

Algorithmen für Routenplanung

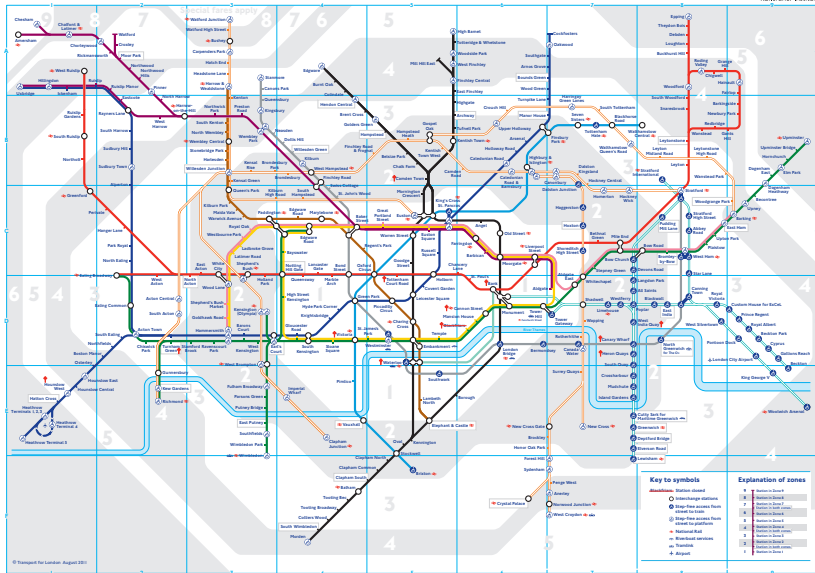
17. Vorlesung, Sommersemester 2019

Jonas Sauer | 8. Juli 2019

INSTITUT FÜR THEORETISCHE INFORMATIK · ALGORITHMIK · PROF. DR. DOROTHEA WAGNER



Fahrplanauskunft



Eingabe bei Straßennetzen: Straßenkarte bestehend aus

- Kreuzungen
- Straßensegmenten
- Verschiedenen Metriken (Reisezeit, Distanz, ...)

Eingabe bei Straßennetzen: Straßenkarte bestehend aus

- Kreuzungen
- Straßensegmenten
- Verschiedenen Metriken (Reisezeit, Distanz, ...)

Was ist die Eingabe bei ÖV-Netzwerken?

Fahrplan:

- Menge \mathcal{S} von **Stops** (Bahnhöfe, Bahnsteige, Haltestellen, ...),
- Menge \mathcal{T} von **Trips** (Züge, Busse, Trams, ...)
- Menge \mathcal{C} von **Connections** (elementaren Verbindungen)
- Zur Modellierung von Umstiegen:
 - Mindestumstiegszeiten an Stops: $\tau_{\text{ch}} : \mathcal{S} \rightarrow \mathbb{N}_0$
 - Fußwege zwischen (nahen) Stops: $\tau_{\text{walk}} : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{N}_0 \cup \{\infty\}$

Fahrplan:

- Menge \mathcal{S} von **Stops** (Bahnhöfe, Bahnsteige, Haltestellen, ...),
- Menge \mathcal{T} von **Trips** (Züge, Busse, Trams, ...)
- Menge \mathcal{C} von **Connections** (elementaren Verbindungen)
- Zur Modellierung von Umstiegen:
 - Mindestumstiegszeiten an Stops: $\tau_{ch} : \mathcal{S} \rightarrow \mathbb{N}_0$
 - Fußwege zwischen (nahen) Stops: $\tau_{walk} : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{N}_0 \cup \{\infty\}$

Connection: 5-Tupel $c = (v_{dep}(c), v_{arr}(c), \tau_{dep}(c), \tau_{arr}(c), trip(c))$

- Repräsentiert Fahrt des Trips $trip(c) \in \mathcal{T}$
- ... von Abfahrtsstop $v_{dep}(c) \in \mathcal{S}$
- ... mit Abfahrtszeit $\tau_{dep}(c) \in \Pi$
- ... zu Ankunftsstop $v_{arr}(c) \in \mathcal{S}$
- ... mit Ankunftszeit $\tau_{arr}(c) \in \Pi$
- ... ohne Zwischenhalt

Trip:

- Folge von Connections $T = (c_1, \dots, c_k)$
- Fahrt *eines* Zuges (Busses, ...)
- von Endstation zu Endstation
- Abfahrten an den Stops zu bestimmten Zeiten

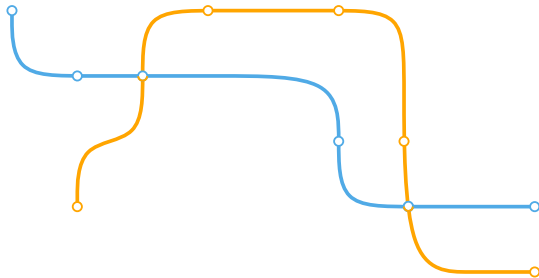
Trip:

- Folge von Connections $T = (c_1, \dots, c_k)$
- Fahrt *eines* Zuges (Busses, ...)
- von Endstation zu Endstation
- Abfahrten an den Stops zu bestimmten Zeiten

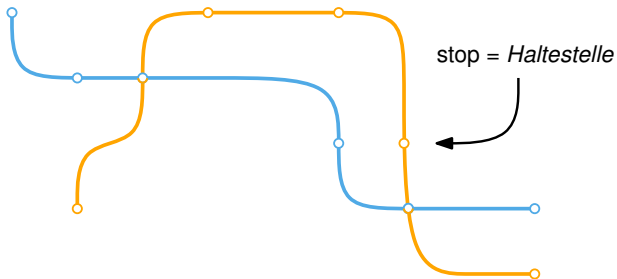
Routen:

- Partitionierung der Trips
- Route entspricht Zug-/Bahnlinie
- Zwei Trips T_1, T_2 gehören zur gleichen Route
 - ⇔ T_1 und T_2 besuchen genau die gleiche Sequenz von Stops
- Ist bei realen Zug-/Bahnlinien nicht immer der Fall!

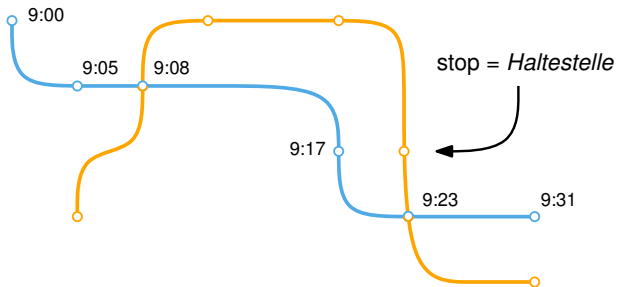
Fahrpläne – Beispiel



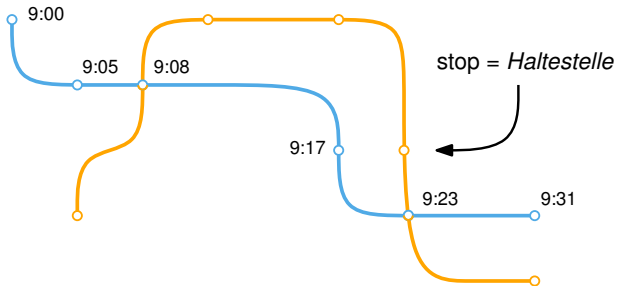
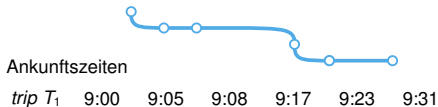
Fahrpläne – Beispiel



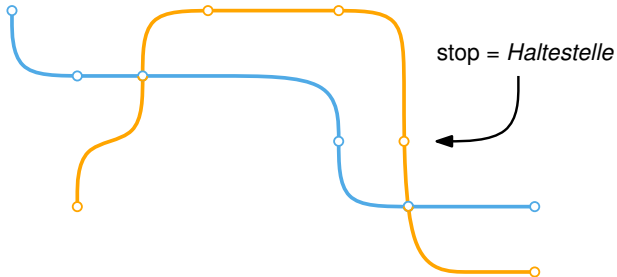
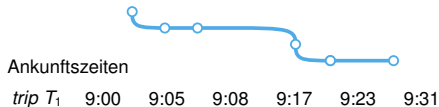
Fahrpläne – Beispiel



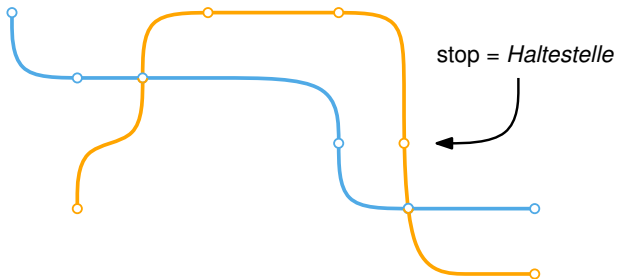
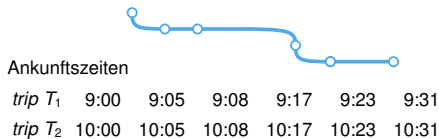
Fahrpläne – Beispiel




Fahrpläne – Beispiel



Fahrpläne – Beispiel

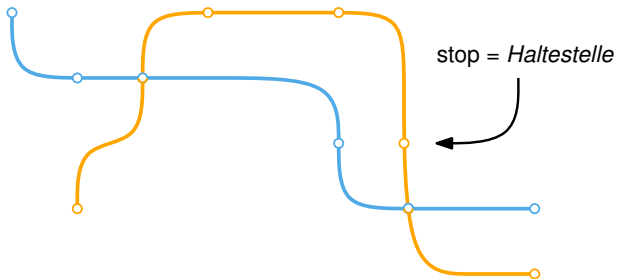


Fahrpläne – Beispiel



Ankunftszeiten

trip T_1	9:00	9:05	9:08	9:17	9:23	9:31
trip T_2	10:00	10:05	10:08	10:17	10:23	10:31
trip T_3	10:30	10:35	10:38	—	—	10:53



Fahrpläne – Beispiel

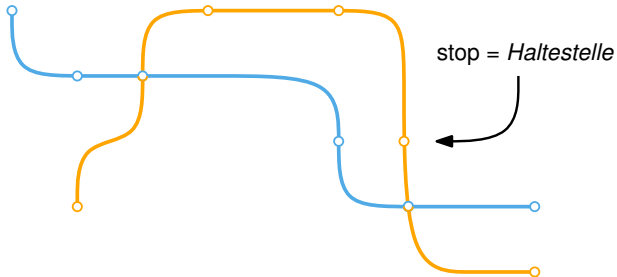


Ankunftszeiten

<i>trip</i> T_1	9:00	9:05	9:08	9:17	9:23	9:31
<i>trip</i> T_2	10:00	10:05	10:08	10:17	10:23	10:31
<i>trip</i> T_3	10:30	10:35	10:38	—	—	10:53

Route R_1

Route R_2



Gegenstück zu einem Pfad in einem Straßennetzwerk?

Gegenstück zu einem Pfad in einem Straßennetzwerk?

Journey:

- Modelliert Reise eines Passagiers durch das Netzwerk
- Folge von **Legs**: Teilsequenzen von Trips aus dem Fahrplan
- (Möglicherweise mit Transfers/Fußwegen dazwischen)

Gegenstück zu einem Pfad in einem Straßennetzwerk?

Journey:

- Modelliert Reise eines Passagiers durch das Netzwerk
- Folge von **Legs**: Teilsequenzen von Trips aus dem Fahrplan
- (Möglicherweise mit Transfers/Fußwegen dazwischen)

Journey ist konsistent, wenn ...

- Erster Stop von Leg T_{i+1} entspricht letztem Stop von Leg T_i
 - (Oder es existiert Fußweg dazwischen)
- Abfahrt von T_{i+1} ist *nach* Ankunft von T_i
 - (+ Zeit für Transfer/Fußweg)

Journeys – Beispiel

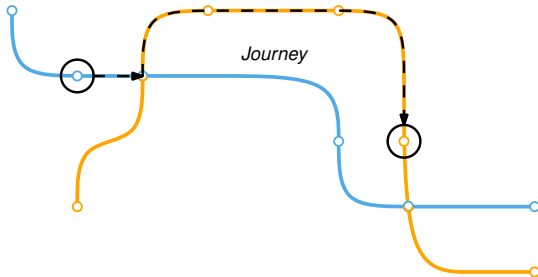


Ankunftszeiten

<i>trip</i> T_1	9:00	9:05	9:08	9:17	9:23	9:31
<i>trip</i> T_2	10:00	10:05	10:08	10:17	10:23	10:31
<i>trip</i> T_3	10:30	10:35	10:38	—	—	10:53

Route R_1

Route R_2



Zwei grundlegende Ansätze:

- 1 Modellierung als gerichteter Graph \rightsquigarrow Dijkstra
- 2 Benutze Fahrplan „direkt“ \rightsquigarrow Neue Algorithmen

Zwei grundlegende Ansätze:

- 1 Modellierung als gerichteter Graph \rightsquigarrow Dijkstra
- 2 Benutze Fahrplan „direkt“ \rightsquigarrow Neue Algorithmen

Jetzt ersteres, später zweiteres.

Zwei grundlegende Ansätze:

- 1 Modellierung als gerichteter Graph \rightsquigarrow Dijkstra
- 2 Benutze Fahrplan „direkt“ \rightsquigarrow Neue Algorithmen

Jetzt ersteres, später zweiteres.

Modellierung als Graph:

- Reduziere auf (bekanntes) Kürzeste-Wege-Problem
- Optimale Journeys entsprechen kürzesten Wegen
- Frage: Wie die Zeitabhängigkeit (Abfahrten/Ankünfte) kodieren?

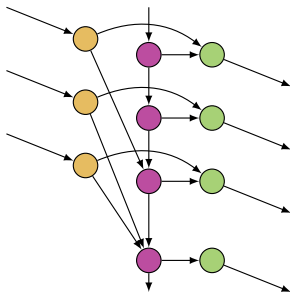
1. Zeitexpandiert

- Zeitabhängigkeiten ausrollen
 - Knoten entsprechen Ereignissen im Fahrplan
 - Kanten verbinden Ereignisse miteinander
 - Fahrt mit Connection
 - Umstieg
 - Warten
- Großer Graph
- + Einfacher Anfragealgorithmus (Dijkstra)

2. Zeitabhängig

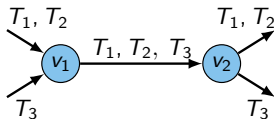
- Zeitabhängigkeit in den Kantengewichten
 - Knoten entsprechen Stops
 - Kante \Leftrightarrow Connection verbindet Stops
 - Umstiegszeiten?
- + Kleiner Graph
- Zeitabhängige KW-Algorithmen?

1. Zeitexpandiert



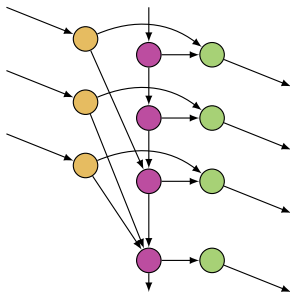
- Arrival-, Transfer- und Departure-Ereignisse
- Für jeden Trip
- Kantengewicht = Zeitdiff.
(alternativ: ungewichtet,
Knotenlabel = Ereigniszeit)

2. Zeitabhängig



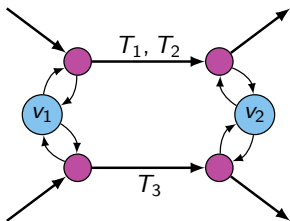
- Pro Stop: Stationsknoten
- Kanten: zeitabhängig
- Umstiege?

1. Zeitexpandiert



- **Arrival-**, **Transfer-** und **Departure-**Ereignisse
- Für jeden Trip
- Kantengewicht = Zeitdiff.
(alternativ: ungewichtet,
Knotenlabel = Ereigniszeit)

2. Zeitabhängig



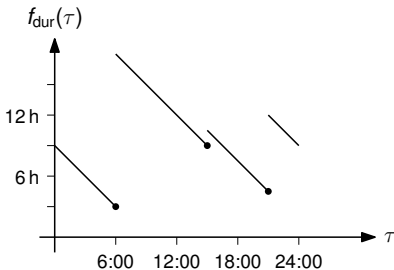
- Pro **Stop**: Stationsknoten
- Partitioniere Trips in Routen
- Pro Route: **Routen-**Knoten
- Routenkanten: zeitabhängig
- Stationskanten: Transferzeit

Connections modelliert durch **stückweise lineare Funktionen**

Connections zwischen v_i und v_j :

id	dep. time	travel time
1	06:00	3 h 00 min
2	15:00	9 h 00 min
3	21:00	4 h 30 min
\vdots	\vdots	\vdots

Entsprechende Funktion:



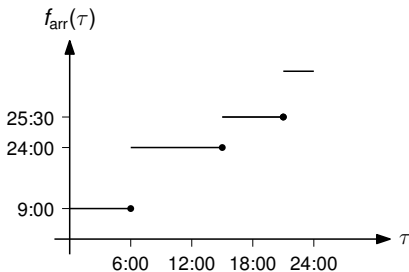
- Für jede Connection: **Connection Point** (τ, w)
 $\tau \hat{=}$ Abfahrtszeit, $w \hat{=}$ Reisezeit (bzw. Ankunftszeit)
- Zwischen Connections: Lineares Warten

Connections modelliert durch **stückweise lineare Funktionen**

Connections zwischen v_i und v_j :

id	dep. time	travel time
1	06:00	3 h 00 min
2	15:00	9 h 00 min
3	21:00	4 h 30 min
\vdots	\vdots	\vdots

Entsprechende Funktion:



- Für jede Connection: **Connection Point** (τ, w)
 $\tau \hat{=}$ Abfahrtszeit, $w \hat{=}$ Reisezeit (bzw. Ankunftszeit)
- Zwischen Connections: Lineares Warten

Definition

Sei $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ eine Reisezeit-Funktion. f erfüllt die **FIFO-Eigenschaft**, wenn für jedes $\varepsilon > 0$ und alle $\tau \in \mathbb{R}_0^+$ gilt, dass

$$f(\tau) \leq \varepsilon + f(\tau + \varepsilon).$$

Diskussion:

- Interpretation: “Warten lohnt sich nie”
 - Kürzeste Wege auf Graphen mit non-FIFO-Funktionen zu finden ist NP-schwer
(wenn Warten an Knoten nicht erlaubt ist)
- ⇒ Sicherstellen, dass Funktionen FIFO-Eigenschaft erfüllen

Definition

Sei $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ eine Ankunftszeit-Funktion. f erfüllt die **FIFO-Eigenschaft**, wenn für jedes $\varepsilon > 0$ und alle $\tau \in \mathbb{R}_0^+$ gilt, dass

$$f(\tau) \leq f(\tau + \varepsilon).$$

Diskussion:

- Interpretation: “Warten lohnt sich nie”
 - Kürzeste Wege auf Graphen mit non-FIFO-Funktionen zu finden ist NP-schwer
(wenn Warten an Knoten nicht erlaubt ist)
- ⇒ Sicherstellen, dass Funktionen FIFO-Eigenschaft erfüllen

Gegeben: Startstop s , Zielstop t

Gegeben: Startstop s , Zielstop t

Zeit-Anfrage (Earliest Arrival Query):

- Finde schnellste Journey für Abfahrtszeit τ_{dep}
- Analog zu Dijkstra?

Gegeben: Startstop s , Zielstop t

Zeit-Anfrage (Earliest Arrival Query):

- Finde schnellste Journey für Abfahrtszeit τ_{dep}
- Analog zu Dijkstra?

Profil-Anfrage (Profile Query):

- Finde schnellste Journey für jeden Abfahrtszeitpunkt $\tau \in \Pi$
- Analog zu Dijkstra?

Gegeben: Startstop s , Zielstop t und Abfahrtszeit τ_{dep}

Gegeben: Startstop s , Zielstop t und Abfahrtszeit τ_{dep}

1. Zeitexpandiert

Startknoten:

- Erstes Transferevent von s mit Zeit $\tau \geq \tau_{\text{dep}}$

Zielknoten:

- Im Voraus unbekannt!
- Stoppkriterium: Erster gesetzter Knoten an t induziert schnellste Verbindung zu t

2. Zeitabhängig

Startknoten:

- Stationsknoten s

Zielknoten:

- Stationsknoten t

Anfrage:

- Time-Dependent Dijkstra mit Zeit τ_{dep}
- Hier: Ankunftszeit im Voraus unbekannt

Algorithm 1: Time-Dijkstra($G = (V, E), s, \tau_{\text{dep}}$)

```
1  $d[s] = 0$ 
2  $Q.clear(), Q.insert(s, 0)$ 
3 while  $!Q.empty()$  do
4    $u \leftarrow Q.deleteMin()$ 
5   for all edges  $e = (u, v) \in E$  do
6     //  $\text{len}(e, \cdot) = f_{\text{dur}}^e(\cdot)$ 
7     if  $d[u] + \text{len}(e, \tau_{\text{dep}} + d[u]) < d[v]$  then
8        $d[v] \leftarrow d[u] + \text{len}(e, \tau_{\text{dep}} + d[u])$ 
9        $p[v] \leftarrow u$ 
10      if  $v \in Q$  then  $Q.decreaseKey(v, d[v])$ 
11      else  $Q.insert(v, d[v])$ 
```

Algorithm 2: Time-Dijkstra($G = (V, E), s, \tau_{\text{dep}}$)

```
1  $d[s] = \tau_{\text{dep}}$ 
2  $Q.\text{clear}(), Q.\text{insert}(s, 0)$ 
3 while  $!Q.\text{empty}()$  do
4    $u \leftarrow Q.\text{deleteMin}()$ 
5   for all edges  $e = (u, v) \in E$  do
6     //  $\text{len}(e, \cdot) = f_{\text{arr}}^e(\cdot)$ 
7     if  $\text{len}(e, d[u]) < d[v]$  then
8        $d[v] \leftarrow \text{len}(e, d[u])$ 
9        $p[v] \leftarrow u$ 
10      if  $v \in Q$  then  $Q.\text{decreaseKey}(v, d[v])$ 
11      else  $Q.\text{insert}(v, d[v])$ 
```

Beobachtung:

- Nur ein Unterschied zu Dijkstra
- Auswertung der Kanten

Beobachtung:

- Nur ein Unterschied zu Dijkstra
- Auswertung der Kanten

Non-FIFO-Netzwerke:

- Im Kreis fahren kann sich lohnen
- NP-schwer (wenn Warten an Knoten nicht erlaubt ist)
- Transportnetzwerke sind FIFO-modellierbar

Beobachtung:

- Nur ein Unterschied zu Dijkstra
- Auswertung der Kanten

Non-FIFO-Netzwerke:

- Im Kreis fahren kann sich lohnen
- NP-schwer (wenn Warten an Knoten nicht erlaubt ist)
- Transportnetzwerke sind FIFO-modellierbar

In unserem Szenario:

- Sicherstellen, dass jede Route FIFO-Eigenschaft erfüllt
- Für alle Trips T_i, T_j der Route muss gelten:
- T_i fährt an *jedem* Stop jeweils vor T_j ab (oder andersherum)

Gegeben: Startstop s , Zielstop t

Gegeben: Startstop s , Zielstop t

1. Zeitexpandiert

?

(Geht, aber nicht Teil der VL)

2. Zeitabhängig

Startknoten:

- Stationsknoten s

Zielknoten:

- Stationsknoten t

Anfrage:

- Label-Correcting-Algorithmus von s aus

Algorithm 3: Profile-Search($G = (V, E), s$)

```
1  $d_*[s] = 0$ 
2  $Q.clear(), Q.insert(s, 0)$ 
3 while ! $Q.empty()$  do
4    $u \leftarrow Q.deleteMin()$ 
5   for all edges  $e = (u, v) \in E$  do
6     if  $d_*[u] \oplus \text{len}(e) \not\leq d_*[v]$  then
7        $d_*[v] \leftarrow \min(d_*[u] \oplus \text{len}(e), d_*[v])$ 
8       if  $v \in Q$  then  $Q.decreaseKey(v, \underline{d}[v])$ 
9       else  $Q.insert(v, \underline{d}[v])$ 
```

Beobachtungen:

- Operationen auf Funktionen
- Knotenlabel: Funktion
- Knotenlabel nicht skalar \Rightarrow keine Totalordnung der Knotenlabel
- Wonach Priority Queue ordnen?
- Priorität im Prinzip frei wählbar
($d[u]$ ist das Minimum der Funktion $d_*[u]$)
- Knoten können mehrfach besucht werden \Rightarrow label-correcting

Funktion gegeben durch:

- Menge von Interpolationspunkten
- $I^f := \{(t_1^f, w_1^f), \dots, (t_k^f, w_k^f)\}$

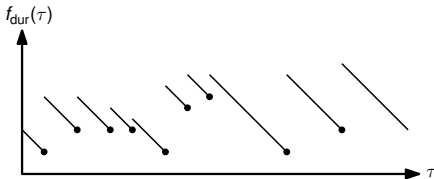
3 Operationen notwendig:

- Auswertung
- Linken \oplus
- **Minimumsbildung**

Auswertung von $f(\tau)$:

- Suche Punkt (t_i, w_i) mit $t_i \geq \tau$ und t_i minimal
- Dann Evaluation durch

$$f(\tau) = w_i + (t_i - \tau)$$



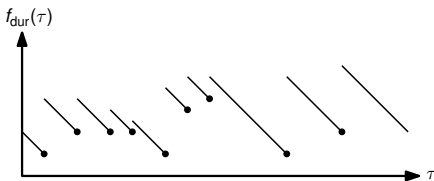
Auswertung von $f(\tau)$:

- Suche Punkt (t_i, w_i) mit $t_i \geq \tau$ und t_i minimal
- Dann Evaluation durch

$$f(\tau) = w_i + (t_i - \tau)$$

Problem:

- Finden von t_i und t_{i-1}
- Theoretisch:
 - Lineare Suche: $\mathcal{O}(|I|)$
 - Binäre Suche: $\mathcal{O}(\log_2 |I|)$



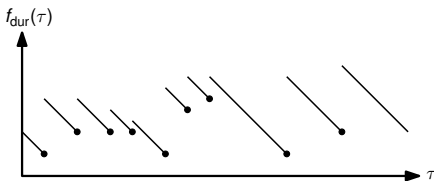
Auswertung von $f(\tau)$:

- Suche Punkt (t_i, w_i) mit $t_i \geq \tau$ und t_i minimal
- Dann Evaluation durch

$$f(\tau) = w_i + (t_i - \tau)$$

Problem:

- Finden von t_i und t_{i-1}
- Theoretisch:
 - Lineare Suche: $\mathcal{O}(|I|)$
 - Binäre Suche: $\mathcal{O}(\log_2 |I|)$
- praktisch:
 - $|I| < 30$: Lineare Suche
 - Sonst: Lineare Suche mit Startpunkt $\frac{\tau}{\bar{t}} \cdot |I|$



Definition

Seien $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ und $g: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ zwei Reisezeit-Funktionen, die die FIFO-Eigenschaft erfüllen. Die Linkoperation $f \oplus g$ ist dann definiert durch

$$f \oplus g := f + g \circ (\text{id} + f)$$

Oder

$$(f \oplus g)(\tau) := f(\tau) + g(\tau + f(\tau))$$

Definition

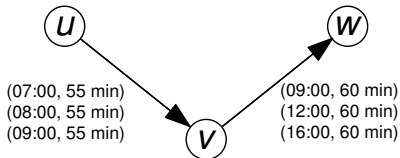
Seien $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ und $g: \mathbb{R}_0^+$ zwei Ankunftszeit-Funktionen, die die FIFO-Eigenschaft erfüllen. Die Linkoperation $f \oplus g$ ist dann definiert durch

$$f \oplus g := g \circ f$$

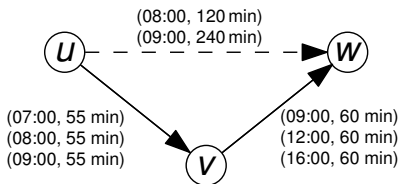
Oder

$$(f \oplus g)(\tau) := g(f(\tau))$$

Linken zweier Funktionen f und g :

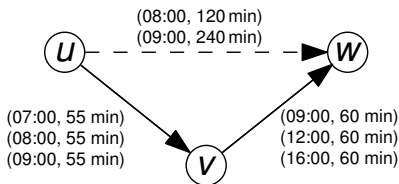


Linken zweier Funktionen f und g :



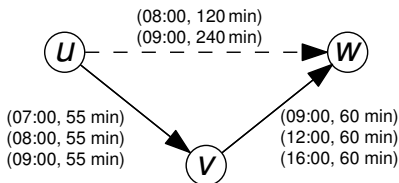
Linken zweier Funktionen f und g :

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - (t_i^f + w_i^f) \geq 0$ minimal
= Erste Verbindung, die man auf g erreichen kann



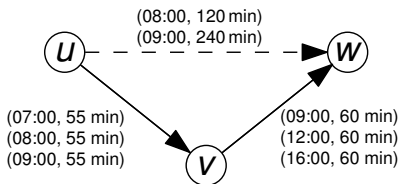
Linken zweier Funktionen f und g :

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - (t_i^f + w_i^f) \geq 0$ minimal
= Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g - t_i^f + w_j^g)$ hinzu



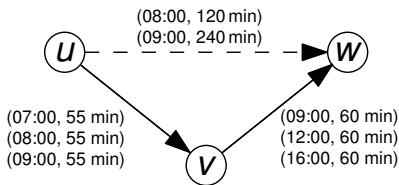
Linken zweier Funktionen f und g :

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - (t_i^f + w_i^f) \geq 0$ minimal
= Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g - t_i^f + w_j^g)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f



Linken zweier Funktionen f und g :

- Für jeden Punkt (t_i^f, w_i^f) bestimme den Verbindungspunkt (t_j^g, w_j^g) mit $t_j^g - (t_i^f + w_i^f) \geq 0$ minimal
= Erste Verbindung, die man auf g erreichen kann
- Füge $(t_i^f, t_j^g - t_i^f + w_j^g)$ hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben, behalte nur den mit größerem t_i^f
- Sweep-Algorithmus



Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig: $\mathcal{O}(1)$

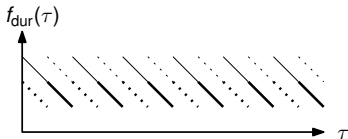
Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig: $\mathcal{O}(1)$

Speicherverbrauch:

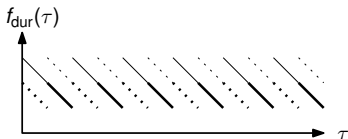
- Geklinkte Funktion hat $\min\{|I^f|, |I^g|\}$ Interpolationspunkte

Minimum zweier Funktionen f und g :



Minimum zweier Funktionen f und g :

- Für alle (t_i^f, w_i^f) : Behalte Punkt, wenn $w_i^f < g(t_i^f)$
- Für alle (t_j^g, w_j^g) : Behalte Punkt, wenn $w_j^g < f(t_j^g)$
- Keine Schnittpunkte möglich!

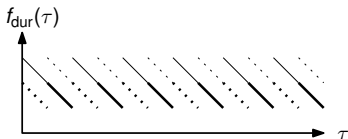


Minimum zweier Funktionen f und g :

- Für alle (t_i^f, w_i^f) : Behalte Punkt, wenn $w_i^f < g(t_i^f)$
- Für alle (t_j^g, w_j^g) : Behalte Punkt, wenn $w_j^g < f(t_j^g)$
- Keine Schnittpunkte möglich!

Vorgehen:

- Linearer Sweep



Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig: $\mathcal{O}(1)$

Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$
- Zum Vergleich: Zeitunabhängig: $\mathcal{O}(1)$

Speicherverbrauch:

- Keine Schnittpunkte
- ⇒ Minimum-Funktion kann maximal $|I^f| + |I^g|$ Interpolationspunkte enthalten

Profil-Anfragen

Gegeben:

Zeitabhängiges Netzwerk $G = (V, E)$ und Startstop s .

Gegeben:

Zeitabhängiges Netzwerk $G = (V, E)$ und Startstop s .

Problem (Profil-Anfrage):

Berechne die **Reisezeitfunktion** $\text{dist}_s(v, \tau)$, sodass für **alle** $\tau \in \Pi$ und $v \in V$ gilt: $\text{dist}_s(v, \tau)$ ist die Länge des **kürzesten Weges** von s nach v in G zur Abfahrtszeit τ an s .

Gegeben:

Zeitabhängiges Netzwerk $G = (V, E)$ und Startstop s .

Problem (Profil-Anfrage):

Berechne die **Reisezeitfunktion** $\text{dist}_s(v, \tau)$, sodass für alle $\tau \in \Pi$ und $v \in V$ gilt: $\text{dist}_s(v, \tau)$ ist die Länge des **kürzesten Weges** von s nach v in G zur Abfahrtszeit τ an s .

Bisheriger Ansatz:

Erweitere Dijkstras Algorithmus zu **Label-Correcting-Algorithmus**

- Benutze Funktionen statt Konstanten
- Verliert **Label-Setting**-Eigenschaft von Dijkstra
- **Deutlich langsamer** als Dijkstra (\approx Faktor 50)

Hauptidee

Beobachtung: Jede Journey ab s (irgendwohin) beginnt mit einer Connection an s .

Hauptidee

Beobachtung: Jede Journey ab s (irgendwohin) beginnt mit einer Connection an s .

Naiver Ansatz

Für jede ausgehende Connection c_i an s : Separate Zeitanfrage mit Abfahrtszeit $\tau_{\text{dep}}(c_i)$.

Beobachtung: Jede Journey ab s (irgendwohin) beginnt mit einer Connection an s .

Naiver Ansatz

Für jede ausgehende Connection c_i an s : Separate Zeitanfrage mit Abfahrtszeit $\tau_{\text{dep}}(c_i)$.

Nachteile:

- Zu viele redundante Berechnungen
- Nicht jede Connection ab s trägt zu $\text{dist}_s(v, \cdot)$ bei
Langsame Züge für weite Reisen machen wenig Sinn

0	1	2	3	4	5	6	7	8	9	10	11
6:30	7:04	9:26	10:34	11:08	12:42	13:01	13:58	16:46	18:24	19:20	21:08
8:30	8:30	14:28	14:28	14:28	14:28	16:46	16:46	23:30	23:30	23:30	23:30

Beobachtung: Jede Journey ab s (irgendwohin) beginnt mit einer Connection an s .

Naiver Ansatz

Für jede ausgehende Connection c_i an s : Separate Zeitanfrage mit Abfahrtszeit $\tau_{\text{dep}}(c_i)$.

Nachteile:

- Zu viele redundante Berechnungen
- Nicht jede Connection ab s trägt zu $\text{dist}_s(v, \cdot)$ bei
Langsame Züge für weite Reisen machen wenig Sinn

0	1	2	3	4	5	6	7	8	9	10	11
6:30	7:04	9:26	10:34	11:08	12:42	13:01	13:58	16:46	18:24	19:20	21:08
8:30	8:30	14:28	14:28	14:28	14:28	16:46	16:46	23:30	23:30	23:30	23:30

Beobachtung: Jede Journey ab s (irgendwohin) beginnt mit einer Connection an s .

Naiver Ansatz

Für jede ausgehende Connection c_i an s : Separate Zeitanfrage mit Abfahrtszeit $\tau_{\text{dep}}(c_i)$.

Nachteile:

- Zu viele redundante Berechnungen
- Nicht jede Connection ab s trägt zu $\text{dist}_s(v, \cdot)$ bei
Langsame Züge für weite Reisen machen wenig Sinn

0	1	2	3	4	5	6	7	8	9	10	11
6:30 8:30	7:04 8:30	9:26 14:28	10:34 14:28	11:08 14:28	12:42 14:28	13:01 16:46	13:58 16:46	16:46 23:30	18:24 23:30	19:20 23:30	21:08 23:30

Beobachtung: Jede Journey ab s (irgendwohin) beginnt mit einer Connection an s .

Naiver Ansatz

Für jede ausgehende Connection c_i an s : Separate Zeitanfrage mit Abfahrtszeit $\tau_{\text{dep}}(c_i)$.

Nachteile:

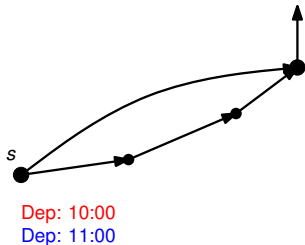
- Zu viele redundante Berechnungen
- Nicht jede Connection ab s trägt zu $\text{dist}_s(v, \cdot)$ bei
Langsame Züge für weite Reisen machen wenig Sinn

0	1	2	3	4	5	6	7	8	9	10	11
6:30 8:30	7:04 8:30	9:26 14:28	10:34 14:28	11:08 14:28	12:42 14:28	13:01 16:46	13:58 16:46	16:46 23:30	18:24 23:30	19:20 23:30	21:08 23:30

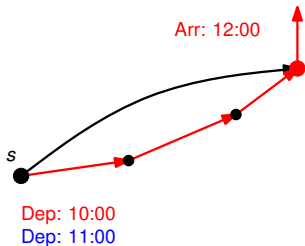
("Connection reduction")

Self-Pruning

Beobachtung:
Connections können sich **dominieren**.

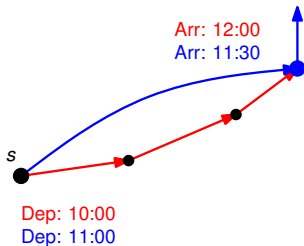


Beobachtung:
Connections können sich **dominieren**.



Self-Pruning

Beobachtung:
Connections können sich **dominieren**.

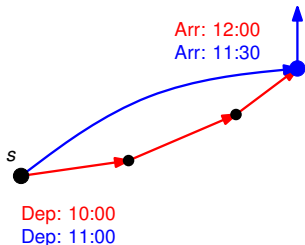


Beobachtung:

Connections können sich dominieren.

Self-Pruning (SP):

- 1 Benutze eine gemeinsame Queue
- 2 Keys sind Ankunftszeiten
- 3 Sortiere Connections c_i an s nach Abfahrtszeit



Beim Settlen von Knoten v und Conn.-Index i :

Falls v bereits gesettled mit Connection $j > i$, **prune** i an v

Integration von Self-Pruning:

- Verwalte Label $\text{maxconn}(v)$ an jedem Knoten v
Gibt maximale Connection an, mit der v gesettled wurde
- Update $\text{maxconn}(v)$ beim Settlen von v

Integration von Self-Pruning:

- Verwalte Label $\text{maxconn}(v)$ an jedem Knoten v
Gibt maximale Connection an, mit der v gesettled wurde
- Update $\text{maxconn}(v)$ beim Settlen von v

Beim Settlen von Knoten v und Conn.-Index i :
Falls v bereits gesettled mit Connection $j > i$, **prune** i an v

Integration von Self-Pruning:

- Verwalte Label $\text{maxconn}(v)$ an jedem Knoten v
Gibt maximale Connection an, mit der v gesettled wurde
- Update $\text{maxconn}(v)$ beim Settlen von v

Beim Settlen von Knoten v und Conn.-Index i :
Falls $\text{maxconn}(v) > i$, **prune** i an v

Integration von Self-Pruning:

- Verwalte Label $\text{maxconn}(v)$ an jedem Knoten v
Gibt maximale Connection an, mit der v gesettled wurde
- Update $\text{maxconn}(v)$ beim Settlen von v

Beim Settlen von Knoten v und Conn.-Index i :
Falls $\text{maxconn}(v) > i$, **prune** i an v

Wiederherstellung von Dijkstras Label-Setting-Eigenschaft pro Connection

Integration von Self-Pruning:

- Verwalte Label $\text{maxconn}(v)$ an jedem Knoten v
Gibt maximale Connection an, mit der v gesettled wurde
- Update $\text{maxconn}(v)$ beim Settlen von v

Beim Settlen von Knoten v und Conn.-Index i :
Falls $\text{maxconn}(v) > i$, **prune** i an v

Wiederherstellung von Dijkstras Label-Setting-Eigenschaft pro Connection

⇒ Self-Pruning Connection-Setting Algorithmus (SPCS)

Stoppkriterium

Dijkstras Algorithmus:

Breche die Suche ab, sobald t abgearbeitet wurde.

Dijkstras Algorithmus:

Breche die Suche ab, sobald t abgearbeitet wurde.

kann adaptiert werden durch

Self-Pruning Connection-Setting:

- Verwalte Label $T_m := -\infty$

Dijkstras Algorithmus:

Breche die Suche ab, sobald t abgearbeitet wurde.

kann adaptiert werden durch

Self-Pruning Connection-Setting:

- Verwalte Label $T_m := -\infty$
- Wenn Connection i an t abgearbeitet wird, setze
 $T_m := \max\{T_m, i\}$

Dijkstras Algorithmus:

Breche die Suche ab, sobald t abgearbeitet wurde.

kann adaptiert werden durch

Self-Pruning Connection-Setting:

- Verwalte Label $T_m := -\infty$
- Wenn Connection i an t abgearbeitet wird, setze $T_m := \max\{T_m, i\}$
- Prune alle Connections $j < T_m$ (an jedem Knoten)

Dijkstras Algorithmus:

Breche die Suche ab, sobald t abgearbeitet wurde.

kann adaptiert werden durch

Self-Pruning Connection-Setting:

- Verwalte Label $T_m := -\infty$
- Wenn Connection i an t abgearbeitet wird, setze $T_m := \max\{T_m, i\}$
- Prune alle Connections $j < T_m$ (an jedem Knoten)
- Halte an, wenn Priority-Queue leer läuft

Parallelisierung: Idee

Gegeben:

Shared-Memory-Parallelisierung mit p Cores

Parallelisierung: Idee

Gegeben:

Shared-Memory-Parallelisierung mit p Cores

Idee:

Verteile Connections c_i von s auf verschiedene Threads

0 6:30	1 7:04	2 9:26	3 10:34	4 11:08	5 12:42	6 13:01	7 13:58	8 16:46	9 18:24	10 19:20	11 21:08
Thread 0			Thread 1			Thread 2			Thread 3		

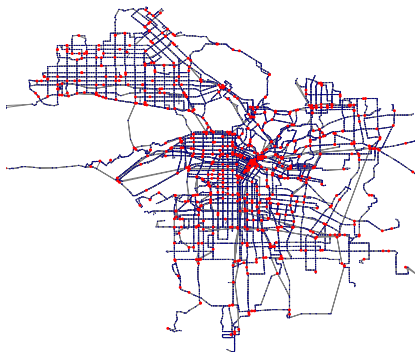
- Jeder Thread führt SPCS auf seiner **Teilmenge** der Verbindungen aus
- **Ergebnisse** werden im Anschluss zu $\text{dist}_s(v, \cdot)$ zusammengeführt
- Führe **Connection Reduction** auf gemergtem Ergebnis durch

Netzwerk von **Los Angeles**:

- 15 581 Stops,
- 1 046 580 Connections

Zugnetz von **Europa**:

- 30 517 Stops,
- 1 775 533 Connections

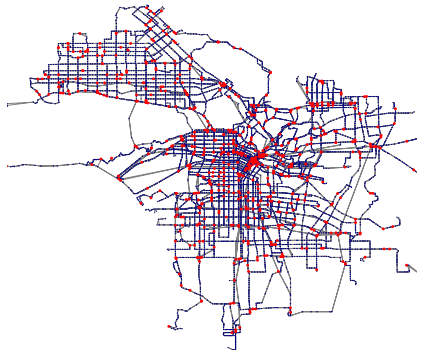


Netzwerk von **Los Angeles**:

- 15 581 Stops,
- 1 046 580 Connections

Zugnetz von **Europa**:

- 30 517 Stops,
- 1 775 533 Connections



Auswertung durch 1 000 Anfragen, wobei Start- und Zielstops zufällig gleichverteilt gewählt werden.

	Los Angeles					Europe			
	Settled p Conns	Time [ms]	Spd Up	Std- Dev		Settled Conns	Time [ms]	Spd Up	Std- Dev
PSPCS	1	2.5 M	1209.0	1.0	—	3.3 M	2152.0	1.0	—
	2	2.5 M	690.0	1.8	14.7 %	3.1 M	1054.2	2.0	16.1 %
	4	2.5 M	417.4	2.9	18.2 %	3.4 M	673.8	3.2	24.4 %
	8	2.5 M	267.7	4.5	20.0 %	4.2 M	510.9	4.2	23.8 %
LC	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

	Los Angeles					Europe			
	p	Settled Conns	Time [ms]	Spd Up	Std- Dev	Settled Conns	Time [ms]	Spd Up	Std- Dev
PSPCS	1	2.5 M	1209.0	1.0	—	3.3 M	2152.0	1.0	—
	2	2.5 M	690.0	1.8	14.7 %	3.1 M	1054.2	2.0	16.1 %
	4	2.5 M	417.4	2.9	18.2 %	3.4 M	673.8	3.2	24.4 %
	8	2.5 M	267.7	4.5	20.0 %	4.2 M	510.9	4.2	23.8 %
LC	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

- PSPCS betrachtet deutlich weniger Connections als LC

	Los Angeles					Europe			
	Settled p Conns	Time [ms]	Spd Up	Std- Dev		Settled Conns	Time [ms]	Spd Up	Std- Dev
PSPCS	1	2.5 M	1209.0	1.0	—	3.3 M	2152.0	1.0	—
	2	2.5 M	690.0	1.8	14.7 %	3.1 M	1054.2	2.0	16.1 %
	4	2.5 M	417.4	2.9	18.2 %	3.4 M	673.8	3.2	24.4 %
	8	2.5 M	267.7	4.5	20.0 %	4.2 M	510.9	4.2	23.8 %
LC	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

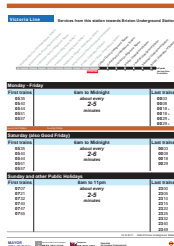
- PSPCS betrachtet deutlich weniger Connections als LC
- PSPCS skaliert sehr gut mit zunehmender Anzahl Cores

	Los Angeles					Europe			
	Settled p Conns	Time [ms]	Spd Up	Std- Dev		Settled Conns	Time [ms]	Spd Up	Std- Dev
PSPCS	1	2.5 M	1209.0	1.0	—	3.3 M	2152.0	1.0	—
	2	2.5 M	690.0	1.8	14.7 %	3.1 M	1054.2	2.0	16.1 %
	4	2.5 M	417.4	2.9	18.2 %	3.4 M	673.8	3.2	24.4 %
	8	2.5 M	267.7	4.5	20.0 %	4.2 M	510.9	4.2	23.8 %
LC	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

- PSPCS betrachtet deutlich weniger Connections als LC
- PSPCS skaliert sehr gut mit zunehmender Anzahl Cores

Erinnerung: Ein Fahrplan besteht aus

- Stops (Bahnhöfe, Bahnsteige, ...),
- Routen (Bus-, U-Bahn-Linien, ...),
- Trips mit Abfahrt-/Ankunftszeiten,
- und Fußwegen zum Umsteigen.



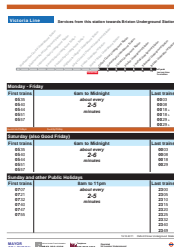
Erinnerung: Ein Fahrplan besteht aus

- Stops (Bahnhöfe, Bahnsteige, ...),
- Routen (Bus-, U-Bahn-Linien, ...),
- Trips mit Abfahrt-/Ankunftszeiten,
- und Fußwegen zum Umsteigen.

Earliest Arrival Problem:

Gegeben Stops s , t und Abfahrtszeit τ , berechne

- Route zu t , die an s nicht früher als τ abfährt
- und an t frühestmöglich ankommt.



Stuttgart Line Services from this station towards Stuttgart-Unterrang Station

Monday - Friday	Weekends	Weekends
First trains	Services to Stuttgart	Last trains
05:00	about every 2-5 minutes	08:00
05:05		08:10
05:10		08:15
05:15		08:20
05:20		08:25

First trains	Services to Stuttgart	Last trains
05:25	about every 2-5 minutes	08:30
05:30		08:35
05:35		08:40
05:40		08:45
05:45		08:50

First trains	Services to Stuttgart	Last trains
07:00	about every 2-5 minutes	20:00
07:05		20:10
07:10		20:15
07:15		20:20
07:20		20:25

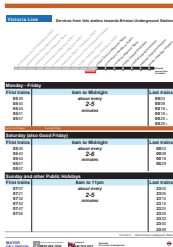
Erinnerung: Ein Fahrplan besteht aus

- Stops (Bahnhöfe, Bahnsteige, ...),
- Routen (Bus-, U-Bahn-Linien, ...),
- Trips mit Abfahrt-/Ankunftszeiten,
- und Fußwegen zum Umsteigen.

Earliest Arrival Problem:

Gegeben Stops s , t und Abfahrtszeit τ , berechne

- Route zu t , die an s nicht früher als τ abfährt
- und an t frühestmöglich ankommt.



S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar

Monday - Friday
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar

Line	Direction	Frequency	Start	End
S-Bahn Rhein-Neckar	to Mannheim	about every 2-5 minutes	08:10	08:15
S-Bahn Rhein-Neckar	to Heidelberg	about every 2-5 minutes	08:10	08:15

Saturday (also Good Friday)
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar

Line	Direction	Frequency	Start	End
S-Bahn Rhein-Neckar	to Mannheim	about every 2-5 minutes	08:10	08:15
S-Bahn Rhein-Neckar	to Heidelberg	about every 2-5 minutes	08:10	08:15

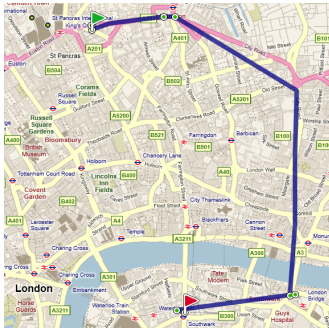
Sunday and other Public holidays
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar
S-Bahn Rhein-Neckar

Line	Direction	Frequency	Start	End
S-Bahn Rhein-Neckar	to Mannheim	about every 2-5 minutes	08:10	08:15
S-Bahn Rhein-Neckar	to Heidelberg	about every 2-5 minutes	08:10	08:15

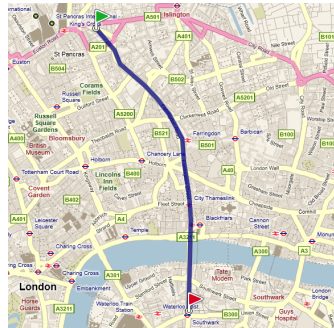
Reicht uns das?

Einbeziehen von Umstiegen

Umstiege zu betrachten ist wichtig!



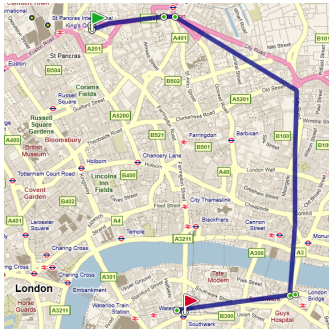
Ankunft 11:08 Uhr, 2 Umstiege



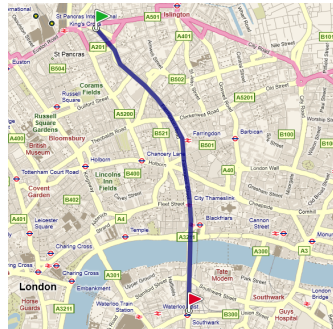
Ankunft 11:09 Uhr, 0 Umstiege

Einbeziehen von Umstiegen

Umstiege zu betrachten ist wichtig!



Ankunft 11:08 Uhr, 2 Umstiege



Ankunft 11:09 Uhr, 0 Umstiege

Idee: Berechne „gute“ Routen für Ankunftszeit *und* Anzahl Umstiege.

Definition (Pareto-Optimum)

Zu einer Menge M von n -Tupeln heißt ein Tupel $m_i = (x_1, \dots, x_n) \in M$ **Pareto-Optimum**, wenn es kein anderes $m_j \in M$ gibt, so dass m_j in *allen* Werten besser als m_i ist (m_j **dominiert** m_i).

Definition (Pareto-Optimum)

Zu einer Menge M von n -Tupeln heißt ein Tupel $m_i = (x_1, \dots, x_n) \in M$ **Pareto-Optimum**, wenn es kein anderes $m_j \in M$ gibt, so dass m_j in *allen* Werten besser als m_i ist (m_j **dominiert** m_i).

Die Menge M heißt **Pareto-Menge**, wenn alle $m \in M$ Pareto-optimal.

Definition (Pareto-Optimum)

Zu einer Menge M von n -Tupeln heißt ein Tupel $m_i = (x_1, \dots, x_n) \in M$ **Pareto-Optimum**, wenn es kein anderes $m_j \in M$ gibt, so dass m_j in *allen* Werten besser als m_i ist (m_j **dominiert** m_i).

Die Menge M heißt **Pareto-Menge**, wenn alle $m \in M$ Pareto-optimal.

Beispiel: Betrachte Tupel aus Ankunftszeit und # Umstiege.

$M = \{(14:00 \text{ Uhr}, 5), (15:13 \text{ Uhr}, 3), (13:45 \text{ Uhr}, 4), (15:15 \text{ Uhr}, 0)\}$

Definition (Pareto-Optimum)

Zu einer Menge M von n -Tupeln heißt ein Tupel $m_i = (x_1, \dots, x_n) \in M$ **Pareto-Optimum**, wenn es kein anderes $m_j \in M$ gibt, so dass m_j in *allen* Werten besser als m_i ist (m_j **dominiert** m_i).

Die Menge M heißt **Pareto-Menge**, wenn alle $m \in M$ Pareto-optimal.

Beispiel: Betrachte Tupel aus Ankunftszeit und # Umstiege.

$M = \{(14:00 \text{ Uhr}, 5), (15:13 \text{ Uhr}, 3), (13:45 \text{ Uhr}, 4), (15:15 \text{ Uhr}, 0)\}$

Definition (Pareto-Optimum)

Zu einer Menge M von n -Tupeln heißt ein Tupel $m_i = (x_1, \dots, x_n) \in M$ **Pareto-Optimum**, wenn es kein anderes $m_j \in M$ gibt, so dass m_j in *allen* Werten besser als m_i ist (m_j **dominiert** m_i).

Die Menge M heißt **Pareto-Menge**, wenn alle $m \in M$ Pareto-optimal.

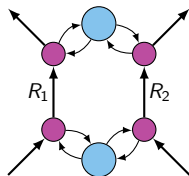
Beispiel: Betrachte Tupel aus Ankunftszeit und # Umstiege.

$M = \{(14:00 \text{ Uhr}, 5), (15:13 \text{ Uhr}, 3), (13:45 \text{ Uhr}, 4), (15:15 \text{ Uhr}, 0)\}$

Wie effizient berechnen?

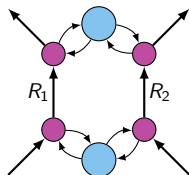
Idee:

- Benutze (zeitabhängiges) Graphmodell
- Grundlage: Dijkstras Algorithmus



Idee:

- Benutze (zeitabhängiges) Graphmodell
- Grundlage: Dijkstras Algorithmus



... aber ...

- Label ℓ sind 2-Tupel aus Ankunftszeit und # Umstiege
- An jedem Knoten $v \in V$: Pareto-Menge B_v von Labeln
- Priority Queue verwaltet Label statt Knoten
- Priorität ist Ankunftszeit
(Wieder: keine Totalordnung der Label \Rightarrow label-correcting)
- Dominanz von Labeln in B_v on-the-fly

Multi-Label-Correcting (MLC)

$MLC(G = (V, E), s, \tau_{dep})$

```
1  $B_v \leftarrow \{\}$  for each  $v \in V$ ;  $B_s \leftarrow \{(\tau_{dep}, 0)\}$ 
2  $Q.clear()$ ,  $Q.insert(s, (\tau_{dep}, 0))$ 
3 while  $!Q.empty()$  do
4    $u$  and  $\ell = (\tau, tr) \leftarrow Q.deleteMin()$ 
5   for all edges  $e = (u, v) \in E$  do
6     if  $e$  is a transfer edge then  $tr' \leftarrow tr + 1$ 
7     else  $tr' \leftarrow tr$ 
8      $\ell' \leftarrow (\tau + \text{len}(e, \tau), tr')$ 
9     if  $\ell'$  is not dominated by any  $\ell'' \in B_v$  then
10       $B_v.insert(\ell')$ 
11      Remove non-Pareto-optimal labels from  $B_v$ 
12       $Q.insert(v, \ell')$ 
```

Diskussion:

- Pareto-Mengen B_v sind *dynamische* Datenstrukturen \rightsquigarrow teuer!
- Sehr viele Queue-Operationen
- Testen der Dominanz in $\mathcal{O}(|B_v|)$ möglich
- Stoppkriterium?

Diskussion:

- Pareto-Mengen B_v sind *dynamische* Datenstrukturen \rightsquigarrow teuer!
- Sehr viele Queue-Operationen
- Testen der Dominanz in $\mathcal{O}(|B_v|)$ möglich
- Stoppkriterium?

Verbesserungen für MLC:

- Jedes B_v verwaltet bestes ungesetztes Label selbst
 \Rightarrow Priority Queue auf Knoten statt Labeln
- Label-Forwarding:
Wenn Kante keine Kosten hat, überspringe Queue
- Target-Pruning:
Verwerfe Label ℓ' an Knoten v , wenn es von B_t dominiert wird



Daniel Delling, Bastian Katz, and Thomas Pajor.
Parallel Computation of Best Connections in Public Transportation Networks.
ACM Journal of Experimental Algorithmics, 17(4):4.1–4.26, July 2012.



Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee.
Multi-Criteria Shortest Paths in Time-Dependent Train Networks.
In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.



Daniel Delling, Thomas Pajor, and Dorothea Wagner.
Engineering Time-Expanded Graphs for Faster Timetable Information.
In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 182–206. Springer, 2009.



Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis.
Efficient Models for Timetable Information in Public Transportation Systems.
ACM Journal of Experimental Algorithmics, 12(2.4):1–39, 2008.