch die frühste Abfahrtzeit von v bei seinem letzten Stopp verändert und damit auch die Ankunftszeit $d_v(v)$. Das bedeutet, dass für alle Knoten $u \in VR(v)$ die Zugehörigkeit zur Voronoi-Zelle neu berechnet werden muss, ebenso wie die Voronoi-Zelle $VR(v)$.

2. Fall Wir nehmen an, dass sich der letzte Stopp von v durch die Insertion ι nicht verändert. In diesem Fall bleibt die Distanz vom letzten Stopp zu allen $v \in V$ gleich. Allerdings verändert sich durch die Detour für ι auch die frühste Abfahrtzeit von v bei seinem letzten Stopp. Auch hier muss für alle Knoten $u \in VR(v)$ die Zugehörigkeit zur Voronoi-Zelle und die Voronoi-Zelle $VR(v)$ neu berechnet werden.

Beide Fälle führen also dazu, dass die Voronoi-Zelle $VR(v)$ aus dem Diagramm entfernt und neu berechnet werden muss. Da das Fahrzeug v nach dem Einfügen der Insertion ι nicht mehr idle ist, können nun Knoten, die nicht in der Voronoi-Zelle von v liegen zukünftig in der Voronoi-Zelle liegen. Das Einfügen von Insertion führt also dazu, dass die Assignment-Listen nicht mehr einelementig sind. Es genügt also nicht mehr bei der Berechnung der Voronoi-Zelle nur die Ankunftszeit zu betrachten, da die Ankunftszeit sich mit der Zeit verändern kann. Hierfür muss die Distanz vom letzten Stopp zu einem Knoten $v \in V$ betrachtet werden, da ein Fahrzeug v' nur dann zukünftig die frühste Ankunftszeit haben kann, wenn die Distanz $dist(s_k(v'), v) < dist(s_k(v), v)$ vom letzten Stopp zu v für v' kleiner ist als für die aktuelle Assignment v.

Berechnung des Voronoi-Diagramms

Die Berechnung des Voronoi-Diagramm kann als eine Dijkstra-Suche mit mehreren Startknoten angesehen werden. Es werden dabei solange die Einträge der PQ mit kleinster Priorität entfernt und gesettlet bis die PQ leer ist (siehe Algorithmus 3). Als Priorität wird hierbei die arrivalTime eines Fahrzeuges v beim Knoten $v \in V$ verwendet. Wird eine Assignment (v, v) aus der PQ entfernt, werden die Nachbarknoten $w \in V$ von v überprüft, ob diese ebenfalls zur selben Voronoi-Zelle $VR(v)$ gehören könnten. Das ist der Fall, wenn die $arrivalTime(v, w)$ oder die Entfernung $dist(center(v), w)$ vom Zentrum der Zelle zu w kleiner ist, als die der aktuellen Assignment und eine bestehende Assignment die mögliche Assignment (w, v) nicht überdeckt. Ist das der Fall, wird für w eine Assignment erstellt und der PQ, sowie p hinzugefügt.

Zunächst beschreiben wir die Priority Queue (PQ). Üblicherweise priorisiert Dijkstra's Algorithmus die Knoten nach ihrer vorläufigen Distanz in der PQ. Da wir als Distanzfunktion die Ankunftszeit eines Fahrzeuges bei einem Knoten $v \in V$ verwenden, priorisieren wir die PQ-Einträge nach der Ankunftszeit. Beachte hierbei, dass die Assignments dynamisch sind. Für einen Knoten können also mehrere mögliche Assignments existieren, für welche

gleichzeitig Einträge in der PQ existieren. Damit ist ein PQ-Eintrag, welcher aus einem Knoten-Distanz-Paar besteht, nicht eindeutig einer Assignment zuzuordnen. Um einen eindeutige Zuordnung eines PQ-Eintrags zu seiner Assignment sicherzustellen, merkt sich ein PQ-Eintrag neben dem Knoten auch das Fahrzeug.

Initial sind alle $p[v]$ leer. Sei s_k der letzte Stopp eines Fahrzeuges v, dann wird $p[s_k]$ die Assignment v mit Distanz 0 hinzugefügt. Die switchTime entspricht dem aktuellen Zeitpunkt. Der PQ wird ebenfalls ein Eintrag für die Assignment mit Distanz 0 hinzugefügt. Beachte, dass initial alle Fahrzeuge idle sind.

Wird eine PQ-Eintrag (v, v) aus der PQ entfernt, werden die Nachbarknoten $w \in V$ von v überprüft, ob die Assignments (w, v) ebenfalls zur selben Voronoi-Zelle $VR(v)$ gehören könnten. Es genügt dabei nicht zu überprüfen, ob die Ankunftszeit von v bei w kleiner ist, als die Ankunftszeit der aktuellen Assignment von v bei w, da w zukünftig zu $VR(v)$ gehören könnte, wenn die Distanz von letzten Stopp von v zu w geringer ist als die Distanz vom letzten Stopp der aktuellen Assignment $p[w]_1$ zu w. Aber selbst wenn das der Fall ist, ist es möglich, dass eine zukünftige Assignment von w die Assignment (w, v) überdeckt, also eine geringere Distanz vom letzten Stopp zu w mit früherer switchTime hat.

Um zu überprüfen, ob (w, v) überdeckt wird, betrachten wir die Assignment $p[w]_j$ in der Liste $p[w]$ mit der nächst kleineren Distanz vom letzten Stopp zu w und vergleichen die switchTime. Wenn die switchTime von $p[w]_j$ vor der switchTime von (w, v), dann wird die Assignment überdeckt und nicht der Assignment-Liste hinzugefügt.

Wenn die Assignment (w, v) nicht überdeckt wird, wurde eine mögliche Assignment gefunden. Nun müssen die Assignment $p[w]$ von w überprüft werden, ob (w, v) diese überdeckt und gegebenenfalls gelöscht werden.

Wir beschreiben zunächst einige Subroutinen, die für die settle-Methode benötigt werden.

arrivalTime Die $arrivalTime$ beschreibt wann ein Fahrzeug v frühestens bei einem Knoten $v \in V$ ankommen kann, wenn das Fahrzeug vorher seinen letzten Stopp s_k, das Zentrum der Voronoi-Zelle, erreichen muss, also

$$arrivalTime(v, v) = t_{dep}^{min}(s_k^v) + dist(s_k^v, v)$$

isNotDominated Wird eine Assignment bestehend aus einem Knoten-Fahrzeug-Paar (v, v) gesettlet, werden alle Nachbarknoten w von v überprüft, ob diese auch zur Voronoi-Zelle von v gehören. Die Methode $isNotDominated(w, v)$ überprüft, ob die Assignment (w, v) eines Nachbarknotens w zur Voronoi-Zelle von v nicht von anderen bestehenden Assignment dominiert wird. Eine Assignment wird dominiert, wenn eine weitere Assignment existiert mit geringerer Distanz zu w und früherer switchTime.

addAssignment Die Methode $addAssignment(w, v, d)$ ordnet den Knoten w der Voronoi-Zelle des Fahrzeuges v zu mit der Distanz d. Dafür bestimmt die Methode in Abhängigkeit von d den Index j von $p[w]$ an der die Assignment in die Liste eingefügt werden soll und berechnet die switchTime relativ zur Assignment an Position j-1. Wenn die Assignment an den Anfang der Liste eingefügt werden soll, wird die switchTime dem aktuellen Zeitpunkt gleichgesetzt. Nachdem die Assignment $p[w]$ hinzugefügt worden ist, wird überprüft, ob diese bestehende Assignments überdeckt. Wird eine Assignment überdeckt, wird sie

aus $p[w]$ gelöscht. Beachte, dass das Löschen einer Assignment eine Neuberechnung der switchTime des nachfolgenden Assignment verursacht, da die switchTime sich auf die vorangehende Assignment bezieht.

center Sei v ein Fahrzeug mit Route $R(v) = \langle s_0, ..., s_k \rangle$, dann entspricht $center(v) := s_k$ dem letzten Stopp $s_k \in V$ des Fahrzeuges v. Beachte, dass der letzte Stopp eines Fahrzeuges dem Zentrum der Voronoi-Zelle $VR(v)$ entspricht.

Algorithmus 3 : Settle PQ-Eintrag

Input : $G = (V, E)$ mit nicht-negativen Kantengewichten $l(e) \geq 0$ für $\forall e \in E$
Data : Q : Priority-Queue, p : Assignment-Array

```
1  while Q is not empty do
2      u ∈ V, v : Fahrzeug ← Q.deleteMin();
3      for (u, w) ∈ E do
4          v_curr : Fahrzeug ← (p[w])_1.assignment;
5          if dist(s_k^v, u) + l(u, w) < dist(s_k^{v_curr}, w) oder
             arrivalTime(v, w) < arrivalTime(v_curr, w) oder center(v) = center(v_curr)
             then
6              if isNotDominated(v, w, dist(s_k^v, u) + l(u, w))) then
7                  if arrivalTime(v, w) < arrivalTime(v_curr, w) and v_curr ist idle
                     then
8                      vToQueue ← v_curr;
9                  else
10                     vToQueue ← v;
11                 if Q.contains(w, vToQueue) then
12                     Q.deleteEntry(w, vToQueue)
13                 Q.insert(w, v, arrivalTime(v, w));
14                 addAssignment(w, v, dist(s_k^v, u) + l(u, w));
```

Umgang mit gleichem Zentrum

Bisher wurde ignoriert, dass der letzte Stopp mehrere Fahrzeuge auf dem selben Knoten $c \in V$ liegen kann. Wir erinnern uns, dass ein Knoten $v \in V$ genau einer beliebigen möglichen Voronoi-Zelle zugeordnet wird, wenn mehrere Voronoi-Zentren existieren mit minimaler Distanz zu v.

Sei F_c die Menge aller Fahrzeuge, dessen letzter Stopp sich auf c befindet und $VR(v)$ die Voronoi-Zelle eines Fahrzeuges $v \in F_c$. Dann muss sichergestellt werden, dass die Voronoi-Zelle $VR(v_{max})$ des Fahrzeuges mit der kürzesten Distanz $d_{v_{max}}(c)$ zum Zentrum der Zelle alle anderen Voronoi-Zellen dominiert, da $VR(v_{max})$ die größte Ausdehnung hat. Wir wählen diese Konvention, damit in einer Voronoi-Zelle keine Löcher entstehen. Der Algorithmus hält diese Konvention auch ein, wenn $VR(v_{max})$ eindeutig ist, da die Voronoi-Zelle $VR(v_{max})$ die geringste arrivalTime für alle $w \in VR(v_{max})$ besitzt und damit

alle Voronoi-Regionen $VR(v)$ dominiert. Ist $VR(v_{max})$ nicht eindeutig, soll die Voronoi-Zelle dominieren, die zuerst dem Voronoi-Diagramm hinzugefügt wurde. Das muss in der *addAssignment*-Methode sichergestellt werden. Wir setzen dies um, indem beim Aufruf der Methode überprüft wird, ob bereits eine Assignment mit gleicher Distanz, switchTime und gleichem Zentrum existiert. In diesem Fall besitzen beide Voronoi-Zellen die gleich Ausdehnung. Die hinzuzufügende Assignment muss also hinter die bestehende Assignment hinzugefügt werden.

Voronoi-Diagramm aktualisieren

Ändert sich der letzte Stopp eines Fahrzeuges v, muss nicht das Voronoi-Diagramm neu berechnet werden. Es genügt die Voronoi-Zelle $VR(v)$ zu entfernen und bei dem neuen letzten Stopp die Voronoi-Zelle neu zu berechnen. In diesem Kapitel wird beschrieben, wie eine Voronoi-Zelle aus dem Diagramm entfernt und hinzugefügt werden kann.

Voronoi-Zelle entfernen

Um eine Voronoi-Zelle zu löschen wird zunächst der Rand der Voronoi-Zelle R_v berechnet. Beachte, dass $v \in V$ genau dann zur Voronoi-Zelle $VR(v)$ gehört, wenn die Assignment (v, v) der aktuellen Assignment $(p[v])_1$ von v entspricht. Anschließend werden alle Assignments von v gelöscht. Diese können mit einer Breitensuche auf $G = (V, E)$ ausgehend vom Zentrum von $VR(v)$ berechnet werden. Wenn ein Knoten $w \in V$ gefunden wird, welcher keine Assignment (w, v) enthält, wird von diesem Knoten ausgehend nicht weiter gesucht.

Wir verwenden den Algorithmus 3, um das durch das Löschen der Voronoi-Zelle $VR(v)$ entstandene Loch zu schließen. Für den Rand der Voronoi-Zelle bleiben die kürzesten Distanzen und damit die aktuellen Assignments auch nach dem Löschen der Zelle bestehen. Es genügt also vom Rand R_v ausgehend neue Assignments zu berechnen. Sei $r \in R_v$ ein Knoten auf dem Rand, dann werden alle Assignments $p[r]$ der PQ hinzugefügt. Beachte, dass die arrivalTime verwendet wird als Priorität für die PQ. Nun wird der Algorithmus 3 solange aufgerufen, bis die PQ leer ist. Der Algorithmus exploriert nur Knoten der Voronoi-Zelle $VR(v)$, da für Knoten $w \notin VR(v)$ bereits die kürzesten Distanzen gefunden wurden. Die Suche beschränkt sich also auf $VR(v)$.

Voronoi-Zelle hinzufügen

Analog zum Löschen einer Voronoi-Zelle verwenden wir den Algorithmus 3, um eine neue Voronoi-Zelle $VR(v)$ dem Voronoi-Graphen hinzuzufügen.

Sei v ein Fahrzeug mit der Route $R(v) = \langle s_0, ..., s_k \rangle$. Wir fügen zunächst die Assignment (s_k, v) mit der Distanz 0 dem Diagramm und der PQ hinzu mit der arrivalTime des Fahrzeuges bei s_k. Anschließend wird Algorithmus 3 solange aufgerufen, bis die PQ leer ist.

4 Evaluation

In diesem Kapitel wollen wir die Performance von LOUD Voronoi evaluieren. Dafür vergleichen wir LOUD Voronoi mit dem LOUD Algorithmus.

4.1 Aufbau der Experimente

Der Quellcode von LOUD Voronoi wurde in C++ 17 implementiert und mit dem GNU 9.3 Compiler mit Optimierungslevel 3 gebaut. Die Experimente wurden auf einem Rechner mit einem 8-Core Intel Xeon E5-4640 Prozessor mit einer Taktrate von 2,4GHz und 512GiB ECC DDR3-PC1600 Hauptspeicher durchgeführt. Wir wollen die Disponibilitätsfunktionen und dessen Einfluss auf die globale Zielfunktion evaluieren. Um nur den Einfluss der Disponibilitätsfunktion auf die Zielfunktion zu messen, haben wir die Kostenfunktion von LOUD verändert, sodass die Kostenfunktion von LOUD die Reisezeitveränderung und die Reisezeit berücksichtigt. Es gilt also $f'_{LOUD}(\iota) = f_{LOUD}(\iota) + t_{trip}(\iota) + t_{\Delta trip}(\iota)$.

Eingabe Alle Experimente verwenden das Straßennetzwerk des Open-Berlin-Scenarios [9], welches das Straßennetz von Berlin, sowie einiger wichtiger Straßen um Berlin enthält. Das Straßennetz besteht aus 73 689 Knoten und 159 039 Kanten. Weiter verwenden wir eine kleine Flotte von 500 Fahrzeugen und eine große Flotte von 5000 Fahrzeugen, dessen Startpositionen auf dem Straßennetz zufällig gezogen wurden. Die Servicezeit eines Fahrzeugs beträgt dabei 30 Stunden und beginnt mit dem Systemstart.

Für die Experimente verwenden wir unterschiedliche Request-Dichten, indem wir auf einen Zeitraum von einem Tag 10.000, 15.000, oder 20.000 Requests für die kleine Flotte und 100.00, 150.000 und 200.000 Requests für die große Flotte verteilen. Die Startzeiten der Requests werden dabei anhand einer Verteilung, welche realistische Anfragezeiten abbildet, zufällig gezogen (siehe [1] für Details). Die Pickup- und Dropoff-Positionen werden mit einer geometrischen Verteilung zufällig gezogen. Der Mittelwert dieser Verteilung ist die mittlere direkte Fahrtzeit von Pickup zu Dropoff, welche in den zugrundeliegenden Daten 11 Minuten für die große Flotte und 12 Minuten für die keine Flotte beträgt.

Parametrisierung Die Parameter für den LOUD Algorithmus entnehmen wir dem LOUD Paper [1]. Dabei wird t_{stop} auf 1 min gesetzt. Die maximale Wartezeit t_{wait}^{max} wird auf 5 min gesetzt. Für die maximale Reisezeit t_{trip}^{max} werden die Parameter α auf 1,7 und β auf 2 min gesetzt. Die Parameter für die Kostenberechnung γ_{wait} und γ_{trip} werden auf 1 und 10 gesetzt. Die Parameter für LOUD Voronoi werden analog gewählt. Zusätzlich wird die Konstante c des LOUD Voronoi für die Berechnung des Malus in LOUD Voronoi auf 15.000 gesetzt. Der Wert lehnt sich an die durchschnittliche Detour über alle Insertions an, die LOUD für das Berlin-1pct [9] Szenario gewählt hat. Für jede Disponibilitätsfunktion werden wir verschiedene γ betrachten.

4.2 Evaluation der Laufzeit

In diesem Kapitel vergleichen wir die Laufzeit von LOUD Voronoi mit den Disponibilitätsfunktionen, die in Kapitel 3 vorgestellt wurden, mit der Laufzeit von LOUD. Wir betrachten hierfür die durchschnittliche Laufzeit, die für das Finden einer besten Insertion benötigt wird. Die Laufzeit gliedert sich dabei in das Durchsuchen aller möglichen Ordinary Insertions, das Durchsuchen aller möglichen Pickup-Before-Next-Stop (PBNS) Insertions, das Durchsuchen aller möglichen Pickup-After-Last-Stop (PALS) Insertions und das Durchsuchen aller möglichen Dropoff-After-Last-Stop (DALS) Insertions. LOUD durchsucht die Insertions in dieser Reihenfolge. Wir verwenden für die Evaluation 500 Fahrzeuge und Request-Dichten mit 10.000, 15.000 und 20.000 Requests verteilt über 24 Stunden, sowie einer größeren Instanz mit 5.000 Fahrzeugen und Request-Dichten mit 100.000, 150.000 und 200.000 Requests. Wir erinnern uns, dass die Disponibilitätsfunktion nur für PALS-Insertions berechnet wird und deshalb nur für PALS-Insertions zusätzliche Rechenzeit für das Überprüfen einer Insertion aufgewendet werden muss. Beachte, dass das Bauen des Voronoi-Diagramms und das Aktualisieren der Voronoi-Zellen nicht gemessen werden beim Suchen nach der besten Insertion.

Ordinary Insertions Betrachten wir zunächst das Durchsuchen der Ordinary Insertions. Dem Diagramm 4.1 ist zu entnehmen, dass LOUD Voronoi mit allen Disponibilitätsfunktionen über alle Request-Dichten eine ähnliche Laufzeit aufweist. LOUD hingegen ist in dieser Phase über alle Request-Dichten hinweg deutlich schneller. Das ist darauf zurückzuführen, dass die Routen der Fahrzeuge mit LOUD Voronoi durch die veränderte Kostenfunktion länger sind als bei LOUD oder beim Scannen der Buckets mehr Fahrzeuge gefunden werden und damit die Anzahl zu betrachtender Insertions erhöht wird. Das belegt auch die Anzahl an Fahrzeugen, die im Durchschnitt für einen Request betrachtet werden. So werden bei 5000 Fahrzeugen und 200.000 Requests für LOUD im Durchschnitt 619 Fahrzeuge in 8268 Bucket-Einträgen, für f_1 und f_2 589 Fahrzeuge in 7981 Bucket-Einträgen, für f_3 624 Fahrzeuge in 8261 Bucket-Einträgen und für f_4 629 Fahrzeuge in 8230 Bucket-Einträgen gefunden. LOUD Voronoi erzeugt also mit f_1 und f_2 längere Routen, da weniger Fahrzeuge nach Insertions durchsucht werden. Mit den Disponibilitätsfunktionen f_3 und f_4 muss LOUD Voronoi mehr Fahrzeuge durchsuchen als LOUD. Dabei werden allerdings im Durchschnitt nicht nicht mehr Bucket-Einträge gescannt als bei LOUD. Das Verhalten für die anderen Instanzen ist ähnlich.

Pickup-Before-Next-Stop-Insertions Im Durchschnitt hat LOUD eine schlechtere Laufzeit für das Durchsuchen der PBNS-Insertions als LOUD Voronoi. Wir betrachten die Eingabe-Instanz mit 500 Fahrzeugen und 15.000 Requests. Für diese Instanz probiert LOUD im Durchschnitt 95,9 PBNS-Insertions pro Request aus. LOUD Voronoi probiert für f_1 und f_2 98,4 PBNS-Insertions aus, für f_3 98,8 PBNS-Insertions und für f_4 97,8 PBNS-Insertions. LOUD Voronoi probiert also mehr PBNS-Insertions aus als LOUD und hat dennoch eine geringere durchschnittliche Laufzeit. Das lässt sich auf die Anzahl an CH-Anfragen zurückführen. Im Schnitt führt LOUD 4,3 CH-Anfragen pro Request aus. LOUD Voronoi hingegen führt für f_1 und f_2 2,9 CH-Anfragen aus und für f_3 und f_4 3,5 CH-Anfragen. Das lässt sich vermutlich mit den längeren Routen und größeren Anzahl an Fahrzeugen, die LOUD Voronoi findet erklären. LOUD Voronoi kann also mehr Ordinary Insertions

Abbildung 4.1: Das Diagramm zeigt die Laufzeit, die für das Überprüfen aller Ordinary Insertions durchschnittlich benötigt wird bei der Suche nach der besten Insertion. Das linke Diagramm verwendet 500 Fahrzeuge und das rechte 5000 Fahrzeuge.

ausprobieren und besitzt damit auch eine erhöhte Chance eine gute Insertion zu finden. Beachte, dass LOUD die Kosten der bisher besten gesehenen Insertion verwendet, um die Anzahl nötiger CH-Suchen zu verringern. Wenn LOUD Voronoi also bessere Ordinary Insertions findet, sollten auch weniger CH-Suchen auftreten.

Pickup-After-Last-Stop-Insertions Für das Durchsuchen aller PALS-Insertions (siehe Diagramm 4.3) haben wir erwartet, dass LOUD Voronoi eine höhere Laufzeit aufweisen würde, was auch für f_1 und f_2 der Fall ist. Im Gegensatz zur Erwartung hat LOUD Voronoi mit f_3 und f_4 eine geringere durchschnittliche Laufzeit. Die Laufzeit für das Suchen nach einer PALS-Insertion wird durch den Radius der Dijkstra-Suche bestimmt. Der Radius wird dabei festgelegt durch die bisher beste gefundene Insertion. Die Anzahl an Insertions, die für einen Request überprüft werden ist ein Indikator für den Such-Radius. Für LOUD werden bei 5000 Fahrzeugen und 150.000 Requests im Durchschnitt 10,8 PALS-Insertions überprüft, für f_1 8 PALS-Insertions, für f_2 7,9 PALS-Insertions, für f_3 3,8 PALS-Insertions und für f_4 3,9 PALS-Insertions überprüft. Es liegt also nahe, dass die Such-Radien von LOUD Voronoi kleiner sind, als die Such-Radien von LOUD. Das zeigt sich in der Anzahl an PALS-Insertions die LOUD Voronoi als beste Insertion wählt. LOUD entscheidet sich bei 80,3% der Requests für eine PALS-Insertion. LOUD Voronoi entscheidet sich mit f_1 bei 75,2% und mit f_2 bei 75,6% der Requests für eine PALS-Insertion. Wir erinnern uns, dass PALS-Insertions mit f_1 und f_2 grundsätzlich benachteiligt werden. Das sorgt vermutlich dafür, dass gute PALS-Insertions nicht als beste Insertions gewählt werden und sich der Such-Radius dadurch während der Suche nicht verringert. Das würde auch die Anzahl an überprüften PALS-Insertions für f_1 und f_2 erklären. Der geringere Such-Radius bei LOUD Voronoi mit f_1 und f_2 lässt sich damit erklären, dass bereits vorher gute Insertions gefunden wurden. Mit den Disponibilitätsfunktionen f_3 entscheidet sich LOUD Voronoi bei 81,7% und mit f_3 bei 81,3% aller Requests für eine PALS-Insertion, was mehr PALS-Insertions sind als bei LOUD. Ich vermute, dass sich der Such-Radius bei LOUD Voronoi mit f_3 und f_4 beim Durchsuchen der PALS-Insertions verringert, weil früh in der Dijkstra-Suche eine neue beste Insertions gefunden werden. Wir erinnern uns, dass für die Berechnung der

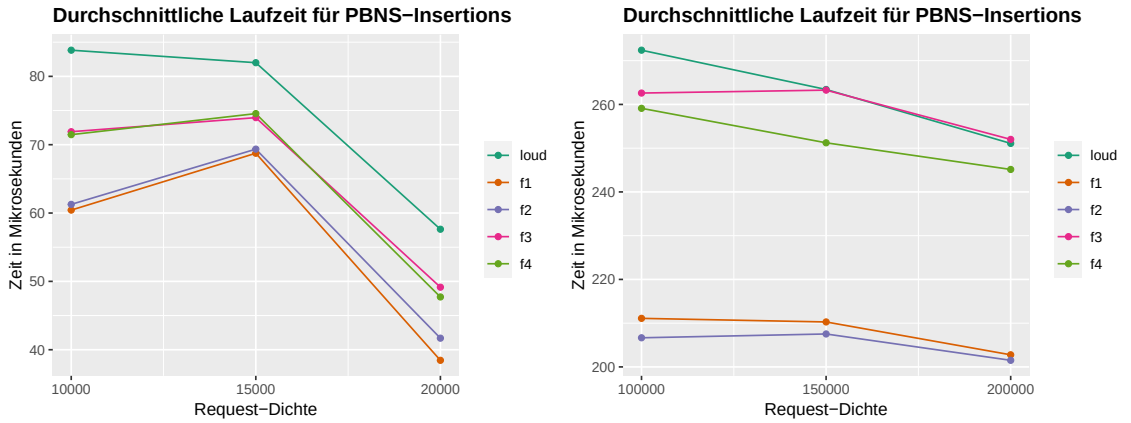

Abbildung 4.2: Das Diagramm zeigt die Laufzeit, die für das Überprüfen aller Pickup-Before-Next-Stop Insertions durchschnittlich benötigt wird bei der Suche nach der besten Insertion. Das linke Diagramm verwendet 500 Fahrzeuge und das rechte 5000 Fahrzeuge.

Kostenfunktion für LOUD Voronoi zusätzlicher Rechenaufwand benötigt wird. LOUD Voronoi überprüft zwar weniger Insertions, die Überprüfung einer PALS-Insertion ist allerdings teurer, sodass LOUD Voronoi mit f_1 und f_2 eine durchschnittlich längere Laufzeit hat als LOUD und mit f_3 und f_4 eine geringere Laufzeit hat.

Dropoff-After-Last-Stop-Insertion Aus dem Diagramm 4.5 wird ersichtlich, dass LOUD Voronoi mit f_1 und f_2 langsamer ist als LOUD und mit f_3 und f_4 schneller ist als LOUD. Das hängt mit der Anzahl der besuchten letzten Stopps bei der Suche nach möglichen DALS-Insertions zusammen. LOUD Voronoi besucht für f_1 und f_2 mehr letzte Stopps als LOUD und für f_3 und f_4 weniger letzte Stopps als LOUD.

Gesamtlaufzeit Zuletzt evaluieren wir noch die durchschnittliche Laufzeit, die benötigt wird, um einen Request einem Fahrzeug zuzuweisen und das Voronoi-Diagramm zu aktualisieren. Zunächst unterscheidet sich die durchschnittliche Laufzeit für das Aktualisieren des Voronoi-Diagramms für alle Disponibilitätsfunktionen nicht. Die Laufzeit ist hierbei für Instanzen mit vielen Fahrzeugen geringer als für Instanzen mit wenig Fahrzeugen, da die Größe der Voronoi-Zellen schrumpft mit einer wachsenden Anzahl an Fahrzeugen. Die durchschnittliche Gesamtlaufzeit ergibt sich im Wesentlichen aus der Laufzeit, die für das Finden der besten Insertion benötigt wird und dem Aktualisieren des Voronoi-Diagramms. Die Verschlechterung der Laufzeit lässt sich mit dem zusätzlichen Aufwand für das Aktualisieren des Voronoi-Diagramms erklären. Diese hatten wir auch erwartet. Trotzdem ist der Zeitaufwand, der für das Aktualisieren des Voronoi-Diagramms benötigt wird, recht gering, da es die Laufzeit von LOUD Voronoi gegenüber LOUD lediglich um ungefähr 25% erhöht. LOUD ist mit Voronoi-Diagrammen immer noch sehr schnell und kann damit als Basis für strategische Überlegungen ohne zu große Performance-Einbußen verwendet werden.

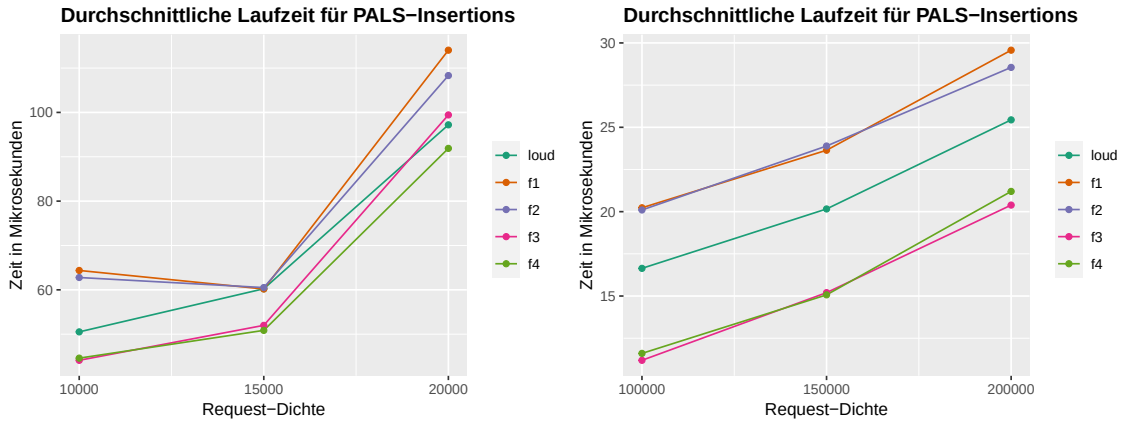

Abbildung 4.3: Das Diagramm zeigt die Laufzeit, die für das Überprüfen aller Pickup-After-Last-Stop Insertions durchschnittlich benötigt wird bei der Suche nach der besten Insertion. Das linke Diagramm verwendet 500 Fahrzeuge und das rechte 5000 Fahrzeuge.

4.3 Evaluation der Qualitätsmetriken

In diesem Kapitel evaluieren wir Qualitätsmetriken von LOUD Voronoi und vergleichen diese mit LOUD. Dafür wählen wir für jede Request-Dichte ein γ, welches die Gesamtfahrzeit minimiert. Diesen optimalen Wert des Parameters γ bestimmen wir zunächst durch Parameter-Tuning Experimente.

Gesamtfahrzeit Betrachten wir zunächst LOUD Voronoi mit den Disponibilitätsfunktionen. Im Diagramm 4.6 ist zu sehen, dass LOUD Voronoi mit den Disponibilitätsfunktionen f_1 und f_2 eine geringere Gesamtfahrzeit aufweist als mit f_3 und f_4 über alle Request-Dichten und γ-Werte hinweg. Für f_1 und f_2 lassen sich auch γ-Werte finden, sodass die Gesamtfahrzeit bei einer Request-Dichte von 10.000 und 20.000 Requests von LOUD Voronoi niedriger ist als bei LOUD. Bei einer Request-Dichte von 15.000 Request können γ-Werte gefunden werden, sodass die Gesamtfahrzeit von LOUD Voronoi der Gesamtfahrzeit von LOUD gleicht. Mit den Disponibilitätsfunktionen f_3 und f_4 kann die Gesamtfahrzeit von LOUD nur für 10.000 Request erreicht werden. Bei den Request-Dichten von 15.000 und 20.000 Requests kann LOUD Voronoi die Fahrzeit von LOUD nicht unterbieten.

Die Diagramme zeigen, dass es möglich ist γ-Werte zu finden, sodass die Gesamtfahrzeit reduziert werden kann. Wir erinnern uns, dass LOUD Voronoi nicht die Insertion mit geringsten Detour wählt, sondern neben der Detour versucht strategisch für die Zukunft zu planen. Da LOUD Voronoi geringere Gesamtfahrzeiten erreichen kann als LOUD, muss die strategische Planung von LOUD Voronoi einen Systemzustand produzieren, welcher bessere Insertions erlaubt als LOUD. Es ist dabei nicht klar erkennbar, wie sich die Gesamtfahrzeit mit wachsendem γ verändert, da dich die Gesamtfahrzeit für verschiedene Request-Dichten sehr unterschiedlich verhalten.

Für die weitere Evaluation wählen wir für jede Disponibilitätsfunktion und jede Request-Dichte einen γ-Wert mit minimaler Fahrzeit (siehe 4.6). Mit diesem γ evaluieren wir weitere Qualitätsmetriken.

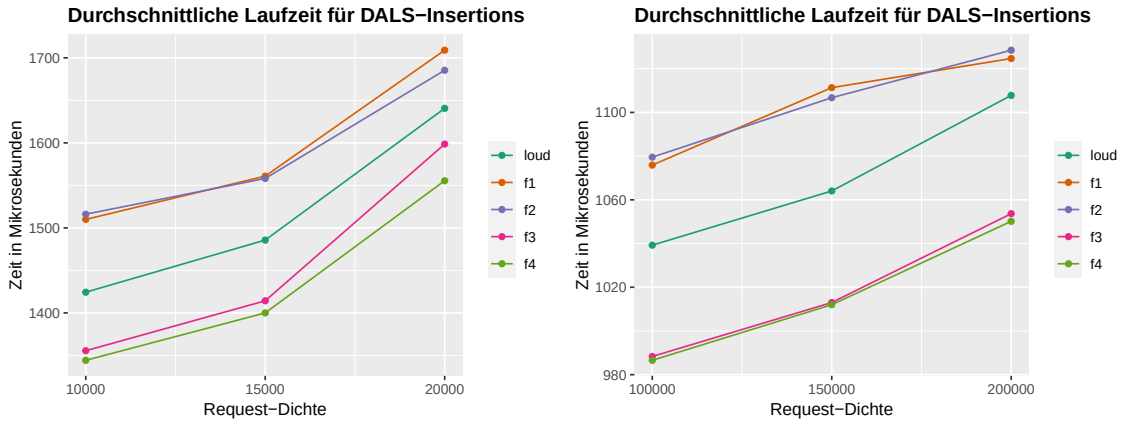

Abbildung 4.4: Das Diagramm zeigt die Laufzeit, die für das Überprüfen aller Dropoff-After-Last-Stop Insertions durchschnittlich benötigt wird bei der Suche nach der besten Insertion. Das linke Diagramm verwendet 500 Fahrzeuge und das rechte 5000 Fahrzeuge.

Disponibilitätsfunktion	10.000 Requests	15.000 Requests	20.000 Requests
f_1	0,3	0,175	0,5
f_2	0,3	0,125	0,3
f_3	0,3	0,075	0,3
f_4	0,5	0,125	0,4

Tabelle 4.1: Die Tabelle enthält die gewählten γ-Werte für jede Disponibilitätsfunktion und jede Request-Dichte.

Durchschnittliche Reisezeit pro Fahrgast Wir evaluieren die durchschnittliche Reisezeit für die gewählten γ-Werte. Dafür betrachten wir zunächst die Wartezeit. Die durchschnittliche Wartezeit von LOUD kann von keiner Disponibilitätsfunktion unterboten werden, außer von f_3. Betrachten wir allerdings das 95%-Quantil der durchschnittlichen Wartezeit, dann sehen wir, dass LOUD Voronoi im Besonderen für hohe Request-Dichten die Wartezeit reduzieren kann. So weist LOUD für 20.000 Requests die höchste Wartezeit auf.

Kommen wir nun zur durchschnittlichen Reisezeit. Hier betrachten wir zunächst die durchschnittliche Reisezeit und anschließend die durchschnittliche Wartezeit für einen Request.

Dem Diagramm 4.8 ist zu entnehmen, dass die durchschnittliche Reisezeit mit den Disponibilitätsfunktionen f_3 und f_4 reduziert werden kann. Die durchschnittliche Reisezeit bei 20.000 Requests von f_4 sinkt um 1% gegenüber LOUD. Im 95%-Quantil kann die Reisezeit sogar noch stärker reduziert werden. Hier sinkt die Reisezeit für f_4 um 3,5 %. Die Disponibilitätsfunktionen f_1 und f_2 reduzieren die Reisezeit nur im 95%-Quantil für hohe Request-Dichten.

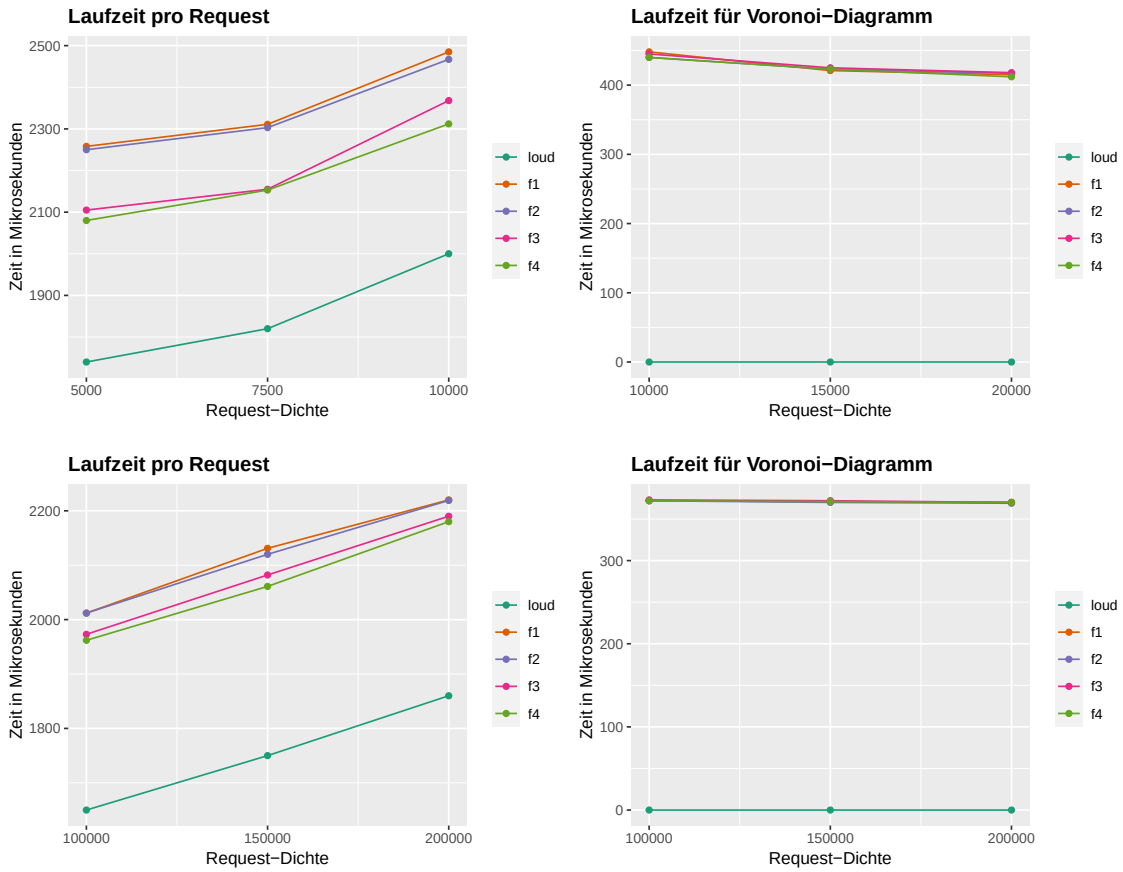

Abbildung 4.5: Das Diagramm zeigt die Laufzeit, die für das Zuweisen eines Requests durchschnittlich benötigt wird. Das obere Diagramm verwendet 500 Fahrzeuge und das untere 5000 Fahrzeuge. Links wird die Laufzeit für den gesamten Request dargestellt und rechts der Anteil der Laufzeit, welcher für das aktualisieren des Voronoi-Diagramms benötigt wird.

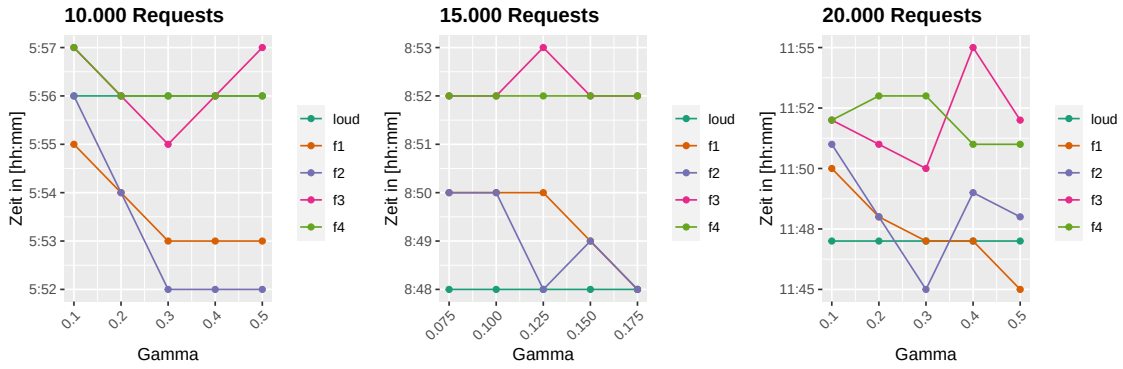

Abbildung 4.6: Das Diagramm zeigt die durchschnittliche Gesamtfahrzeit pro Fahrzeug für unterschiedliche γ für LOUD Voronoi mit den verschiedenen Disponibilitätsfunktionen und die Summe der Fahrzeiten für LOUD. Hierfür wurden 500 Fahrzeuge verwendet.

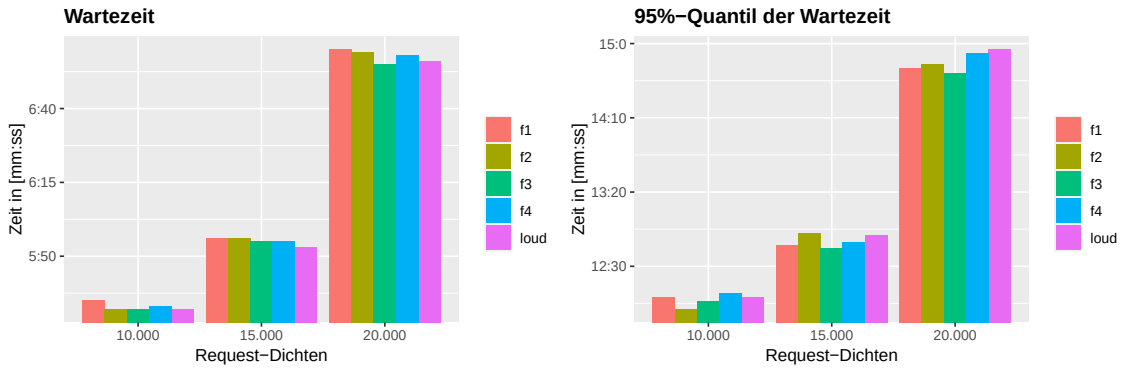

Abbildung 4.7: Das Diagramm zeigt die durchschnittliche Wartezeit, die ein Fahrgast warten muss von Zeitpunkt an dem ein Request gestellt wird bis zu seiner Abholung. Das linke Diagramm zeigt die durchschnittliche Wartezeit und das rechte Diagramm das 95%-Quantil der durchschnittlichen Wartezeit.

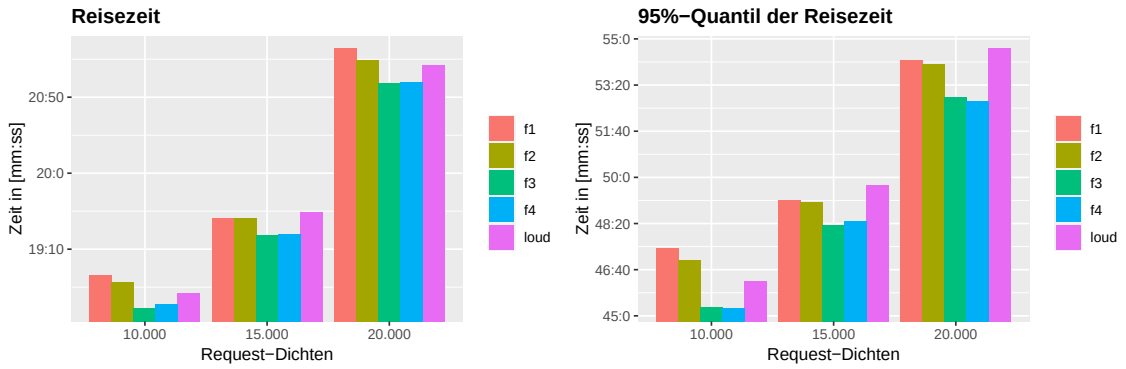

Abbildung 4.8: Das Diagramm zeigt die durchschnittliche Reisezeit eines Fahrgastes vom Zeitpunkt des Eingangs des Requests bis zum Ankommen an der Dropoff-Position. Das linke Diagramm zeigt die durchschnittliche Reisezeit und das rechte Diagramm das 95%-Quantil der durchschnittlichen Reisezeit.

5 Schlussfolgerung

In dieser Arbeit haben haben wir den LOUD Voronoi Algorithmus entwickelt, welcher Voronoi-Zellen verwendet, um die Wichtigkeit eines Fahrzeuges für zukünftige Requests zu berechnen. Ein Fahrzeug, welcher eine hohe Wichtigkeit für zukünftige Requests hat soll dabei nicht durch einen aktuellen Request gebunden werden, sodass das Fahrzeug nicht für die Request, für die eine gute PALS-Insertion existiert, zur Verfügung steht. Die Wichtigkeit eines Fahrzeuges berücksichtigen wir dazu bei der Berechnung der Kosten für eine Insertion bei der Suche nach der besten Insertion für einen Request. Dafür haben wir verschiedene Disponibilitätsfunktionen verwendet. Das Ziel der Arbeit ist es die Gesamtfahrzeit aller Fahrzeuge und die Reisezeit eines Fahrgastes zu minimieren.

Die Leistung des Algorithmus wurde mit dem Straßennetzwerk von Berlin mit 500 Fahrzeugen und unterschiedlichen Request-Dichten evaluiert. Für die Disponibilitätsfunktionen f_1 und f_2 konnte ein γ gefunden werden, sodass die Gesamtfahrzeit von LOUD Voronoi geringer ist als von LOUD. Allerdings reduzieren diese nicht die durchschnittliche Reisezeit. Für große Request-Dichten konnte im 0.95-Quantil die durchschnittliche Fahrzeit dennoch reduziert werden. Die Disponibilitätsfunktionen f_3 und f_4 reduzieren die Gesamtfahrzeit nicht, weisen aber eine geringere Reisezeit auf, sowohl in Durchschnitt als auch im 0.95-Quantil. Der Unterschied zwischen f_1, f_2 und f_3, f_4 besteht in der Konstante c, die f_3 und f_4 aufaddiert wird. Durch das generelle Benachteiligen von PALS-Insertions lässt sich also die Gesamtfahrzeit reduziert. Werden nur PALS-Insertions benachteiligt, wenn diese eine Wichtigkeit für zukünftige Requests haben, dann kann die Reisezeit reduziert werden. Dafür muss allerdings ein geeignetes γ gefunden werden.

5.1 Zukünftige Arbeit

In dieser Arbeit sind wir davon ausgegangen, dass bekannt ist, zu welchem Zeitpunkt ein Request eintrifft und damit auch die Pickup-Position des Request. Eine zukünftige Arbeit besteht darin für einen Knoten v des Straßennetzes zu schätzen wie viele Requests mit Pickup-Position $p = v$ innerhalb eines gewissen Zeitraums eintreffen werden.

Literatur

[1] Valentin Buchhold, Peter Sanders und Dorothea Wagner. „Fast, Exact and Scalable Dynamic Ridesharing“. In: *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)* (2021), S. 98–112.

[2] J.-F. Cordeau und G. Laporte. „The dial-a-ride problem: Models and algorithms“. In: *Annals of Operations Research* 153(1) (2007), S. 29–46.

[3] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* (1959), 1:269–271.

[4] Robert Geisberger u. a. „Exact Routing in Large Road Networks Using Contraction Hierarchies“. In: *Transportation Science* 46 (2012), S. 388–404. ISSN: 1526-5447.

[5] Michael Joswig und Thorsten Theobald. *Algorithmische Geometrie - Polyedrische und algebraische Methoden*. Friedr. Vieweg & Sohn Verlag, 2008, S. 83–84. ISBN: 978-3-8348-0281-1.

[6] Sebastian Knopp u. a. „Computing Many-to-Many Shortest Paths Using Highway Hierarchies“. In: *ALENEX* (2007), S. 36–45.

[7] Kurt Mehlhorn. „A FASTER APPROXIMATION ALGORITHM FOR THE STEINER PROBLEM IN GRAPHS *“. In: *Information Processing Letters 27* 7 (1988), S. 125–128.

[8] Alex Wallar u. a. „Vehicle Rebalancing for Mobility-on-Demand Systems with Ride-Sharing“. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2018), S. 4539–4546.

[9] D. Ziemke, I. Kaddoura und K. Nagel. „The MATSim Open Berlin Scenario: A multimodal agent-based transport simulation scenario based on synthetic demand modeling and open data“. In: *Procedia Computer Science* (2019), S. 870–877.