technische universität dortmund

fakultät für informatik

Labeltechnik für Anfragen in Nahverkehrsnetzen

Ausarbeitung zum Seminar *Algorithm Engineering* - 5. Februar 2016

Verfasser
Kai Sauerwald

Betreuer
Dipl.-Inf. Denis Kurz

Prof. Dr. Petra Mutzel

Basierend auf
Delling, Dibbelt, Pajor und Werneck, Public Transit Labeling, SEA 2015

Fakultät für Informatik
Technische Universtät Dortmund
Lehrstuhl 11: Algorithm Engineering
http://ls11-www.cs.uni-dortmund.de

Inhaltsverzeichnis

1 Einleitung

Algorithmen zur Pfadberechnung sind omnipräsent im modernen Leben der westlichen Gesellschaft, ob in Navigationssystemen oder auch weniger sichtbar in Sozialen Netzwerken. Die wohl prominenteste Methode zur Berechnung von kürzesten Pfaden ist der Algorithmus von Dijkstra. Allerdings ist der Algorithmus von Dijkstra für das schnelle Beantworten von Anfragen in großen Netzen, die in der Praxis weit über Zehntausende von Knoten haben, mit aktueller Hardware ungeeignet. Daher wird im *Algorithm Engineering* nach besseren Methoden für die Praxis gesucht, und es hat sich bereits ein breites Forschungsfeld etabliert [BDG+15]. Wir betrachten in dieser Ausarbeitung die Arbeit von Delling und anderen zu Nahverkehrsnetzen, insbesondere die Beantwortung typischer Anfragen. Dies kann die Berechnung von Abfahrtsprofilen sein, aber auch das Finden der schnellsten Verbindung. Dabei wird viel Wert darauf gelegt, die zugrundeliegenden Techniken zu präsentieren, die sich auch in anderen Anwendungsfeldern, wie zum Beispiel Straßennetzen, anwenden lassen.

In Kapitel 2 führen wir zunächst Objekte wie Sequenzen, Graphen und Bäume ein. Das klassische Probleme kürzeste Pfade zu berechnen wird in einigen Varianten betrachtet und der Algorithmus von Dijkstra als bekannte Lösung präsentiert. Es werden Pfadbäume eingeführt, die sich im weiteren Verlauf als nützlich herausstellen werden. In Kapitel 3 führen wir Labeling als Vorberechnungsmethode für Pfadprobleme vor, und betrachten im weiteren Hublabelings. Ein kurzer Exkurs gibt Überblick über die theoretischen Eigenschaften von Hublabelings. Die Klasse der hierarchischen Hublabelings wird in Abschnitt 3.1 eingeführt. In Abschnitt 3.2 betrachten wir wie hierarchische Hublabelings berechnet werden können, indem wir eine Approximationsmethode von Delling vorstellen werden. Nicht für alle vorgestellten Methoden sind dabei theoretische Schranken bekannt, so dass wir auch auf experimentelle Ergebnisse eingehen werden. Kapitel 4 widmet sich der Anwendung der in Kapitel 3 eingeführten Techniken auf Nahverkehrsnetze. Dazu wird zunächst eine Modellierung von Nahverkehrsnetzen präsentiert. Abschnitt 4.1 zeigt wie sich das Modell für Nahverkehrsnetze in Eingaben für Label-Algorithmen umwandeln lässt - die Technik der so genannten zeiterweiterten Graphen. Die Anfrage-Algorithmen aus Kapitel 3 lassen sich auch dann noch nicht direkt anwenden, so dass in Abschnitt 4.1.1 eine Modifikation der Anfrage-Algorithmen vorgenommen wird.

Diese Ausarbeitung bezieht sich in erster Linie auf die Arbeit von Delling, Dibbelt, Pajor und Wernecke [DDPW15], und deren gegebenen Literaturverweisen. Zu erwähnen sei, dass Kapitel 3 in vielen Teilen auf der Arbeit von Delling, Goldberg, Pajor und Werneck beruht [DGPW14].

2 Grundlagen

Im Folgenden sei $\mathbb{N} = \{1, 2, 3, 4 \ldots\}$ die Menge der natürlichen Zahlen und $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Ist M eine Menge, so bezeichnet M^* den kleeneschen Abschluss der Menge M. Ist beispielsweise $M = \{m_1, m_2, \ldots\}$ so ist $M^* = \bigcup_{i \in \mathbb{N}_0} M^i = \{\varepsilon, m_1, m_2, \ldots, m_1 m_1, m_1 m_2, \ldots\}$. Ein Element aus M^* nennen wir eine Sequenz von Objekten aus M. Ist $x = x_1 x_2 x_3 \ldots x_n$ eine Sequenz von Objekten aus M, so ist die Länge $|x| = n$. Oft werden wir nur von Sequenz sprechen, da M aus dem Kontext klar ist. Wir sagen y ist in x, kurz $y \in x$, falls es einen Index i gibt, mit $y = x_i$. Seien x und y zwei Sequenzen von Objekten aus M, so ist der Schnitt $x \cap y$ die Menge aller Elemente $m \in M$, die jeweils in x und y vorkommen, die Vereinigung $x \cup y$ die Menge aller Elemente, die in x oder in y vorkommen.

Eine (binäre) Relation R heißt partielle Ordnung, falls R reflexiv, transitiv und antisymmetrisch ist. Eine partielle Ordnung, die total ist, heißt totale Ordnung. Ist R eine (totale) Ordnung auf M, so sei $\max_R E$, mit $E \subseteq M$, das größte Element aller Elemente in E, bezüglich R (Analog für $\min_R$). Seien R_1, R_2 zwei Relationen auf derselben Menge M, so sagen wir, dass R_1 konsistent mit R_2 ist, falls für alle $a, b \in M$ gilt: $R_1(a, b) \Rightarrow R_2(a, b)$.

2.1 Graphen

Das zentrale mathematische Objekte in dieser Ausarbeitung ist der Graph. Klassische ist ein Graphen eine Strukturen mit einer Grundmenge und einer zweistelligen Relation über der Grundmenge. Meist werden wir hier aber gewichtete Graphen betrachten:

Definition 2.1 (Gewichteter Graph)
Ein Tupel $G = (V, A, l)$ wird als (gerichteter) **gewichteter Graph** bezeichnet, falls $A \subseteq V \times V$ und $l : A \to \mathbb{N}$.

Aus Gründen der Einfachheit werden wir im Folgenden gewichtete Graphen als **Graphen** bezeichnen. Sprechen wir hingegen von (klassischen) Graphen, so werden wir diese als **ungewichtete Graphen** bezeichnen. Zusätzlich identifizieren wir einen ungewichteten Graphen $G = (V, A)$ mit dem gewichteten Graphen $(V, A, \mathbb{1})$, wobei $\mathbb{1} : A \to \mathbb{N}$ die konstante Funktion ist, die jedes Element aus A auf 1 abbildet.

Sei $G = (V, A, l)$ ein Graph. Üblicherweise bezeichnet man die Elemente von V als Knoten, die Elemente von A als Kanten (engl. *arc*) und $l(e)$ als das Kantengewicht einer Kante $e \in A$. Ist A eine symmetrische Relation, so bezeichnen wir G als ungerichtet. Ein Pfad P ist eine Sequenz von Kanten $e_1 \ldots e_n$, so dass für jedes $i < n$ und $e_i = (v, v')$, $e_{i+1} = (u, u')$ gilt, dass $v' = u$. Wir sagen P ist ein Pfad von s nach t in G, falls es $u, v \in V$ gibt mit $e_1 = (s, v)$ und $e_n = (u, t)$. Die Länge eines Pfades $P = e_1, \ldots, e_n$ ist die Summe $|P| = \sum_{i=1}^{n} l(e_i)$. Die Distanz $\mathrm{dist}_G(s, t)$ zwischen zwei Knoten s und t in G ist die kleinste Länge eines Pfades von s nach t in G. Falls es keinen Pfad gibt, so ist $\mathrm{dist}(s, t) = \infty$. Mit $[s, t]$ bezeichnen wir die Menge aller kürzesten Pfade von s nach t. Ein Knoten v überdeckt (engl. *covers*) $[s, t]$, kurz $v \in [s, t]$, falls es einen kürzesten

Pfad $P \in [s,t]$ gibt mit $v \in P$. Ein Graph (V', A', l') heißt **Untergraph**[1] von G, falls $V' \subseteq V$ und A' alle Kanten zwischen Elementen aus V' enthält, die auch A enthält und $l'(e) = l(e)$ für alle $e \in A'$. Für $V' \subseteq V$ bezeichne $G[V']$ den eindeutig bestimmten Untergraph (V', A', l') von G.

2.2 Kürzeste Pfade-Probleme

Wir betrachten Varianten des wohlbekannten Problems, den kürzesten Pfad zu berechnen. Darunter fällt, den kürzesten Weg zwischen zwei Knoten s und t zu berechnen, was wir hier als s-t-SHORTESTPATH-Problem bezeichnen wollen. Aber auch das Problem für einen Graphen die Menge aller Pfade zwischen einer Knotenmenge und einer anderen Knotenmenge zu berechnen. Das Problem für einen gegeben Knoten s die Distanz zu allen anderen Knoten zu berechnen wollen wir hier als das SHORTESTPATH-Problem bezeichnen[2]:

Problem (SHORTESTPATH-Problem)
Gegeben: Graph $G = (V, E, l)$ und ein Knoten $s \in V$.
Ausgabe: Die Distanzen von s zu jedem anderen Knoten v in G.

Problem (s-t-SHORTESTPATH-Problem)
Gegeben: Graph $G = (V, E, l)$, zwei Knoten $s, t \in V$.
Ausgabe: Die Distanz $\text{dist}(s,t)$ eines Pfades von s nach t in G.

Die wohl bekannteste Lösung für das Problem ist der Algorithmus von Dijkstra (Algorithmus 2.1). Inspiriert durch eine Breitensuche, werden alle Knoten ihrer Entfernung nach besucht. Ein Prioritäts-Warteschlange (z.B. Min-Heap) wird als unterliegende Datenstruktur benutzt.

Algorithmus 2.1 : Algorithmus von Dijkstra

Eingabe : Graph (V, A, l), Knoten $s \in V$
Ausgabe : Distanzen von s zu allen $v \in V$
`// Initialisierung`
foreach $v \in V$ **do**
  $d[v] \leftarrow \infty$
$d[s] \leftarrow 0$
$Q \leftarrow V$
while $Q \neq \emptyset$ **do**
  $v \leftarrow \text{argmin}_{u \in Q}\, d[u]$ `// Entferne aus „Prioritäts-Warteschlange“`
  $Q \leftarrow Q - \{v\}$
  foreach $v \in \{u | (v,u) \in A,\ d[v] + l(v,u) < d[u]\}$ **do**
    $d[u] \leftarrow d[v] + l(v,u)$ `// Aktualisiere Distanz zu den Nachbarn von` v
return d

Aus dem Algorithmus 2.1 lässt sich leicht ein Algorithmus gewinnen, der das s-t-SHORTESTPATH-Problem beantwortet: Gegenüber Algorithmus 2.1 breche ab, sobald t markiert wurde, und gib den Abstand zwischen s und t aus.

Der Algorithmus von Dijkstra hat bekanntermaßen eine Laufzeit von $\mathcal{O}(|A| + |V| \log |V|)$ (in der Variante von Fredman und Tarjan [FT84]). Diese Laufzeit ist für große Graphen, wie

[1] Üblich ist auch der Begriff Teilgraph, dieser wird aber gelegentlich anders definiert. Deswegen folgen wir hier der Definition von Diestel [Die12].
[2] In der Literatur auch als s-PATH-Problem oder SINGLESOURCESHORTESTPATH-Problem bezeichnet.

Straßennetze, unakzeptabel. Daher werden in der Forschung Algorithmen untersucht, die bessere Laufzeit haben, sofern eine Vorberechnung (*preprocessing*) durch einen Algorithmus P zugelassen wird. Dies wollen wir hier allgemein als das P-SHORTESTPATH-Problem, für festes P, bezeichnen.

Problem (s-t-SHORTESTPATH-P-Problem)
Gegeben: Graph $G = (V, E, l)$, zwei Knoten $s, t \in V$ und $P(G)$.
Ausgabe: Die kleinste Länge des Pfades von s nach t in G: dist(s, t).

Eine Zusammenfassung aktueller Vorberechnungstechniken ist bei Bast *et al.* zu finden [BDG+15]. Darunter fallen beschränkte *Hop*-Techniken, Hierarchie-Techniken, Zielrichtungs-Techniken. Unter *Hop*-Techniken versteht man dabei das Vorberechnen von *"Abkürzungen"*, d.h. Aussagen über Distanzen zwischen Knoten, so dass jede Anfrage allein aus den Labels berechnet werden kann. Die Labeling-Methode, die wir im Kapitel 3 betrachten, fällt unter die beschränkten *Hop*-Techniken.

2.3 Bäume

Ein Klasse von Graphen sind Bäume. Dies sind im graphentheoretischen Sinne kreisfreie zusammenhängende ungerichtete Graphen [Die12]. Wobei ein Graph kreisfrei ist, wenn es für jeden Knoten v keinen Pfad von v nach v gibt, der Länger als 1 ist. Ein Graph heißt zusammenhängend wenn es einen Weg der Länge größer als 1 von jedem Knoten zu jedem anderen Knoten gibt. Wir betrachten hier eine Erweiterung um eine festgelegte Wurzel:

Definition 2.2 (Baum)
Ein Tupel (T, r) heißt **Baum**, falls $T = (V, A, l)$ ein kreisfreier zusammenhängender ungerichteter Graph ist und $r \in V$ (die **Wurzel**).

Wie üblich definiert die Wurzel r eine partielle Ordnung $\precsim_T$ auf den Knoten des Baums (T, r). Mit desc$(T, v) = \{w | w \precsim_T v\}$ bezeichnen wir die Menge der Vorgänger von v in T; das sind alle Knoten, die auf dem kürzesten Pfad von der Wurzel zu v liegen.

Eng verknüpft mit dem SHORTESTPATH-Problem ist die Konstruktion eines *Shortest-Path Tree* (in etwa kürzeste Pfade-Baum). Unter einem ***Shortest-Path Tree*** für einen Graphen $G = (V, A, l)$ verstehen wir einem Baum mit Knotenmenge V, so dass die Summe der Längen aller Pfade, die in r beginnen, minimal ist. *Shortest-Path Trees* sind dabei von minimalen Spannbäumen (engl. *Minimal Spanning Tree*), bei denen die Summe der Kantengewichte minimal ist, verschieden. Das Problem, einen *Shortest-Path Tree* zu berechnen, bezeichnen wir mit SHORTESTPATHTREE.

Problem (SHORTESTPATHTREE-Problem) nach [Tar82]
Gegeben: Graph $G = (V, A, l)$ und Knoten $r \in V$
Ausgabe: Baum (G', r), mit $G' = (V, A', l)$, so dass $\sum_{s \in V} \text{dist}_{G'}(r, s)$ minimal ist.

Eine einfache Lösung kann aus dem Algorithmus von Dijkstra, Algorithmus 2.1, gewonnen werden: Immer wenn in Algorithmus 2.1 der Abstand eines Knoten zu s aufgrund eines Knoten u aktualisiert wird, können wir u als Vorgänger im Pfad-Baum aktualisieren. Dies ist korrekt, denn u muss dann der Vorgänger im kürzesten Pfad zu s sein, für den bisher betrachteten Untergraphen aller markierten Knoten und u. So wird während der Berechnung des Algorithmus

von Dijkstra intern ein kürzeste-Pfade Baum konstruiert, den wir nur expliziert speichern müssen. Diesen Algorithmus bezeichnen wir mit ShortestPathTree.

Algorithmus 2.2 : ShortestPathTree

Eingabe : Graph (V, A, l), Knoten $s \in V$
Ausgabe : Kürzester-Pfad Baum zu G mit Wurzel s

1: **foreach** $v \in V$ **do**
2: $\quad d[v] \leftarrow \infty$
3: $\quad P[v] \leftarrow$ `nil`
4: $d[s] \leftarrow 0$
5: $Q \leftarrow V$
6: **while** $Q \neq \emptyset$ **do**
7: $\quad v \leftarrow \text{argmin}_{u \in Q}\, d[u]$ `// Entferne aus „Min-Heap“`
8: $\quad v \leftarrow Q - \{v\}$
9: $\quad$ **foreach** $v \in \{u | (v,u) \in A,\ d[v] + l(v,u) < d[u]\}$ **do**
10: $\quad\quad d[u] \leftarrow d[v] + l(v,u)$ `// Update Distanz zu den Nachbarn von` v
11: $\quad\quad P[u] \leftarrow v$
12: **return** (V, A', l, s) mit $A' = \{(v,u) \mid P[u] = v\}$

3 Hublabeling-Verfahren

Das Problem in einem Graphen den kürzesten Weg zwischen zwei Punkten zu berechnen ist immanent für viele Probleme. So auch für die Berechnung vieler Anfragen an Verkehrsnetzte, denkbar wäre zum Beispiel die Berechnung einer Route. Dabei ist im Verkehrsszenario das Netz oft eine Komponente, die sich verhältnismäßig langsam ändert und auch selten gegenüber den anderen Parametern eines Problems. So werden oft viele verschiedene Routen in ein und demselben festen Netz berechnet.

Im Kontext dieser Beobachtung können Vorberechnung (engl. *preprocessing*) lohnenswert sein. Wir betrachten nun die Berechnung von Labels als Vorberechnungstechnik. Ein Lableling eines Graphen ist das Versehen jedes Knoten mit Distanzinformationen zu anderen Knoten, so dass allein aus diesen Informationen eine Anfrage beantwortet werden kann. Intuitiv sind Labels als Abkürzungen oder Sprünge (engl. *hops*) im Graphen zu verstehen. Ein *preprocessing* Verfahren P das Labels berechnet, lässt sich dann danach charakterisieren, wie viele *hops* nötig sind um das s-t-SHORTESTPATH-P-Problem zu beantworten. Die simpelste Variante wäre ein 1-*Hop*-Verfahren, das für jeden Knoten die Menge der erreichbaren Knoten mit Distanz vor berechnet. Sind genau zwei „Sprünge“ von Nöten, so sprechen wir von einem *Hublabeling*-Verfahren, oder auch 2-*Hop* Labeling-Verfahren. Die Knoten-Einträge in den Labels werden dabei als ***Hubs*** bezeichnet.

Definition 3.1 (*(Hub)Labeling*-Verfahren, *Labeling*) (angelehnt an [ADGW12])
Ein ***Labeling*-Verfahren** ist eine Funktion, die jedem Graph zwei Funktionen $L_f, L_r : V \to (V \times \mathbb{N})^*$ zuweist, so dass für jedes $v \in V$ gilt:
Ist $L_f(v) = (v_1, d_1) \dots (v_n, d_n)$ und $L_r(v) = (v'_1, d'_1) \dots (v'_m, d'_m)$, so gilt

- für alle $1 \leq i \leq n$: $d_i = \text{dist}(v, v_i)$, und
- für alle $1 \leq i \leq m$: $d'_i = \text{dist}(v'_i, v)$.

Sei HL ein ***Labeling*-Verfahren**. Dann heißt HL ***Hublabeling*-Verfahren**, falls für alle Graphen $G = (V, E, l)$ das Tupel $\text{HL}(G) = (L_f, L_r)$ die so genannte **Überdeckungseigenschaft** (engl. *cover property*) erfüllt:
Für alle Knoten s, t, mit $\text{dist}(s, t) < \infty$, gibt es $(v, d_1) \in L_f(s)$ und $(v, d_2) \in L_r(t)$, so dass $v \in [s, t]$.

Kurzgefasst: Ein *Hublabeling*-Verfahren berechnet für jeden Knoten v zwei Sequenzen $L_f(v), L_r(v)$ von Knoten-Distanz-Tupeln vor, so dass jede Anfrage durch Berechnen von $L_f(s) \cap L_r(t)$ entschieden werden kann. Das Tupel $(L_f, L_r) = \text{HL}(G)$ nennen wir dann ***Hublabeling*** für den Graphen G durch HL.

Als **Größe eines *Hublabelings*** (L_f, L_r) für einen Graphen $G = (V, A, l)$ bezeichnen wir die Summe $|(L_f, L_r)| := \sum_{v \in V} |L_f(v)| + |L_r(v)|$. Die Größe $\max(L_f, L_r) := \max_{v \in V} |L_f(v)| + |L_r(v)|$ definieren wir als **maximale Labelgröße** eines *Hublabelings* (L_f, L_r) für einen Graphen $G = (V, A, l)$. Aus Gründen der Einfachheit nennen wir ein *Hublabeling* für ein Graphen G in allen nachfolgenden Abschnitten und Kapiteln *Labeling*.

Mit einem vorberechnetem Labeling lässt sich dann eine s-t-SHORTESTPATH-P-Anfragen beantworten. In der Praxis hat sich dabei gezeigt, dass falls man eine Ordnung der Knoten zugrunde

legt, sich die Anfrage effizient beantworten lässt, indem ein geordneter Lauf (engl. *sweep over*), wie im Merge von MergeSort [DGPW14, S. 2], auf den Labels von Start- und Ziel-Knoten durchgeführt wird. Dazu nimmt man an, dass die Labels schon nach dieser Ordnung sortiert vorliegen. Siehe dazu auch Algorithmus 3.1:

Algorithmus 3.1 : HublabelingQuery

Eingabe : Graph $G = (V, E, l)$, totale Ordnung $<$ auf V und $s, t \in V$ und *Labeling* (L_f, L_r) für G
Ausgabe : $\text{dist}(s, t)$

1: Annahme: $L_f(s) = (s_1, d_1)(s_2, d_2) \dots (s_n, d_n)$ ist sortiere, so dass $s_1 < s_2 < \dots < s_n$ gilt
2: Annahme: $L_r(t) = (t_1, d'_1)(t_2, d'_2) \dots (t_m, d'_m)$ ist sortiere, so dass $t_1 < t_2 < \dots < t_n$ gilt
3: $d \leftarrow \infty; i, j \leftarrow 1$ `// Initialisierung`
4: **for** $i \leq n \wedge j \leq m$ **do**
5: **if** $s_i < t_j$ **then** $i \leftarrow i + 1$
6: **else if** $s_i > t_j$ **then** $j \leftarrow j + 1$
7: **else**
8: **if** $d_i + d'_j < d$ **then** $d \leftarrow d_i + d'_j$ `// Entspricht:` $\text{dist}(s, u) + \text{dist}(u, t) < d$
9: $i \leftarrow i + 1$
10: **return** d

Algorithmus 3.1 besucht immer alle Element aus $L_f(s)$ und $L_r(t)$. Damit ist die Komplexität des Problems $\mathcal{O}(|L_f(s)| + |L_r(t)|)$. Die worst-case Komplexität des s-t-SHORTESTPATH-HL-Problems, lässt sich daher mit $\mathcal{O}(\max(HL(G)))$ angeben.

Theoretische Labelgrößen

Cohen und andere haben die Größe von *Hublabelings* im Allgemeinen untersucht [CHKZ03]: Es konnte gezeigt werden, dass es Klassen von Graphen mit Labeling-Größe $\Theta(|V| \cdot |A|^{1/2})$ gibt [CHKZ03, Korl. 5.3]. Cohen konnte sogar zeigen, dass es für jeden Graphen ein Labeling $L_{\max}$ gibt der Größe $\mathcal{O}(|V| \cdot |A|^{1/2})$.

Ebenso konnte Cohen zeigen, dass das Berechnen des optimalen Labelings für einen Graphen **NP**-schwierig ist [CHKZ03, Thm. 4.1]. Allerdings existiert nach Cohen ein Hublabeling-Verfahren, dass eine $(\log |V|)$-Approximation, bezüglich der Labeling-Größe, mit einer Laufzeit von $O(|V|^4)$ berechnet [CHKZ03, Thm. 4.2]. Diese Laufzeit ist in der Praxis leider zu groß für große Graphen, weshalb wir im weiteren Verlauf ein Verfahren von Delling betrachten werden, das schneller ist.

3.1 Hierarchische Labelings

Die theoretische Anfragekomplexität für s, t-SHORTESTPATH-P-Probleme hängt von der Größe der von P generierten Labels ab. Die Methode von Cohen liefert für einen Graphen $G = (V, A, l)$ Labelings die theoretisch garantiert höchstens um den Faktor $\mathcal{O}(\log |V|)$ größer als das optimale Labeling sind. Leider ist die Laufzeit von $\mathcal{O}(|V|^4)$ für Vorberechnungen auf großen Graphen mit aktueller Hardware unpraktikabel. Daher entwickeln wir nun eine Einschränkung der Labelings. Dafür führen wir zunächst den Begriff der Knotenordnung ein. Dieses ist eine Relationen auf den Knoten mit der intuitiven Bedeutung: Ein Knoten w ist wichtiger, im informationstheretischen Sinne, als v [DGPW14, S. 2].

Definition 3.2 (Knotenordnung $\preccurlyeq$)
Sei $G = (V, A, l)$ ein Graph und (L_f, L_r) ein Hublabeling für G. Dann definiere $v \preccurlyeq w$ genau dann, wenn $w \in L_f(v) \cup L_r(v)$.

Nun können wir hierarchische Labelings als Einschränkung definieren:

Definition 3.3 (Hierarchisches *Hublabeling*(-Verfahren)) (angelehnt an [ADGW12])
Sei HL ein *Hublabeling*-Verfahren und $G = (V, E, l)$ ein Graph mit $\text{HL}(G) = (L_f, L_r)$.
Das *Hublabeling* $\text{HL}(G)$ heißt **hierarchisch** wenn $\preccurlyeq$ eine partielle Ordnung ist. HL heißt **hierarchisch** wenn für jeden Graph G das *Hublabeling* $\text{HL}(G)$ hierarchisch ist.

Interessant sind hierarchische Labelings, die konsistent (siehe Kapitel 2) mit „einer" linearen Ordnung der Knoten sind: Sei eine lineare Ordnung gegeben, die Knoten nach ihrem Grad ordnet. So muss ein hierarchisches Labeling, das konsistent mit dieser Ordnung ist, Knoten mit höheren Grad in Labels bevorzugen. Dieses führt intuitiv zu kleineren Labels, denn Knoten mit größerem Grad haben mehr Nachbarn.

Es sind aber nicht nur Labelings gewünscht, die konsistent mit einer Ordnung sind, sondern auch solche die minimal sind. Dafür führen wir den Begriff des kanonischen Labelings ein:

Definition 3.4 (Kanonisches *Hublabeling*-Verfahren)
Sei HL ein Hublabeling-Verfahren. Sei $G = (V, A, l)$ ein beliebiger Graph und $<$ eine totale Ordnung auf V. Das Hublabeling $\text{HL}(G) = (L_f, L_r)$ heißt **kanonisch** über $<$, falls für alle Knoten $v \in V$ gilt:

- Ist $(w, d) \in L_f(v)$, so gibt es ein u, so dass $w = \max_<\{x \in [v, u]\}$ gilt.
- Ist $(w, d) \in L_r(v)$, so gibt es ein u, so dass $w = \max_<\{x \in [u, v]\}$ gilt.

HL heißt **kanonisch**, falls für alle Graphen G das Hublabeling $\text{HL}(G)$ kanonisch ist.

Kurzgefasst enthält ein kanonisches Labeling nur solche Hubs, die der größte Knoten, bezüglich $<$, auf allen kürzesten Pfaden von v zu einem Konten u sind. Es ist klar, dass ein kanonisches Labeling eindeutig und konsistent mit der Ordnung $<$ ist. Auch muss ein kanonisches Labeling hierarchisch sein, denn jede Menge $L_f(v) \cup L_r(v)$ kann nur Knoten w enthalten, für die $v \leq w$ gilt (im Zweifelsfalle nur v selbst) [ADGW12, S. 3]. Folglich gilt $v \preccurlyeq w$ nur wenn $v < w$.

Es lässt sich sogar zeigen: Ein kanonisches Labeling ist immer das kleinste hierarchische Labeling, das konsistent mit einer totalen Ordnung der Knoten ist:

Satz 3.5 [ADGW12, Thm 3.2]
Sei $G = (V, A, l)$ ein Graph. Ist (L_f, L_r) ein hierarchisches Hublabeling, so das $\preccurlyeq$ konsistent mit einer totalen Ordnung R ist, so ist das kanonische Hublabeling (C_l, C_r) eine Teilmenge von (L_f, L_r). D.h. für alle $v \in V$ gilt $C_l(v) \subseteq L_l(v)$ und $C_r(v) \subseteq L_r(v)$.

Beweisskizze nach [ADGW12, Thm 3.2]: Wir nehmen an, dass kürzeste Pfade eindeutig sind. Sei $P \in [s, t]$ der kürzeste Pfad von s nach t und $v = \max_R x \in P$. Wir zeigen, dass $v \in L_f(s)$ sein muss. Sei P' der kürzeste Pfad von s nach v. Da $\preccurlyeq$ konsistent mit R ist, muss v der einzige Hub in $L_r(v)$ sein. Die Überdeckungseigenschaft garantiert uns, dass somit v auch ein Hub in $L_f(s)$ ist. Es lässt sich analog zeigen, dass $v \in L_r(t)$ gilt. □

Dieser Beweis funktioniert unter der Annahme, dass kürzeste Pfade eindeutig sind. Es lässt sich aber auch zeigen, dass Satz 3.1 immer noch gilt, wenn man diese Annahme fallen lässt [GRS13, DGPW14, S. 2].

Aus theoretischer Sicht sind hierarchische Lablings keine sinnvolle Einschränkung (und damit auch kanonische Labelings). Goldberg konnte zeigen, dass es Klassen von Graphen gibt, in denen hierarchische Lablings immer größer als das Optimum sind [GRS13]. Allerdings kann man in Experimenten beobachten, dass hierarchische Labelings in vielen Anwendungsfällen, wie zum Beispiel Routing, klein sind.

3.2 Berechnung von Labels

Hieratische Labelings sind eine intuitive Einschränkung der Hublabelings, deren Minimal-Form sich durch kanonische Labelings charakterisieren lässt. Diese Charakterisierung macht es einfach(er) einen Algorithmus anzugeben, der Labelings berechnet, da bei der Konstruktion eines Labelings die Minimalität sehr einfach aufrechterhalten werden kann. Akiba folgend ist es Möglich wie folgt vor zu gehen [AIY13]: Ist eine totale Ordnung der Knoten gegeben, so initialisiere ein leeres Labeling und führe für jeden Knoten v in absteigender Reihenfolge einen modifizierten Breitendurchlauf durch (Algorithmus 3.3). Besucht der Breitendurchlauf einen Knoten u, so prüft der Algorithmus zunächst ob sich aus den Labels ein kürzerer Abstand zwischen v und u bestimmen lässt, als über den Breitendurchlauf errechnet wurde. Wenn ja, so ignoriere u, und die Nachbarn von u werden nicht der Queue hinzugefügt. Andernfalls wird ein neues Label eingetragen und der Abstand zu den Nachbarn von u aktualisiert, sowie die Nachbarn von u zur Queue hinzugefügt (Algorithmus 3.2). Diese Methode wollen wir PrundedLabeling, kurz PL, nennen.

Algorithmus 3.2 : PrunedBFS$_x$ (angelehnt an [AIY13, S. 4])

input : Graph $G = (V, A, l)$, Knoten $v_k \in G$ und (L_f^{k+1}, L_r^{k+1})
output : Menge L_k von partiellen Labels

1: $Q \leftarrow \{v_k\}$
2: $P[v_k] \leftarrow 0$
3: $P[v] \leftarrow \infty$ für alle $v \in V \backslash \{v_k\}$
4: $L_x^k[v] \leftarrow L_x^{k-1}[v]$ für alle $v \in V$
5: **while** `nonempty`(Q) **do**
6: $\quad u \leftarrow \text{argmin}_{v \in Q} P[v]$ `// Entferne aus „Prioritäts-Warteschlange“`
7: $\quad Q \leftarrow Q - \{u\}$
8: $\quad$ **if** $\min\{\delta_{sv} + \delta_{vt} | (v, \delta_{sv}) \in L_x^{k+1}[v_k], (v, \delta_{vt}) \in L_{\overline{x}}^{k+1}[u], \overline{x} \neq x\} > P[u]$ **then**
9: $\quad\quad L_x^k[u] \leftarrow L_x^{k+1}[u] \cup \{(v_k, P[u])\}$
10: $\quad\quad$ **forall** $(u, w) \in A$ *mit* $P[w] \geq P[u] + l(u, w)$ **do**
11: $\quad\quad\quad P[w] \leftarrow P[u] + l(u, w)$
12: $\quad\quad\quad Q \leftarrow Q \cup \{w\}$
13: **return** L_x^k

Die Korrektheit des PL-Algorithmus (Algorithmus 3.3) ist durch Zeile 7 im Algorithmus 3.2 gegeben. Ein Beweis der Korrektheit von PL ist bei Akiba für ungerichtete Graphen zu finden. Das durch PL berechnete Labeling ist garantiert ein minimales Labeling, in dem Sinne, dass das Entfernen eine Eintrags aus einem Label für einen Knoten die Überdeckungseigenschaft verletzten würde. Das Labeling ist also inklusionsminimal aber nicht zwingend das kleinste.

Satz 3.6 [AIY13, Thm. 4.2]
Sei $G = (V, E, l)$ ein Graph.
Für jede totale Ordnung $<$ auf V ist $PL(G, <)$ ein inklusionsminimales Labeling.

Algorithmus 3.3 : PL (PrundedLabelingCalculation)

Eingabe : Graph $G = (V, E, l)$ mit $V = \{v_1, \ldots, v_n\}$ und eine totale Ordnung $<$ auf V
Ausgabe : Labeling (L_f, L_r)

1: Sei O.B.d.A. $v_1 < v_2 < \ldots < v_n$
2: $G' \leftarrow \texttt{transpose}(G)$ `// Kanten umdrehen`
3: $L_r^{n+1}[v], L_f^{n+1}[v] \leftarrow \emptyset$ für alle $v \in V$
4: **for** $k = n, \ldots, 2, 1$ **do**
5: $\quad L_f^k \leftarrow \texttt{PrundedBFS}_f(G', v_k, L_f^{k+1}, L_r^{k+1})$
6: $\quad L_r^k \leftarrow \texttt{PrundedBFS}_r(G, v_k, L_f^{k+1}, L_r^{k+1})$
7: **return** $\texttt{toSeq}(L_f^1, L_r^1)$ `// Menge umwandeln in einer nach < geordneter Sequenz`

Auch ist das berechnete Labeling hierarchisch: Dazu reicht zu sehen, dass für jeden Pfad in $[s, t]$, nur der größte Hub, bezüglich der Ordnung der Knoten, in $L_f(s)$ und $L_r(t)$ eingetragen wurde. Wird in der i-ten Runde von PL, bei der Betrachtung des Knoten v_i, ein Eintrag im Labeling für einen Knoten u hinzugefügt, so lagen alle größeren Knoten nicht auf einen kürzesten Pfad von v_k nach u [DGPW14, S. 2f]. Also stellt die Reihenfolge in der die Knoten in PL betrachtet werden (und Zeile 7 im Algorithmus 3.2) sicher, dass das Labeling hierarchisch ist.

Satz 3.7 [DGPW14, Lem. 2.1]
Sei $G = (V, E, l)$ ein Graph. Für jede totale Ordnung $<$ auf V ist $PL(G, <)$ ein hierarchisches Hublabeling.

Insgesamt erhalten wir aus der Minimalität und der Hierarchie-Eigenschaft, dass ein durch PL berechnetes Labeling kanonisch sein muss:

Korollar 3.8
Sei $G = (V, E, l)$ ein Graph. Für jede totale Ordnung $<$ auf V ist $PL(G, <)$ ein kanonisches Labeling.

Aus den theoretischen Untersuchungen von Akiba ist bekannt, dass PL auf Graphen mit beschränkter Baumweite w Labelings der Größe $\mathcal{O}(w|V| \log |V|)$ berechnet [AIY13, Thm. 4.4]. Allerdings konnten sowohl Dellung, als auch Akiba, keine allgemeinen theoretischen Aussagen über die Größe der durch PL berechneten Labelings angeben. Aber dass die Größe der Labels von der totale Ordnung abhängt ist unbestreitbar. Dies wollen wir im folgenden Abschnitt betrachten.

Path-Greedy Ordnungen

Mit dem PL-Algorithmus lassen sich kanonische Labelings für einen Graphen berechnen. Die Qualität dieses Labelings ist abhängig von der auf den Knoten verwendeten Ordnung. Wir geben nun eine Methode an mit der sich Ordnungen konstruieren lassen, die in Experimenten kleine Labelings erzeugt hat. Diese totale Ordnung der Knoten werden wir die ***path-greedy* Ordnung** $<_{\text{pg}}$ nennen, da sie eine einfache Idee implementiert: Der i-te Knoten der Ordnung ist derjenige, der am meisten kürzeste Pfade überdeckt, die von den größeren Knoten, bezüglich $<_{\text{pg}}$, nicht überdeckt werden [DGPW14, S.3]. Dafür berechnet der Algorithmus für jeden Knoten s im Graphen einen *Shortest-Path Tree* T_s. Die Anzahl der kürzesten Pfade, die ein Knoten v überdeckt, entspricht dann genau der Summe der Anzahl von Nachfolger, die v in jedem *Shortest-Path Tree*

T_s hat [DGPW14]. Dies gilt allerdings nur wenn kürzeste Pfade eindeutig sind, weshalb wir also im Allgemeinen eine Näherung berechnen. Wir können nun den Knoten bestimmen, der die meisten Pfade (näherungsweise) überdeckt. Danach entfernen wir alle überdeckten Pfade aus den *Shortest-Path Trees* und iterieren dieses Verfahren, bis alle Knoten betrachtet wurden. Für Details siehe Algorithmus 3.4.

Algorithmus 3.4 : TD (*top-down*-Algorithmus)

Eingabe : Graph $G = (V, A, l)$ mit $V = \{v_1, \ldots, v_n\}$
Ausgabe : Totale Ordnung $<_{pg}$ auf V

```
1:  <_pg ← ∅
2:  Q ← V
3:  T[s] ← ShortestPathTree(G, s) für alle s ∈ V      // Siehe Algorithmus auf Seite 5
4:  while Q ≠ ∅ do
5:      v ← argmax_{v∈Q} Σ_{s∈V} |desc(T[s], v)|      // Bei Gleichstand wähle zufällig
6:      <_pg ← <_pg ∪ {(s, v) | s ∈ Q}                // v ist größer als alle anderen in Q
7:      Q ← Q − {v}
8:      foreach s ∈ V do
9:          Entferne Teilbaum mit Wurzel v aus T[s]
10: return <_pg
```

Es lässt sich zeigen, dass sich der *top-down*-Algorithmus (kurz TD, Algorithmus 3.4) in Zeit $\Theta(n^2 m \log n)$ und Speicher $\Theta(n^2)$ für einen Graphen mit n Kanten und m Knoten ausführen lässt[1]: Die Berechnung der *Shortest-Path Trees* in der Initialisierung ist das n-malige ausführen des Algorithmus von Disjkstra. Bleibt das geschickte Berechnen der Aktualisierungen in der **while**-Schleife: Die Anzahl der Pfade die jeder Knoten (näherungsweise) überdeckt lässt sich einmalig in $\mathcal{O}(n^2)$ berechnen, so das wir während der Aktualisierung nur die Anzahl der überdeckenden Pfade für jeden Knoten korrigieren müssen. Es lässt sich beobachten, dass die Vereinigung aller Teilbäume mit Wurzel v genau der Baum $T[v]$ ist. Der Korrekturwert eines Knoten w im *update* entspricht in diesem Fall der Anzahl der Vorfahren von w in $T[v]$. Denn diese Knoten sind genau diejenigen, in denen sich kürzeste Pfade von v und w „überlappen“. Dieser Wert lässt sich gleichzeitig für alle Knoten durch eine *bottom-up* Traversierung in $\mathcal{O}(m \cdot n \log n)$ berechnen. Da wir dies n-mal durchführen, ist die Laufzeit von TD insgesamt $\Theta(n^2 m \log n)$ [ADGW12, Thm. 4.2], wobei sich die untere Schranke durch weitere Argumente ergibt. Dabei braucht der Algorithmus $\theta(n^2)$ Speicher.

Der TD-Algorithmus kann Grundsätzlich so erweitert werden, dass er direkt Labels berechnet. Wählen wir den nächsten Knoten aus, der am meisten Pfade überdeckt, so fügen wir ihn den Labels hinzu. Allerdings vernachlässigt der Algorithmus die Fälle, in denen die kürzesten Pfade nicht eindeutig sind [DGPW14, Abs. 3.2]. Da diese bei der Berechnung des *Shortest-Path Trees* verloren gehen. Daher wird dieser Algorithmus nur zur Label-Berechnung genutzt.

Sampling

Nutzen wir den TD-Algorithmus um gute Ordnungen vorauszuberechnen und dann den PL-Algorithmus, so erhalten wir den Gesamtalgorithmus den Algorithmus, den Delling als HybPG (*hybrid and path-greedy*) bezeichnet [DGPW14, S. 4].

[1] In [DGPW14] schreibt Delling, dass sich dieser Algorithmus in $\mathcal{O}(mn \log n)$ Zeit ausführen lässt. Dies widerspricht aber der Originalquelle [ADGW12], die von $\mathcal{O}(n)$ mal die Laufzeit von Dijkstra spricht. Daher geben wir hier zweitere Laufzeit an.

	label size		preprocessing [s]				space [MiB]				query [µs]			
instance	PLL	RXL	PLL	Tree	RXL	CRXL	PLL	Tree	RXL	CRXL	PLL	Tree	RXL	CRXL
Gnutella*	644×16	791	54	209	307	451	209	68	95.7	49.1	5.2	19.0	7.1	45.9
Epinions*	33×16	118	2	128	31	39	32	42	19.1	7.7	0.5	11.0	1.1	4.1
Slashdot*	68×16	219	6	343	85	110	48	83	37.4	17.8	0.8	12.0	1.7	8.0
NotreDame*	34×16	25	5	243	18	22	138	120	22.9	11.9	0.5	39.0	0.2	1.0
WikiTalk*	34×16	113	61	2459	1076	1278	1000	416	560.8	86.5	0.6	1.8	1.0	3.4
Skitter	123×64	273	359	–	2862	3511	2700	–	1074.6	316.7	2.3	–	2.3	20.6
Indo*	133×64	43	173	–	173	201	2300	–	158.6	90.2	1.6	–	0.5	1.8
MetroSec	19×64	116	108	–	2300	2573	2500	–	592.8	207.7	0.7	–	0.8	3.6
Flickr*	247×64	360	866	–	5888	7110	4000	–	1794.6	345.9	2.6	–	2.8	19.9
Hollywood	2098×64	2114	15164	–	61736	75539	12000	–	5934.3	2050.0	15.6	–	13.9	204.0
Indochina*	415×64	91	6068	–	8390	8973	22000	–	1978.8	876.8	4.1	–	0.9	3.9

Tabelle 3.1: Tabellarischen Übersicht der experimentellen Ergebnisse zum Vergleich von PL (hier PLL) und SamPG (RXL) aus [DGPW14, Tab. 2].

Beide Algorithmen haben allerdings einen Speicher- und Zeit-Bedarf von $\Omega(n^2)$, so dass dies auch auf HybPG zutrifft. Daher hat Delling einen Sampling-Algorithmus vorgeschlagen, den wir hier kurz skizzieren wollen: Der Algorithmus berechnet nicht alle *n Shortest-Path Trees*, sondern zunächst nur $k \ll n$ viele (die zufällig bestimmt werden). Dann verfährt der Algorithmus genauso wie der HybPG-Algorithmus. Bis zu dem Punkt, an dem die *k Shortest-Path Trees* zu klein werden, um zuverlässige Informationen zu berechnen. Dieses Problem löst Delling wie folgt: Es werden zufällig neue (noch nicht betrachtete) Knoten gewählt und für diese dann die *Shortest-Path Trees* konstruiert. Dabei werden die *Shortest-Path Trees* so konstruiert, dass Knoten für die bereits Informationen vorliegen nicht mit einbezogen werden [DGPW14]. Der Algorithmus PrunedBFS wird dafür für jeden neu gewählten Knoten aufgerufen um die Labels neu zu berechnen. Diese Methode erlaubt es den Speicher- und Zeit-Bedarf klein zu halten. Da diese Methode eine recht hohe Varianz aufweist, wird dieses Verfahren konstant oft parallel ausgeführt um Ausreißer auszugleichen. Die Parameter für das Verfahren hat Delling experimentell bestimmt [DGPW14, S.5]. Dieser Algorithmus soll nach Delling SamPG (*sampling path-greedy*) heißen.

Experimentelle Ergebnisse

Es sind keine (nicht trivialen) theoretischen Garantien für den Algorithmus SamPG bekannt. Allerdings hat Delling SamPG in Experimenten mit dem Algorithmus PL verglichen: Tabelle 3.1 gibt einen Überblick über die Experimente. PLL bezeichnet dabei einen Variante von PL, bei denen die Knoten nach ihrem Grad geordnet werden. Mit RXL bezeichnet Dellings eine Implementierung, die nach ausführen von SamPG eine Komprimierung der Labelings vornimmt (siehe Abschnitt 3.3). Das Darstellungsformat $T \times B$ der Labelgröße bei PLL ist die tatsächliche Anzahl an Labeln T. B bezeichnet die Größe eines Parameters für eine Optimierung, auf die wir hier nicht eingehen. Die *lables sizes* ist die Angabe der mittlere Größe der Labels. Zusammenfassend zeigen sowohl PL als auch SamPG auf unterschiedlichen Graphen stark abweichende Labeling-Größen. In einigen Instanzen, wie `Epinions` und `Slashdot`, ist die mittlere Labelgröße bei PL besser als bei SamPG. Allerdings gibt es auch Instanzen wie `Indo` und `Indochina` bei denen SamPG im Mittelwert kleinere Labels erzeugt. Es sei erwähnt das der Algorithmus PLL (in optimierter Variante) nicht auf gewichteten Graphen funktioniert [AIY13, Abs. 6]. Somit ist SamPG insgesamt breiter anwendbar.

instance						degree		SamPG		RXL		CRXL	
type	name	n	m/n	d	w	prep [s]	lab	prep [s]	lab	[MiB]	[µs]	[MiB]	[µs]
sensor	rgg20	1048576	13.1	○	○	2804	1135.7	977	220.0	806.5	2.0	167.3	23.4
	rgg20-w	1048576	13.1	○	●	52962	5502.7	3608	588.8	3154.3	4.9	436.4	76.1
roads	fla-t	1070376	2.5	○	●	1321	791.8	103	41.4	260.9	0.5	55.0	3.4
	eur-t	18010173	2.3	●	●	–	–	8364	82.4	17202.8	0.8	1589.3	13.3
	eur-d	18010173	2.3	●	●	–	–	18664	163.1	33059.5	1.5	2184.2	32.1

Tabelle 3.2: Ausschnitt aus der tabellarischen Übersicht der experimentellen Ergebnisse zum Vergleich der Kompressionsmethoden aus [DGPW14, Tab. 1].

3.3 Kompression der Labels

Mit dem Algorithmus HublabelingQuery (Algorithmus 3.1) können s-t-SHORTESTPATH-Anfragen beantwortet werden, wobei die Laufzeit allein von der Größe der Labels abhängt. Für große Graphen ist allerdings nicht nur die Anzahl der Hubs praxisrelevant, sondern auch die Darstellung im Arbeitsspeicher, und der damit verbundene Speicherverbrauch. Daher werden Labelings in vielen Implementierungen komprimiert [DGPW14, DDPW15, ADGW12, AIY13, AIKK14]. Dabei spielen zusätzliche Überlegungen eine Rolle, auf die wir nicht weiter eingehen werden: Wie können die Labels/Knoten so effizient angeordnet werden, dass schnelle Dekompression möglich ist? Skaliert die Kompressionsmethode mit wachsender Größe der Labelings? ...

Dass sich Kompression lohnen kann, sei durch ein Beispiel belegt [DGPW14, S. 8]: Delling hat in Experimenten zwei Darstellungsarten (Kompressionsverfahren) verglichen, in dem er die durch SamPG (siehe Abschnitt 3.2) berechneten Labelings durch diese komprimiert hat. RXL nutzt ein Deltakompressionsverfahren zur Kompression der durch SamPG berechneten Labels. Bei diesem Verfahren wird ein Label für eine Knoten wird nicht durch eine Sequenz von Hub-Distanz Tupeln repräsentiert, sondern jeweils nur durch die Differenz zum vorhergehenden Element in der Sequenz. CRXL verwendet zur Kompression weitere Kompressionstechniken auf die wir hier nicht eingehen werden.

In den Experimenten hat sich gezeigt, dass CRXL die Anfragezeit um 20% bis 50% verschlechtert [DGPW14, S. 9]. In einigen Experimenten sogar um den Faktor 15. Hingegen ist die Kompression bei CRXL gegenüber RXL um bis zu einem Faktor 16 besser. Nennenswert sei hier, dass auf Straßendaten die Darstellungsgröße der Label von etwa 33GB auf etwa 2.2GB gesenkt werden konnte. In Tabelle 3.2 ist ein Ausschnitt aus [DGPW14, S. 8] zu finden, der die Ergebnisse veranschaulicht.

4 Nahverkehrsnetze

Wir betrachten als Setting Nahverkehrsnetze, deren typische Komponenten Haltestellen und Linien sind. Die Fahrpläne in solchen Netzen sind von ihrer Natur aus aperiodisch, da Linien in der Praxis oft unregelmäßig bedient werden. Beispielsweise werden in den Abendstunden einige Linien seltener bedient, als während der Mittagsstunden. Dieses System gilt es im folgenden zu Modellieren:

Definition 4.1 (Fahrplan) (angelehnt an [DDPW15])
Ein Tupel $\mathcal{L} = (S, E, T, F)$ heißt **Pseudo-Fahrplan** (engl. *Timetable*), falls wie folgt gilt:

- S ist eine endliche Menge von **Haltestellen** (engl. *Stops*),
- $E \subseteq S \times \mathbb{N}$ ist eine endliche Menge von **Ereignissen** (engl. *Events*), mit $\texttt{stop}(e) = s$ und $\texttt{time}(e) = t$ für jedes Ereignis $e = (s, t) \in E$,
- $T \subseteq E^*$ ist eine endliche Menge von **Fahrten** (engl. *Trips*), mit $\texttt{stop}(t) = s$, $\texttt{dest}(t) = d$ und $\texttt{dur}(t) = x' - x$ für eine Fahrt $t = (s, x) \dots (d, x')$, und
- $F \subseteq S \times \mathbb{N} \times S$ ist eine endliche Menge von **Fußwegen** (engl. *Footpaths*), mit $\texttt{stop}(f) = s$, $\texttt{dest}(f) = d$,$s \neq d$ und $\texttt{dur}(f) = t$ für jeden Fußweg $f = (s, t, d) \in F$.

Ist $\mathcal{L} = (S, E, T, F)$ ein Pseudo-Fahrplan, so bezeichnen wir ihn als **Fahrplan**, falls gilt:

- Zu jedem Ereignis $e \in E$ existiert eine Fahrt t mit $e \in t$
- Ist $e_0 \dots e_n \in T$ eine Fahrt, so muss
  1. $\texttt{stop}(e_0) \neq \texttt{stop}(e_1)$ und $\texttt{stop}(e_{n-1}) \neq \texttt{stop}(e_n)$ gelten,
  2. ebenfalls $e_2 \dots e_n \in T$ und $e_0 \dots e_{n-1} \in T$ gelten,
  3. und für jedes $i \in \{1, \dots, n-1\}$ muss gelten:
     - $\texttt{time}(e_{i-1}) \leq \texttt{time}(e_i) \leq \texttt{time}(e_{i+1})$ gilt, und entweder
     - gilt $\texttt{stop}(e_{i-1}) = \texttt{stop}(e_i)$ und $\texttt{stop}(e_i) \neq \texttt{stop}(e_{i+1})$
     - oder es gilt $\texttt{stop}(e_{i-1}) \neq \texttt{stop}(e_i)$ und $\texttt{stop}(e_i) = \texttt{stop}(e_{i+1})$.

Die Pseudo-Fahrpläne entsprechen weitgehend den Fahrplänen bei Delling[1] [DDPW15, S. 274]. Fahrpläne unterschieden sich von Pseudo-Fahrplänen dadurch, dass in Pseudo-Fahrplänen für ein Nahverkehrsnetz untypische Situationen nicht modelliert werden können, wie Fahrten in die Vergangenheit. Das ist in anderen Szenarien nicht unbedingt unlogisch: Beispielsweise im Flugverkehr kann dies beim Wechsel der Zeitzone geschehen. Ebenso sollen Fahrgäste an jeder Station aussteigen können; dies wird durch die Existenz von *„Teilfahrten“* sichergestellt (2. Punkt). Zudem fordern wir für Fahrten, dass an jeder besuchten Station aufeinanderfolgend ein Ankunfts-Ereignis und ein Abfahrts-Ereignis vorhanden ist (mit Ausnahme für die Start und End-Haltestelle).

[1] Die Modellierung basiert auf der Arbeit von Delling, wurde aber ergänzt, da Delling in einigen Punkten ungenau ist.

In diesem Modell können wir nun den Begriff der Verbindung und der Reise präzisieren:

Definition 4.2 (Verbindung, Anschluss und Reise)
Sei $\mathcal{L} = (S, E, T, F)$ ein Fahrplan.
Ein Tupel $(x, y) \in E \times E$ heißt **Verbindung** von $\texttt{stop}(x)$ nach $\texttt{stop}(y)$, falls es eine Fahrt $e_1 \dots e_n \in T$ gibt mit $x = e_i$ und $y = e_{i+1}$.

Ein Tupel $(e_1, e_2) \in E \times E$, mit $e_1 = (s_1, t_1)$, $e_2 = (s_2, t_2)$ und $t_1 \leq t_2$, heißt

- **Haltestellen-Anschluss**, falls $s_1 = s_2$, oder
- **Fußweg-Anschluss** über f, falls $f \in F$ und wie folgt gilt:

$$\texttt{stop}(f) = s_1, \texttt{dest}(f) = s_2 \text{ und } \texttt{dur}(f) \leq t_2 - t_1$$

Eine **Reise** ist eine Sequenz $j = j_1 j_2 \dots j_n$ von Fahrten und Fußwegen, so dass wie folgt gilt:

- j_1 und j_n sind Fahrten
- Sind j_i und j_{i+1} zwei Fahrten, so ist (j_i, j_{i+1}) ein Haltestellen-Anschluss
- Ist j_i ein Fußweg, so ist das Tupel $(j_{i-1} j_{i+1})$ ein Fußweg-Anschluss über j_{i+1}

Eine Reise ist also eine Aneinanderreihung von Fahrten mit Anschluss (oder ein direkter Fußmarsch) zwischen zwei Haltestellen s und t. Explizit sagen wir manchmal auch s-t-Reise. Reisen kann man nach verschiedensten Kriterien wie Ankunftszeit, Reisedauer oder Umstiege beurteilen [DDPW15]. Formal formulieren wir ein Kriterium durch eine Bewertungsfunktion κ, die jeder Reise eine Bewertung aus $\mathbb{N} \cup \{\infty\}$ zuordnet. Zu einer Reise $r = j_1 \dots j_n$ sei wie folgt definiert[2]: die Starthaltestelle $\texttt{stop}(r) = \texttt{stop}(j_1)$, die Zielhaltestelle $\texttt{dest}(r) = \texttt{stop}(j_n)$, die Abfahrtszeit $\texttt{time}(r) = \texttt{time}(j_1)$, die Ankunftszeit $\texttt{arrival}(r) = \texttt{time}(j_n)$, die Reisedauer $\texttt{dur}(r) = \texttt{arrival}(r) - \texttt{time}(r)$ und die Umstiege $\texttt{trips}(r) = n$.

Nach Delling führen wir nun den Begriff der Dominanz ein: Eine Reise r_1 $(\kappa_1, \dots, \kappa_k)$**-dominiert** eine Reise r_2, falls $\kappa_i(r_1) \leq \kappa_i(r_2)$ für alle $i \in \{1, \dots, k\}$ gilt. Für das Kriterium $\texttt{departure}(r) := \texttt{time}(r)$ invertieren wir die Ordnung, d.h. eine Reise r_1 $\texttt{departure}$-dominiert eine Reise r_2, falls $\texttt{departure}(r_1) \geq \texttt{departure}(r_2)$. Ein Menge R von Reisen heißt ($(\kappa_1, \dots, \kappa_k)$-)**Pareto-Menge**, falls sich die Elemente aus R paarweise nicht dominieren. Eine Reise r von s nach t nennen wir *tight*, falls es keine Reise von s nach t gibt, die r in den Kriterien $\texttt{departure}$ und $\texttt{arrival}$ dominiert [DDPW15].

Mit Hilfe dieses geschaffenen Begriff-Apparates lassen sich die üblichen Anfrageprobleme in Nahverkehrsnetzen formulieren:

Problem (EARLIESTARRIVAL-Problem)
Gegeben: Fahrplan $L = (S, E, T, F)$, Haltestellen $s, t \in S$ und ein Abreisezeitpunkt $\tau \in \mathbb{N}$
Ausgabe: Eine Reise r von s nach t mit $\texttt{departure}(r) \geq \tau$, die alle Reisen $\texttt{arrival}$-dominiert für die $\texttt{departure}(r) \geq \tau$ gilt

Das EARLIESTARRIVAL-Problem ist das Problem, zu einer gegebene Abfahrtszeit, eine schnellste Verbindung zwischen zwei Haltestelle zu finden. Dieses Problem lässt sich auf verschiedene Arten verallgemeinern, die wir hier auch betrachten wollen (nach [DDPW15]):

- PROFIL-Problem: Die Berechnung eines Fahrplan-Profiles der Verbindungen zwischen zwei Haltestellen. Das heißt die Menge alle schnellsten Verbindungen zwischen zwei Haltestellen, unabhängig von der Abfahrtszeit.

[2] Für den Fall, dass j_1, j_n keine Fahrten sind, muss dies entsprechend angepasst werden.

- PARETOARRIVAL-Problem: Für eine gegebene Abfahrtszeit eine Menge von Reisen zwischen den Haltestellen s und t berechnen, die die Ankunftszeit und die Anzahl der Umstiege minimiert.
- Verallgemeinerungen auf *Locations*: Anstelle der Angabe einer Ziel- und Start-Haltestelle, sind Mengen möglich. Dies könnte zum Beispiel von Interesse sein, wenn man die Verbindung von einer Adresse aus sucht, in dessen Nähe es mehrere Haltestellen gibt.

Problem (PROFIL-Problem)
Gegeben: Ein Fahrplan $L = (S, E, T, F)$, zwei Haltestellen $s, t \in S$
Ausgabe: Eine Pareto-Menge aller s-t-Reisen die *tight* sind

Problem (PARETOARRIVAL-Problem)
Gegeben: Fahrplan $L = (S, E, T, F)$, Haltestellen $s, t \in S$ und ein Abreisezeitpunkt $\tau \in \mathbb{N}$
Ausgabe: $(\texttt{arrival}, \texttt{trips})$-Pareto-Menge von Reisen mit Abreisezeitpunkt nicht vor τ

Das PARETOARRIVAL-Problem wird von Delling auch als (s, t, r)-MULTICRITERIAL-Problem bezeichnet, da hier zwei Parameter minimiert werden müssen. Dies ist im Allgemeinem **NP**-schwierig[3] [DDPW15]. Dellings Lösung für dieses Problem wird in Abschnitt 4.2 skizziert werden. Zunächst werden wir aber betrachten, wie die in Kapitel 3 beschrieben Methoden zur Lösung des EARLIESTARRIVAL-Problem und des PROFIL-Problem genutzt werden können.

4.1 Zeiterweiterte Graphen

Wie wir in Kapitel 3 gesehen haben, sind Labeling-Methoden sehr effizient für Pfad-Anfragen. Daher ist es naheliegend die Methoden des Labeling, auf Probleme in Nahverkehrsnetzen zu übertragen. Die genannten Probleme sind allerdings nicht direkt übertragbar. Dazu müssen die Fahrpläne zunächst in ein Graphmodell umgewandelt werden. Dies hat Delling in [DDPW15, S. 275] präsentiert. Delling beruft sich dabei primär auf die Arbeiten von Pyrga und anderen [PSWZ07]. Diese haben das (simple) zeiterweiterte Graphmodell für Fahrpläne entwickelt, dessen sich Delling bedient hat.

Das zeiterweiterte Graphmodell für Fahrpläne repräsentiert Ereignisse als Knoten und verbindet ein Ereignis an der selben Station immer mit dem zeitlich nachfolgenden Ereignis. Zusätzlich wird von jedem Knoten v aus eine Kante zu einem Knoten w einer anderen Haltestelle eingefügt, wenn w der frühste Knoten in dieser Station ist, mit dem v eine Verbindung hat. Für jeden Fußweg füge nach gleichem Schema, unter Beachtung der Zeit, Kanten ein. Die Kantengewichte ergeben sich aus den Zeitdifferenzen der Ereignisse. Die Berechnung ist in Algorithmus 4.1 zusammengefasst.

4.1.1 Berechnung der Anfragen

Mit dem Algorithmus 3.1 für Hublabeling-Anfragen können wir keines der Probleme EARLIESTARRIVAL, PROFIL, PARETOARRIVAL im zeiterweiterten Graphen lösen (nach Label Berechnung), denn die Knoten des zeiterweiterten Graphen sind Ereignisse. So das sich direkt nur Ereignis zu Ereignis-Anfragen beantworten lassen. Allerdings sind die zeiterweiterten Graphen so konstruiert, dass es reicht den Weg zum Ziel-Ereignis von dem frühsten passenden Ereignis einer

[3] Allerdings gibt es pseudo-polynomielle Algorithmen für in der Praxis relevante Instanzen [Hoc08].

Algorithmus 4.1 : FahrplanGraphConstruct

Eingabe : Ein Fahrplan $L = (S, E, T, F)$
Ausgabe : Graph (V, A, l)

1: Sei $\leq_t$ die partielle Ordnung der Ereignisse nach `time`
2: $A \leftarrow \emptyset$
3: $V \leftarrow E$ `// Knoten repräsentieren Ereignisse`
`// Zeitlich direkt aufeinanderfolgende Ereignise am selben Standort werden miteinander verbunden`
4: $A \leftarrow A \cup \{(e_1, e_2) \mid e_2 = \min_{\leq_t}\{e \mid \texttt{time}(e) > \texttt{time}(e_1) \text{ und } \texttt{stop}(e) = \texttt{stop}(e_1)\}\}$
`// Füge Kanten für Verbindungen ohne Zwischenhalte ein`
5: $A \leftarrow A \cup \{(e_1, e_2) \mid (e_1, e_2) \text{ ist eine Verbindung }\}$
`// Füge Kanten für die Fußwege zwischen den Stationen ein`
6: **foreach** $(s_1, n, s_2) \in F$ **do**
7: $\quad A \leftarrow A \cup \{(e_1, e_2) \mid \texttt{stop}(e_1) = s_1, \texttt{stop}(e_2) = s_2, e_2 = \min_{\leq_t}\{e \mid \texttt{time}(e) \geq \texttt{time}(e_1) + n\}\}$
`// Kantengewichte sind die zeitliche Abstände`
8: $l \leftarrow \{((e_1, e_2), \texttt{time}(e_1) - \texttt{time}(e_2)) \mid (e_1, e_2) \in A\}$
9: **return** (V, A, l)

Haltestelle aus zu suchen, da diese alle untereinander in zeitlich aufsteigender Reihenfolge Verbunden sind.

Ein Algorithmus, der das EARLIESTARRIVAL-Problem für Knoten s, t und Zeitpunkt τ löst, berechnet zunächst das frühste Event (nach τ) an Haltestelle s. Dann wird für jedes Ereignis an Haltestelle s nach τ ein HublabelQuery (Algorithmus 3.1) durchgeführt, um zu bestimmen ob es eine Verbindung gibt. Der Algorithmus bricht ab, wenn er eine Verbindung gefunden hat. Detailliert ist der EARLIESTARRIVAL-Query Algorithmus unter Algortihmus 4.2 zu finden.

Algorithmus 4.2 : EARLIESTARRIVAL-Query

Eingabe : Zeiterweiteter Graph $G = (V, A, l)$ zum Fahrplan $L = (S, E, T, F)$, Haltestellen s, t, Zeitpunkt τ und Labeling L_f, L_r für G.
Ausgabe : Abfahrtszeit a und Ankunftszeit b einer s-t-Reise r mit $\texttt{departure}(r) > \tau$

`// Finde das frühste Ereignis an Haltestelle` s `nach Zeitpunkt` τ
1: $a \leftarrow \text{argmin}_{e \in \{e \mid \texttt{stop}(e)=s \text{ und } \texttt{time}(e) \geq \tau\}} \texttt{time}(e)$
`// Finde das frühste Ereignis an der Haltestelle` t`, so dass es eine Verbindung von` a `aus gibt`
2: $b \leftarrow \text{argmin}_{e \in \{e \mid \texttt{stop}(e)=t\}} \text{HublabelingQuery}(G, a, e, L_f, L_r)$
3: **return** a, b

Das PROFIL-Problem für Haltestellen s, t lässt sich algorithmisch wie folgt lösen [DDPW15]: Liegen alle Ereignisse der Haltestellen s und t sortiert nach Zeitpunkt vor, so können alle Pfade durch einen „koordinierten“Lauf über die Ereignisse (dem Merge aus MergeSort ähnlich) berechnet werden. Für jedes Ereignis, das mit Haltestelle s korrespondiert, versucht der Algorithmus das erste mit t korrespondierende Ereignis zu finden, so dass es einen Weg zwischen beiden gibt. Da die Knoten nach Zeitpunkt sortiert sind, ist dieser auch der frühste, und somit auch der kürzeste Weg. Die Details sind Algorithmus 4.3 zu entnehmen.
Sowohl der EARLIESTARRIVAL-Query-Algorithmus, als auch der PROFIL-Query-Algorithmus, berechnet nicht die gesuchten Reisen, sondern die Länge der Reisen, bzw. Abfahrtszeiten und Ankunftszeiten. Generell ist es aber möglich bei der Vorberechnung des Labelings, neben der

Algorithmus 4.3 : PROFIL-Query

Eingabe : Zeiterweiteter Graph $G = (V, A, l)$ zum Fahrplan $L = (S, E, T, F)$, Haltestellen s, t und Labeling L_f, L_r für G.

Ausgabe : Menge von Tupeln $(a.b)$, Abfahrtszeit a und Ankunftszeit b, aller s-t-Reisen die *tight* sind.

```
1: P ← ∅ // Initialisierung
2: i, j ← 1
3: e_1 ... e_n ← sortierte Sequenz aller Knoten aus {e | stop(e) = s} nach time
4: f_1 ... f_m ← sortierte Sequenz aller Knoten aus {f | stop(f) = t} nach time
5: while i ≤ n und j ≤ m do
6:     if L_f(e_i) ∩ L_r(f_j) = ∅ then
7:         j ← j + 1
8:     else
           // Nun gilt L_f(e_i) ∩ L_r(f_j) ≠ ∅, also gibt es einen Weg.
           // Finde nun die kürzeste:
9:         while i + 1 ≤ n und L_f(e_{i+1}) ∩ L_r(f_j) ≠ ∅ do i ← i + 1
10:        P ← P ∪ {(time(e_i), time(f_j))}
11:        i ← i + 1
12: return P
```

Distanz, auch den kürzesten Pfad im Label zu speichern [ADF+12, Abs. 3.2], da dieser während der Vorberechnung implizit aufgebaut wird. So kann mit (theoretisch) gleicher Laufzeit auch der Pfad (die Reise), in der Anfrage, berechnet werden. Das komplette Vorgehen aus einem Fahrplan einen Graphen und dann ein Labeling zu berechnen wollen wir, nach Delling, PTL-Verfahren (engl. *public transit labeling*) nennen. Dies schließt Erweiterungen und Verbesserungen von Delling ein, die wir im folgenden Abschnitt skizzieren wollen.

4.1.2 Verbesserungen und Erweiterungen

Wir betrachten einige Verbesserungungs- und Erweiterungs-Möglichkeiten der Algorithmen aus Abschnitt 4.1.1, die Delling vorgeschlagen hat.

Location zu *Location*-Anfragen

In Realweltapplikationen von (Nahverkehrs)-Netzen gibt es oft die Möglichkeit Routen von oder zu Orten zu berechnen. Für diese Orte kommen dann oft mehrere Haltestellen in der näheren Umgebung in Frage. Die Algorithmen von Delling können so angepasst werden, dass sie solche Anfragen auch beantworten können. Angelehnt an Dellings Ausführungen [DDPW15, S. 280] könnte man dafür wie folgt vorgehen: Sei s^* der Startort und t^* der Zielort. Zunächst berechne $\mathcal{S}$, die Menge der naheliegenden Starthaltestellen, und $\mathcal{T}$, die Menge der möglichen Zielhaltestellen mit den jeweiligen Abständen. Um die Berechnung mit den Algorithmen aus Abschnitt 4.1.1 zu ermöglichen, werden virtuelle Knoten (Ereignisse) für $\mathcal{S}$ und $\mathcal{T}$ in den Graphen eingefügt. Diese werden „koordiniert“aus Ereignissen erzeugt, die zugehörig zu Haltestellen aus $\mathcal{S}$ sind. Ebenfalls werden neue Kanten eingefügt. Dabei müssen die Zeitpunkte der neuen Ereignisse immer um die zeitliche Differenz zu s^* angepasst werden. Die Label für diese Ereignisse lassen sich dann aus den Labels bisheriger Knoten errechnen, unter Berücksichtigung der zeitlichen Differenz zu s^*. Analog kann man für $\mathcal{T}$ verfahren.

Verkleinerung der zeiterweiterten Graphen

Nach Delling sind in zeiterweiterten Graphen nicht alle Pfade gleich interessant: Ereignisse, die reine Ankunfts-Ereignisse sind, benötigen keine L_f-Label, und Ereignisse, die reine Abreise-Ereignisse sind, benötigen keine L_r Label [DDPW15, S. 277]. Darüber hinaus ist es möglich den Graph noch weiter zu verkleinern [DDPW15, Kap. 4]: Zunächst werden Kanten zwischen einem Abfahrts-Ereignis und einem Ankunfts-Ereignis entfernt, falls es ein Abfahrts-Ereignis in der Zukunft gibt, das zum selben Ankunfts-Ereignis führt (die Erreichbarkeit bleibt so erhalten). Nun ist es mit den in Abschnitt 4.1.1 angegebenen Algorithmen nicht mehr sicher möglich Anfragen zu berechnen; man müsste alle nachfolgenden Knoten auch betrachten (was die Laufzeit erhöht). Zur Lösung dieses Problems schlägt Delling *Stop Labels* vor, bei denen alle Labels der Ereignisse einer Station verschmolzen werden. Nun kann mit anpassten Algorithmen die Anfrage wieder beantwortet werden. Auf die Details verweisen wir an [DDPW15, Kap. 4].

Pareto-Anfragen

Nach Delling ist es möglich das Graphmodell so zu modifizieren, dass es möglich ist, das ParetoArrival-Problem zu lösen [DDPW15, Kap. 5]. Dazu muss ein Algorithmus in der Lage sein, für eine Reise die Umstiege zu berechnen. Hierzu wird ein zweites Labeling berechnet: Im ersten Schritt wird der zeiterweiterte Graph so modifiziert, dass alle Kanten ein Gewicht von 0 erhalten. Nun werden in einer Station so Knoten und Kanten für Haltestellen-Anschlüsse hinzugefügt, das ein Umstieg zu einer Distanz von 1 führt. Die in diesem neuem Graph berechneten Labels enthalten nun die Anzahl der Umstiege auf einer Reise. Nun kann das ParetoArrival-Problem wie folgt gelöst werden: Einen Eintrag der Pareto-Menge erhalten wir durch die Ausführung einer EarliestArrival-Anfrage (auf den *normalen* Labels). Dieser liefert auch ein Ankunfts-Ereignis t. Wir überprüfen nun, mit den zeitlichen nach t liegenden Ereignissen und den *neuen* Labels, ob es eine Reise gibt die weniger Umstiege hat. Dies sind weitere Einträge in die Pareto-Menge.

4.2 Experimente

Da für den Algorithmus PTL keine theoretischen Garantien bekannt sind, gibt es nur experimentelle Ergebnisse als Indikator für die Qualität des Algorithmus. Delling hat Experimente für den PTL-Algorithmus zum Vergleich mit anderen Algorithmen veröffentlicht [DDPW15, Kap. 6]. Die experimentell bestimmten Anfragezeiten für die verschiedenen Probleme sind in Tabelle 4.1 zu finden. Betrachtet wurden neben PTL von Delling dabei folgende Techniken: Den RAPTOR-Algorithmus für Nahverkehrsnetze, der ohne Vorberechnungen auskommt, wobei (A)CSA eine modifizierte Varianten des RAPTOR-Algorithmus ist. Bei CH handelt es sich um einen Labeling-Algorithmus von Cohen (siehe auch Kapitel 3). TP ist eine Vorberechnungstechnik, die speziell für Nahverkehrsnetze entwickelt wurde, näheres siehe [BDG+15]. Weiterführende Literatur zu diesen Algorithmen ist bei Delling zu finden [DDPW15, S. 282].

Der Algorithmus PTL hat auf allen betrachten Datenquellen und Anfragetypen jeweils die geringste Laufzeit. Allerdings wurden nur die Algorithmen CSA, TP auf denselben Daten wie PTL ausgeführt. Als Beispiel sei genannt, dass der Algorithmus TP auf dem Madrid-Datensatz für ParetoArrival-Anfragen, $3.1ms$ benötigt, der Algorithmus PTL hingegen nur $0.0643ms$. Auch gegenüber dem Algorithmus CH sind die Antwortzeiten von PTL auf Datensätzen ähnlicher Größe (Europe und Switzerland) um den Faktor 10^2 kleiner. Gegenüber den Algorithmen (A)CSA und RAPTOR sind alle Algorithmen, die eine Vorberechnung vornehmen, schneller. Somit auch PTL.

	Instance				Criteria					
Algorithm	Name	Stops [$\cdot 10^3$]	Conns [$\cdot 10^6$]	Dy.	Arr.	Tran.	Prof.	Prep. [h]	Jn.	Query [ms]
CSA	London	20.8	4.9	1	●	○	○	—	n/a	1.8
ACSA	Germany	252.4	46.2	2	●	○	○	0.2	n/a	8.7
CH	Europe (LD)	30.5	1.7	p	●	○	○	<0.1	n/a	0.3
TP	Madrid	4.6	4.8	1	●	○	○	19	n/a	0.7
TP	Germany	248.4	13.9	1	●	○	○	249	0.9	0.2
PTL	London	20.8	5.1	1	●	○	○	0.9	0.9	0.0028
PTL	Madrid	4.7	4.5	1	●	○	○	0.4	0.9	0.0030
PTL	Sweden	51.1	12.7	2	●	○	○	0.5	1.0	0.0021
PTL	Switzerland	27.1	23.7	2	●	○	○	0.7	1.0	0.0021
RAPTOR	London	20.8	5.1	1	●	●	○	—	1.8	5.4
TP	Madrid	4.6	4.8	1	●	●	○	185	n/a	3.1
TP	Germany	248.4	13.9	1	●	●	○	372	1.9	0.3
PTL	London	20.8	5.1	1	●	●	○	49.3	1.8	0.0266
PTL	Madrid	4.7	4.5	1	●	●	○	10.9	1.9	0.0643
PTL	Sweden	51.1	12.7	2	●	●	○	36.2	1.7	0.0276
PTL	Switzerland	27.1	23.7	2	●	●	○	61.6	1.7	0.0217
CSA	London	20.8	4.9	1	●	○	●	—	98.2	161.0
ACSA	Germany	252.4	46.2	2	●	○	●	0.2	n/a	171.0
CH	Europe (LD)	30.5	1.7	p	●	○	●	<0.1	n/a	3.7
TP	Germany	248.4	13.9	1	●	○	●	249	16.4	3.3
PTL	London	20.8	5.1	1	●	○	●	0.9	81.0	0.0743
PTL	Madrid	4.7	4.5	1	●	○	●	0.4	110.7	0.1119
PTL	Sweden	51.1	12.7	2	●	○	●	0.5	12.7	0.0121
PTL	Switzerland	27.1	23.7	2	●	○	●	0.7	31.5	0.0245

Tabelle 4.1: Experimentelle Ergebnisse für den PTL-Algorithmus [DDPW15, Tab. 4].

5 Abschlussbemerkungen

In Kapitel 2 wurden die Grundlagen für kürzeste Pfade-Probleme rekapituliert. Dazu gehörten Algorithmen wie der Algorithmus von Dijkstra und das Einführen von *Shortest-Path Trees*. Die klassischen Techniken sind allerdings aufgrund ihres Speicherbedarfs und ihrer Zeitkomplexität für Anfragen in großen Graphen ungeeignet. Neue Techniken, die unter anderen Vorberechnungen nutzen, kommen eher in Frage. In Kapitel 3 wurde das Konzept der Labelings als Vorberechnungsmethode vorgestellt. Aus theoretischer Perspektive ist es **NP**-schwerig minimale Labelings zu berechnen. Daher wurden hierarchische Labelings eingeführt, die effizient berechenbare minimale Labelings vorweisen. Primär wurde auf die Arbeit von Delling zu dem Thema eingegangen, aber auch Referenzen zu anderen Labeling-Methoden gegeben. Dann wurde in Kapitel 4 die Anwendung der Labeling-Technik für Nahverkehrsnetze präsentiert. Dazu wurden diese erst modelliert, und dann in ein Graphmodell übertragen, so dass Labeling-Methoden anwendbar sind. Zuletzt wurden die experimentellen Ergebnisse von Delling präsentiert.

Insgesamt kann man Labeling-Techniken als erfolgreiches Konzept ansehen. Auf Graphen mit Hundertausenden von Knoten in kurzer Zeit exakt Anfragen beantworten zu können ist ein größer Vorteil dieser Methode. Auch für Nahverkehrsnetze scheint diese Technik geeignet zu sein. Da es Methoden gibt, die Fahrpläne in geeignete Graphen abbilden können.

Zur Labeling-Technik SamPG lässt sich folgendes ergänzen: Die experimentellen Ergebnisse die Delling vorgelegt hat, stützen nur Teilweise die Nützlichkeit des Algorithmus SamPG (bzw. RXL): Die durchschnittliche Größe der berechneten Labelings durch SamPG war gegenüber der durch PL Berechneten Labelings nicht immer überlegen (siehe Abschnitt 3.2), aber häufig genug um diese Methode zu nutzen. Die Kompressionsvarianten RXL und CRXL von Delling erzeugen erstaunlich kleine Labeling-Darstellungen (Abschnitt 3.3). Es wäre durchaus interessant diese für andere Label-Techniken zu nutzen.

Auch die Adaption der Labeling-Technik für das Setting der Nahverkehrsnetze kann man als Erfolg verbuchen. Der Algorithmus von Delling konnte auf großen Nahverkehrsnetzen Antwortzeiten für Profil-Anfragen von unter 30 μs erreichen. Man sollte anmerken, dass die experimentellen Ergebnisse nicht gut vergleichbar sind, da die verglichenen Algorithmen auf teilweise unterschiedlichen Datensätzen ausgeführt wurden. Darüber hinaus merkt Delling selbst an, dass es lohnenswert wäre andere Labeling-Techniken für Nahverkehrsnetze zu evaluieren und bestehende Methoden zu erweitern [DDPW15, S. 284].

Literaturverzeichnis

[ADF+12] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck, *HLDB: location-based services in databases*, SIGSPATIAL 2012, 2012, pp. 339–348.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Hierarchical hub labelings for shortest paths*, ESA 2012, 2012, pp. 24–35.

[AIKK14] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata, *Fast shortest-path distance queries on road networks by pruned highway labeling*, ALENEX 2014, 2014, pp. 147–154.

[AIY13] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida, *Fast exact shortest-path distance queries on large networks by pruned landmark labeling*, SIGMOD 2013, 2013, pp. 349–360.

[BDG+15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck, *Route planning in transportation networks*, CoRR **abs/1504.05140** (2015).

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick, *Reachability and distance queries via 2-hop labels*, SIAM J. Comput. **32** (2003), no. 5, 1338–1355.

[DDPW15] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck, *Public transit labeling*, SEA 2015, 2015, pp. 273–285.

[DGPW14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck, *Robust distance queries on massive networks*, ESA 2014, 2014, pp. 321–333.

[Die12] Reinhard Diestel, *Graph theory*, Graduate texts in mathematics, vol. 173, Springer, 2012.

[FT84] Michael L. Fredman and Robert E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, FoCS 1984, 1984, pp. 338–346.

[GRS13] Andrew V. Goldberg, Ilya Razenshteyn, and Ruslan Savchenko, *Separating hierarchical and general hub labelings*, MFCS 2013, 2013, pp. 469–479.

[Hoc08] Hartwig H. Hochmair, *Grouping of optimized pedestrian routes for multi-modal route planning: A comparison of two cities*, AGILE 11, 2008, pp. 339–358.

[PSWZ07] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis, *Efficient models for timetable information in public transportation systems*, ACM Journal of Experimental Algorithmics **12** (2007), 2.4:1–2.4:39.

[Tar82] Robert E. Tarjan, *Sensitivity analysis of minimum spanning trees and shortest path trees*, Inf. Process. Lett. **14** (1982), no. 1, 30–33.