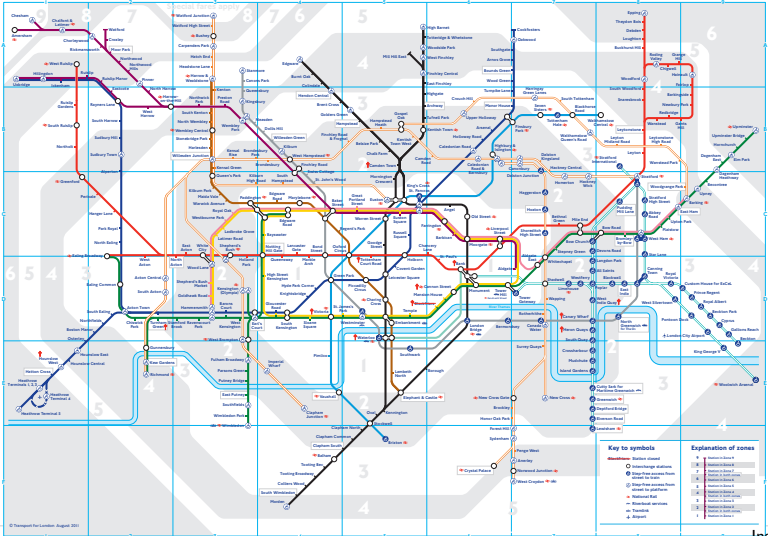


# Algorithmen für Routenplanung

17. Vorlesung, Sommersemester 2024

Moritz Laupichler | 24. Juni 2024





## Eingabe bei Straßennetzen:

Straßenkarte modelliert als Graph, bestehend aus

- Kreuzungen  $\hat{=}$  Knoten
- Straßensegmenten  $\hat{=}$  Kanten
- Metriken (Reisezeit, Distanz, ...)  $\hat{=}$  Kantengewichte

## Eingabe bei Straßennetzen:

Straßenkarte modelliert als Graph, bestehend aus

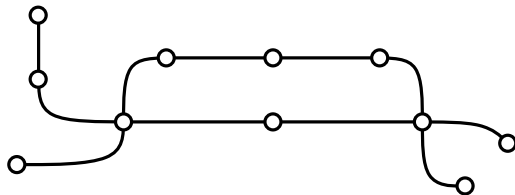
- Kreuzungen  $\hat{=}$  Knoten
- Straßensegmenten  $\hat{=}$  Kanten
- Metriken (Reisezeit, Distanz, ...)  $\hat{=}$  Kantengewichte

Was ist die Eingabe bei ÖV-Netzwerken?

# Fahrpläne

## Fahrplan:

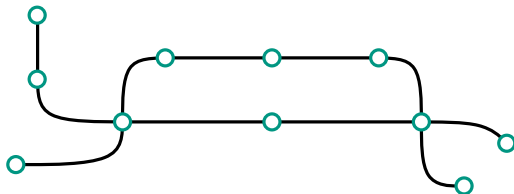
- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)



# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)



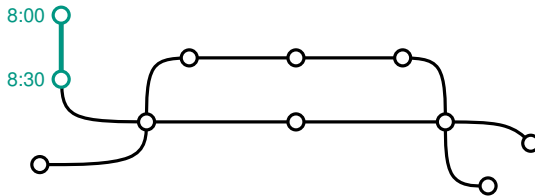
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von **Connections** (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

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

- Abfahrtsstop  $v_{\text{dep}}(c) \in \mathcal{S}$
- Abfahrtszeit  $\tau_{\text{dep}}(c) \in \Pi$
- Ankunftsstop  $v_{\text{arr}}(c) \in \mathcal{S}$
- Ankunftszeit  $\tau_{\text{arr}}(c) \in \Pi$
- Trip  $\text{trip}(c) \in \mathcal{T}$
- Kein Zwischenhalt!



# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## 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 1



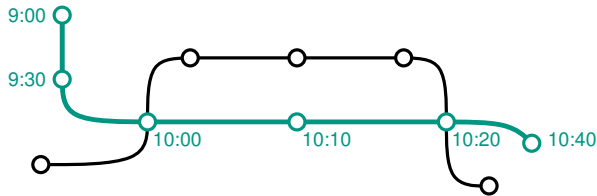
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## 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 2

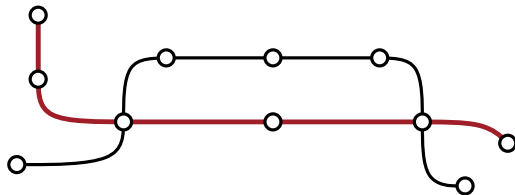
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## Routen: (Linien)

- Partitionierung der Trips
- Zwei Trips  $T_1, T_2$  gehören zur gleichen Route
  - $\Leftrightarrow T_1$  und  $T_2$  besuchen genau die gleiche Sequenz von Stops
- Ist bei realen Zug-/Bahnlagen nicht immer der Fall!



Route 1

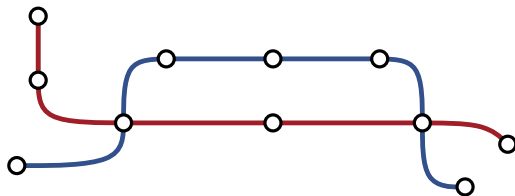
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## Routen: (Linien)

- Partitionierung der Trips
- Zwei Trips  $T_1, T_2$  gehören zur gleichen Route
  - $\Leftrightarrow T_1$  und  $T_2$  besuchen genau die gleiche Sequenz von Stops
- Ist bei realen Zug-/Bahnlagen nicht immer der Fall!



Route 1

Route 2

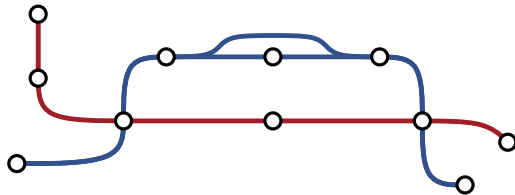
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## Routen: (Linien)

- Partitionierung der Trips
- 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-/Bahnlinsen nicht immer der Fall!



Route 1  
keine Route!

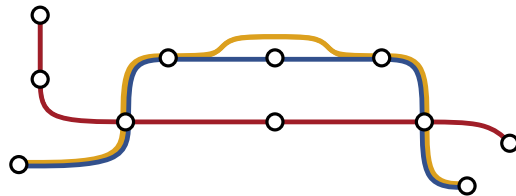
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## Routen: (Linien)

- Partitionierung der Trips
- 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-/Bahnlagen nicht immer der Fall!



Route 1  
 Route 2  
 Route 3

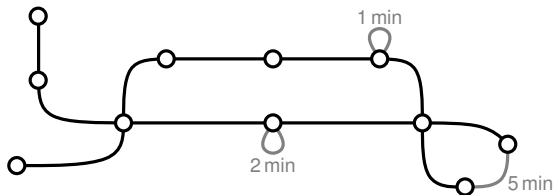
# Fahrpläne

## Fahrplan:

- Menge  $\mathcal{S}$  von Stops (Bahnhöfe, Bahnsteige, Haltestellen, ...)
- Menge  $\mathcal{C}$  von Connections (elementare Verbindungen)
- Menge  $\mathcal{T}$  von Trips (Züge, Busse, Trams, ...)

## Zur Modellierung von Umstiegen:

- Mindestumstiegszeiten an Stops:  $\tau_{\text{ch}}: \mathcal{S} \rightarrow \mathbb{N}_0$
- Fußwege zwischen (nahen) Stops:  $\tau_{\text{walk}}: \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{N}_0 \cup \{\infty\}$

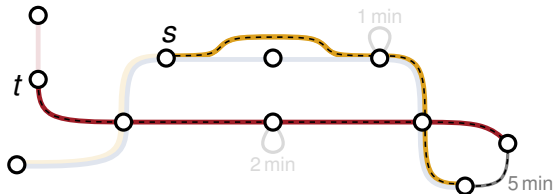


Gegenstück zu einem Pfad in einem Straßennetzwerk?

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 Umstiegen/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 Umstiegen/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 Umstieg/Fußweg)

## Zwei grundlegende Ansätze:

- 1 Modellierung als gerichteter Graph
- 2 Benutze Fahrplan „direkt“

↔ Dijkstra

↔ Neue Algorithmen

## Zwei grundlegende Ansätze:

- 1 Modellierung als gerichteter Graph
- 2 Benutze Fahrplan „direkt“

↔ Dijkstra

↔ Neue Algorithmen

Jetzt ersteres, später zweiteres.

## Zwei grundlegende Ansätze:

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

Jetzt ersteres, später zweiteres.

## Modellierung als Graph:

- Reduziere auf (bekanntes) Kürzeste-Wege-Problem
- Optimale Journeys entsprechen kürzesten Wegen

## Zwei grundlegende Ansätze:

- 1 Modellierung als gerichteter Graph ↔ Dijkstra
- 2 Benutze Fahrplan „direkt“ ↔ 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

---

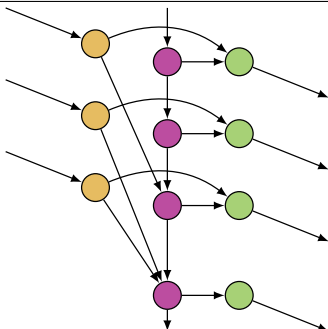
- Zeitabhängigkeiten ausrollen
  - Knoten entsprechen Ereignissen im Fahrplan
  - Kanten verbinden Ereignisse miteinander
    - Fahrt mit Connection
    - Umstieg zwischen Trips
    - Warten auf Trip
- Großer Graph
- + Einfacher Anfragealgorithmus (Dijkstra)

## 2. Zeitabhängig

---

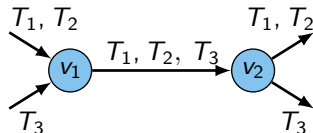
- Zeitabhängigkeit in den Kantengewichten
  - Knoten entsprechen Stops
  - Kante  $\Leftrightarrow$  Connection verbindet Stops
    - Umstiegszeiten?
- + Kleiner Graph
- Zeitabhängige Algorithmen?

## 1. Zeitexpandiert



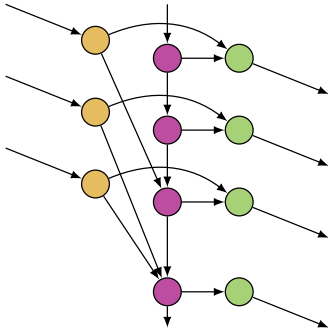
- Arrival-, Transfer- und Departure-Ereignisse
- Für jede Connection
- Kantengewicht = Zeitdiff.  
(alternativ: ungewichtet, Knotenlabel = Ereigniszeit)

## 2. Zeitabhängig



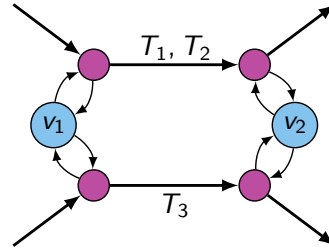
- Pro Stop: Stationsknoten
- Kanten: zeitabhängig
- Umstiege?

## 1. Zeitexpandiert



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

## 2. Zeitabhängig



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

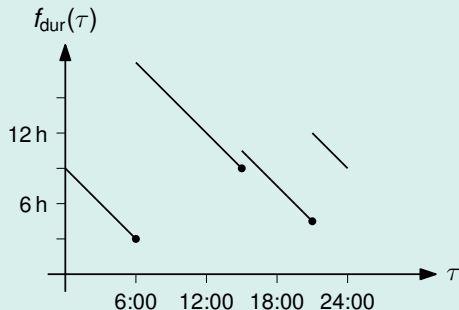


Connections modelliert durch stückweise lineare Funktionen  $f_{\text{dur}}^e, f_{\text{arr}}^e$

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
⋮	⋮	⋮

Entsprechende Funktion:



- Für jede Connection: Connection Point  $(\tau, w)$   
 $\tau \triangleq$  Abfahrtszeit,  $w \triangleq$  Reisezeit (bzw. Ankunftszeit)
- Zwischen Connections: Lineares Warten

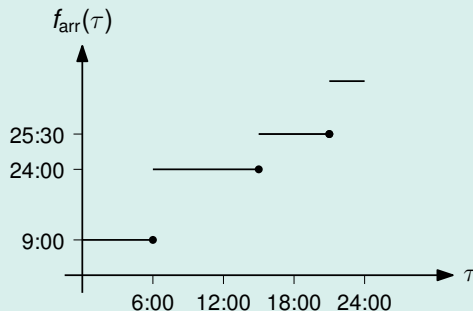
# Zeitabhängige Kanten

Connections modelliert durch stückweise lineare Funktionen  $f_{\text{dur}}^e, f_{\text{arr}}^e$

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  $(\tau, w)$   
 $\tau \hat{=}$  Abfahrtszeit,  $w \hat{=}$  Reisezeit (bzw. Ankunftszeit)
- Zwischen Connections: Lineares Warten

## Definition

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

$$f_{\text{dur}}(\tau) \leq \varepsilon + f_{\text{dur}}(\tau + \varepsilon).$$

## Diskussion:

- Interpretation: „Warten lohnt sich nie“
- Kürzeste-Wege-Problem ohne FIFO-Eigenschaft ist NP-schwer  
(wenn Warten an Knoten nicht erlaubt ist)

⇒ Sicherstellen, dass Funktionen FIFO-Eigenschaft erfüllen

## Definition

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

$$f_{\text{arr}}(\tau) \leq f_{\text{arr}}(\tau + \varepsilon).$$

## Diskussion:

- Interpretation: „Warten lohnt sich nie“
- Kürzeste-Wege-Problem ohne FIFO-Eigenschaft ist NP-schwer  
(wenn Warten an Knoten nicht erlaubt ist)

⇒ Sicherstellen, dass Funktionen FIFO-Eigenschaft erfüllen

# Fahrplanauskunft mit Dijkstra

# Anfrageszenarien

**Gegeben:** Startstop  $s$ , Zielstop  $t$

**Gegeben:** Startstop  $s$ , Zielstop  $t$

**Zeit-Anfrage (Earliest Arrival Query):**

- Finde schnellste Journey für Abfahrtszeit  $\tau_{\text{dep}}$
- Analog zu Dijkstra?

**Gegeben:** Startstop  $s$ , Zielstop  $t$

**Zeit-Anfrage (Earliest Arrival Query):**

- Finde schnellste Journey für Abfahrtszeit  $\tau_{\text{dep}}$
- Analog zu Dijkstra?

**Profil-Anfrage (Profile Query):**

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



# Zeit-Anfragen

**Gegeben:** Startstop  $s$ , Zielstop  $t$  und Abfahrtszeit  $\tau_{\text{dep}}$

**Gegeben:** Startstop  $s$ , Zielstop  $t$  und Abfahrtszeit  $\tau_{\text{dep}}$

## 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  $\tau_{\text{dep}}$
- Nur 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  $Q.\text{clear}(), Q.\text{insert}(s, \tau_{\text{dep}})$ 
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  $Q.\text{insert}(v, d[v])$ 
```

# Diskussion Zeit-Anfragen

## 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)

# Profil-Anfragen

**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  
(z.B.:  $d[u] := \text{Minimum der Funktion } d_*[u]$ )
- Knoten können mehrfach besucht werden  $\Rightarrow$  label-correcting

## Funktion gegeben durch:

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

## 3 Operationen notwendig:

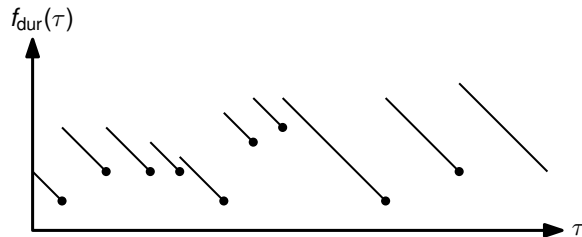
- Auswertung
- Linken  $\oplus$
- Minimumsbildung

# Public Transport: Auswertung

## Auswertung von $f(\tau)$ :

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

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



# Public Transport: Auswertung

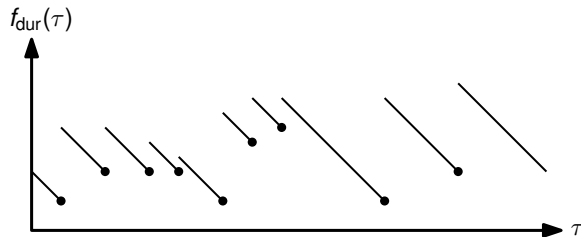
## Auswertung von $f(\tau)$ :

- 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  $t_i$
- Theoretisch:
  - Lineare Suche:  $\mathcal{O}(|I|)$
  - Binäre Suche:  $\mathcal{O}(\log_2 |I|)$



# Public Transport: Auswertung

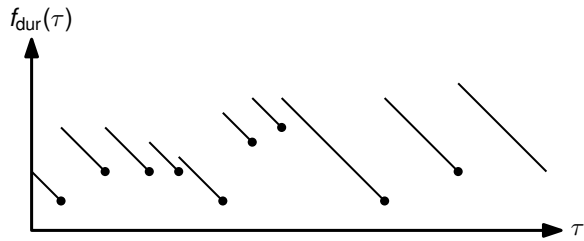
## Auswertung von $f(\tau)$ :

- 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  $t_i$
- Theoretisch:
  - Lineare Suche:  $\mathcal{O}(|I|)$
  - Binäre Suche:  $\mathcal{O}(\log_2 |I|)$
- Praktisch:
  - ggf. lineare Suche mit Startpunkt  $\frac{\tau}{\Pi} \cdot |I|$   
(wobei  $\Pi$  die Periodendauer ist)
  - Benchmarken!



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

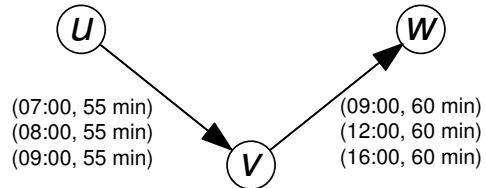
$$f \oplus g := g \circ f$$

Oder

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

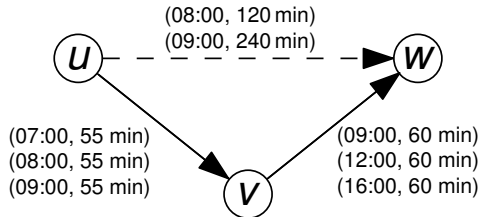
# Public Transport: Link

Linken zweier Funktionen  $f$  und  $g$ :



# Public Transport: Link

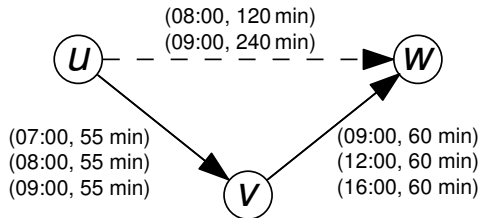
Linken zweier Funktionen  $f$  und  $g$ :



# Public Transport: Link

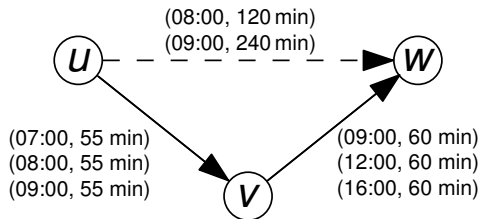
## Linken zweier Funktionen $f$ und $g$ :

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



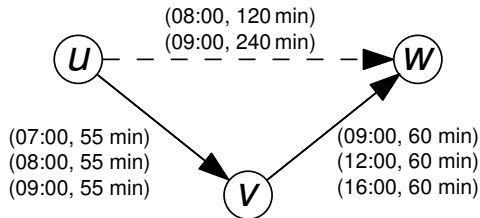
## Linken zweier Funktionen $f$ und $g$ :

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



## Linken zweier Funktionen $f$ und $g$ :

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

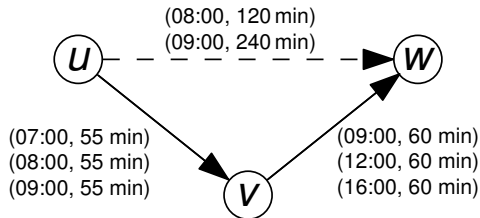


## Linken zweier Funktionen $f$ und $g$ :

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

- Füge  $(t_i^f, t_j^g - t_i^f + w_j^g)$  zum Ergebnis hinzu
- Wenn zwei Punkte den gleichen Verbindungspunkt haben:  
Behalte nur den mit größerem  $t_i^f$

⇒ Sweep-Algorithmus



## Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$



## Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$

## Speicherverbrauch:

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

## Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$

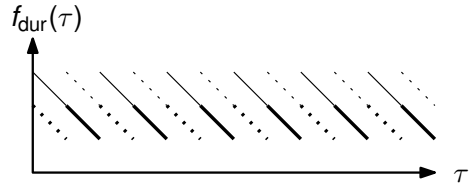
## Speicherverbrauch:

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

## Auf Straßengraphen (zum Vergleich):

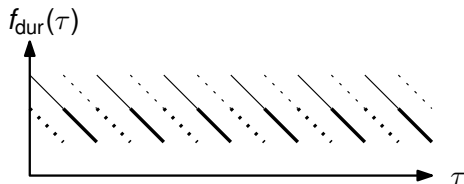
- Laufzeit auch  $\mathcal{O}(|I^f| + |I^g|)$
- Aber  $\approx |I^f| + |I^g|$  Interpolationspunkte

Minimum zweier Funktionen  $f$  und  $g$ :



## Minimum zweier Funktionen $f$ und $g$ :

- Für alle  $(t_i^f, w_i^f)$ : Behalte Punkt, wenn  $w_i^f \leq g(t_i^f)$
- Für alle  $(t_j^g, w_j^g)$ : Behalte Punkt, wenn  $w_j^g \leq f(t_j^g)$   
(Bei Gleichheit Duplikate entfernen)
- Keine Schnittpunkte möglich!

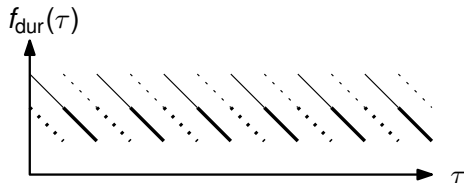


## Minimum zweier Funktionen $f$ und $g$ :

- Für alle  $(t_i^f, w_i^f)$ : Behalte Punkt, wenn  $w_i^f \leq g(t_i^f)$
- Für alle  $(t_j^g, w_j^g)$ : Behalte Punkt, wenn  $w_j^g \leq f(t_j^g)$   
(Bei Gleichheit Duplikate entfernen)
- Keine Schnittpunkte möglich!

## Vorgehen:

- Linearer Sweep



## Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$

## Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$

## Speicherverbrauch:

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

## Laufzeit:

- Sweep-Algorithmus
- $\mathcal{O}(|I^f| + |I^g|)$

## Speicherverbrauch:

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

## Auf Straßengraphen (zum Vergleich):

- Laufzeit auch  $\mathcal{O}(|I^f| + |I^g|)$
- Aber mehr als  $|I^f| + |I^g|$  Interpolationspunkte (Schnittpunkte!)



# Self-Pruning Connection-Setting (SPCS)

# Profil-Anfragen

## Gegeben:

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

## Gegeben:

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

## Problem (Profil-Anfrage):

Berechne Reisezeitfunktion  $\text{dist}_s(v, \tau)$  für alle  $\tau \in \Pi$  und  $v \in V$ :  
Länge des kürzesten Weges von  $s$  nach  $v$  in  $G$  zur Abfahrtszeit  $\tau$

## Gegeben:

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

## Problem (Profil-Anfrage):

Berechne Reisezeitfunktion  $\text{dist}_s(v, \tau)$  für alle  $\tau \in \Pi$  und  $v \in V$ :  
Länge des kürzesten Weges von  $s$  nach  $v$  in  $G$  zur Abfahrtszeit  $\tau$

## 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 initialen Connection an  $s$ .

# Hauptidee

**Beobachtung:** Jede Journey ab  $s$  (irgendwohin) beginnt mit einer initialen 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 initialen 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 initiale Connection an  $s$  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 initialen 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 initiale Connection an  $s$  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 initialen 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 initiale Connection an  $s$  trägt zu  $\text{dist}_s(v, \cdot)$  bei

Langsame Züge für weite Reisen machen wenig Sinn

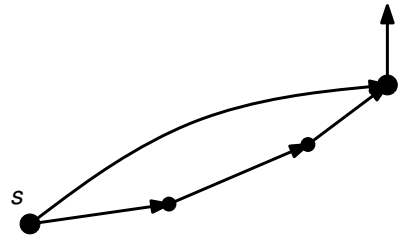
<del>0</del>	1	<del>2</del>	<del>3</del>	<del>4</del>	5	<del>6</del>	7	<del>8</del>	<del>9</del>	<del>10</del>	11
<del>6:30 8:30</del>	7:04 8:30	<del>9:26 14:28</del>	<del>10:34 14:28</del>	<del>11:08 14:28</del>	12:42 14:28	<del>13:01 16:46</del>	13:58 16:46	<del>16:46 23:30</del>	<del>18:24 23:30</del>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

("Connection reduction")

# Self-Pruning

## Beobachtung:

Initiale Connections können sich dominieren.



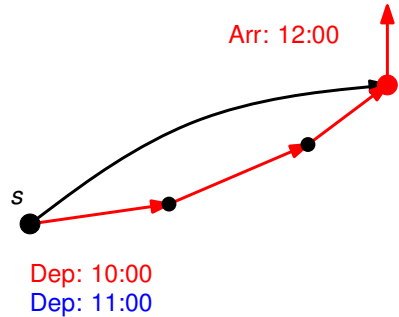
Dep: 10:00

Dep: 11:00

# Self-Pruning

## Beobachtung:

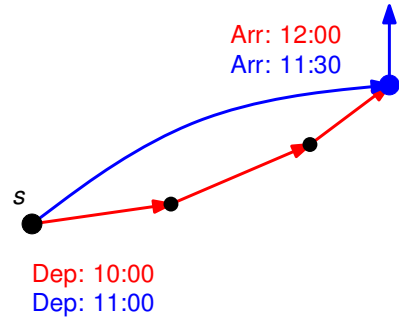
Initiale Connections können sich dominieren.



# Self-Pruning

## Beobachtung:

Initiale Connections können sich dominieren.

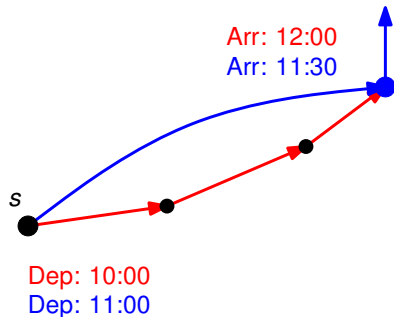


## Beobachtung:

Initiale Connections können sich dominieren.

## Self-Pruning (SP):

- Benutze eine gemeinsame Queue
- Sortiere initiale Connections  $c_i$  aufsteigend nach Abfahrtszeit
- Speichere an erreichten Knoten Index der initialen Connection
- Keys sind Ankunftszeiten



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

## Integration von Self-Pruning:

- Verwalte Label  $\text{maxconn}(v)$  an jedem Knoten  $v$   
Gibt maximale initiale 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 initiale 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 initialer Connection  $j > i$ , **prune**  $i$  an  $v$

## Integration von Self-Pruning:

- Verwalte Label  $\text{maxconn}(v)$  an jedem Knoten  $v$   
Gibt maximale initiale 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 initiale 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  $v$

**Dijkstras Label-Setting-Eigenschaft pro initialer Connection wiederhergestellt**

## Integration von Self-Pruning:

- Verwalte Label  $\text{maxconn}(v)$  an jedem Knoten  $v$   
Gibt maximale initiale 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  $v$

**Dijkstras Label-Setting-Eigenschaft pro initialer Connection wiederhergestellt**

⇒ Self-Pruning Connection-Setting Algorithmus (SPCS)

# Stoppkriterium

(Falls Zielknoten  $t$  gegeben)

## **Dijkstras Algorithmus:**

Breche die Suche ab, sobald  $t$  abgearbeitet wurde.

# Stoppkriterium

(Falls Zielknoten  $t$  gegeben)

## Dijkstras Algorithmus:

Breche die Suche ab, sobald  $t$  abgearbeitet wurde.

kann adaptiert werden durch

## Self-Pruning Connection-Setting:

- Verwalte Label  $T_m := -\infty$

# Stoppkriterium

(Falls Zielknoten  $t$  gegeben)

## Dijkstras Algorithmus:

Breche die Suche ab, sobald  $t$  abgearbeitet wurde.

kann adaptiert werden durch

## Self-Pruning Connection-Setting:

- Verwalte Label  $T_m := -\infty$
- Wenn  $t$  mit initialer Connection  $i$  gesettled wird, setze  $T_m := \max\{T_m, i\}$

(Falls Zielknoten  $t$  gegeben)

## Dijkstras Algorithmus:

Breche die Suche ab, sobald  $t$  abgearbeitet wurde.

kann adaptiert werden durch

## Self-Pruning Connection-Setting:

- Verwalte Label  $T_m := -\infty$
- Wenn  $t$  mit initialer Connection  $i$  gesettled wird, setze  $T_m := \max\{T_m, i\}$
- Prune alle Labels mit init. Connection  $j \leq T_m$  (an jedem Knoten)

(Falls Zielknoten  $t$  gegeben)

## Dijkstras Algorithmus:

Breche die Suche ab, sobald  $t$  abgearbeitet wurde.

kann adaptiert werden durch

## Self-Pruning Connection-Setting:

- Verwalte Label  $T_m := -\infty$
- Wenn  $t$  mit initialer Connection  $i$  gesettled wird, setze  $T_m := \max\{T_m, i\}$
- Prune alle Labels mit init. Connection  $j \leq T_m$  (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 initiale Connections  $c_i$  von  $s$  auf verschiedene Threads

0			3			6			9		
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
Thread 0			Thread 1			Thread 2			Thread 3		

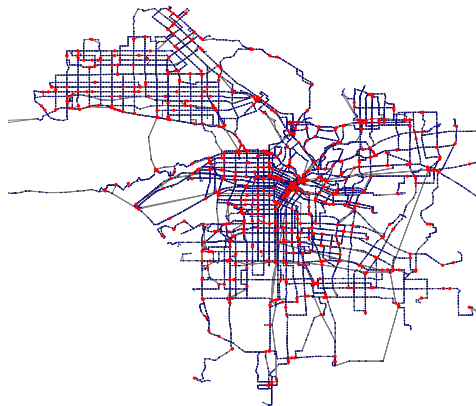
- Jeder Thread führt SPCS auf seiner Teilmenge der Connections aus
- Ergebnisse werden im Anschluss zu  $\text{dist}_s(v, \cdot)$  zusammengeführt
- Führe Connection Reduction auf gemergtem Ergebnis durch

Netzwerk von **Los Angeles**:

- 15 581 Stops
- 1 046 580 Connections

Zugnetz von **Europa**:

- 30 517 Stops
- 1 775 533 Connections

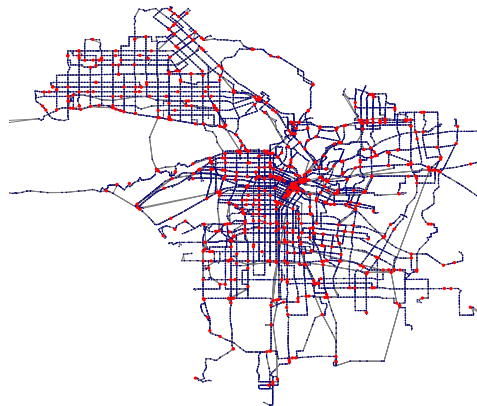


Netzwerk von **Los Angeles**:

- 15 581 Stops
- 1 046 580 Connections

Zugnetz von **Europa**:

- 30 517 Stops
- 1 775 533 Connections



Auswertung von 1 000 Anfragen  
Wähle Start- und Zielstops zufällig gleichverteilt

		Los Angeles				Europe			
		Settled	Time	Spd	Std-	Settled	Time	Spd	Std-
		$\rho$ Conns	[ms]	Up	Dev	Conns	[ms]	Up	Dev
<b>PSPCS</b>	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 %
<b>LC</b>	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

		Los Angeles				Europe			
		Settled $\rho$ Conns	Time [ms]	Spd Up	Std- Dev	Settled Conns	Time [ms]	Spd Up	Std- Dev
<b>PSPCS</b>	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 %
<b>LC</b>	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

- PSPCS betrachtet deutlich weniger Connections als LC

		Los Angeles				Europe			
		Settled $\rho$ Conns	Time Spd [ms] Up	Std- Dev					
		Conns	[ms]	Up	Dev	Settled Conns	Time Spd [ms] Up	Std- Dev	
<b>PSPCS</b>	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 %
<b>LC</b>	1	18.9 M	1482.1	—	—	17.4 M	2497.1	—	—

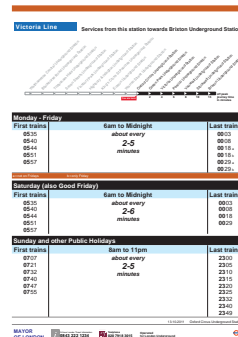
- PSPCS betrachtet deutlich weniger Connections als LC
- PSPCS skaliert sehr gut mit zunehmender Anzahl Cores

# Multi-Label-Correcting (MLC)

# Bis jetzt...

**Erinnerung:** Ein Fahrplan besteht aus

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





# Bis jetzt...

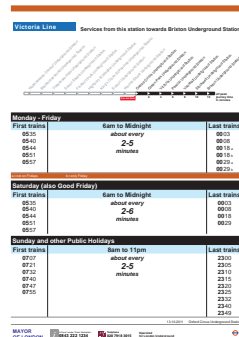
**Erinnerung:** Ein Fahrplan besteht aus

- 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  $\tau$ , berechne

- Journey zu  $t$ , die an  $s$  nicht früher als  $\tau$  abfährt
- und an  $t$  frühestmöglich ankommt.



# Bis jetzt...

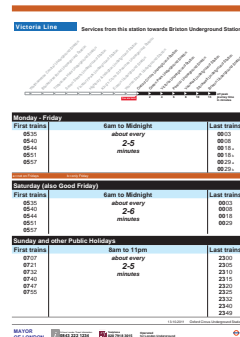
**Erinnerung:** Ein Fahrplan besteht aus

- 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  $\tau$ , berechne

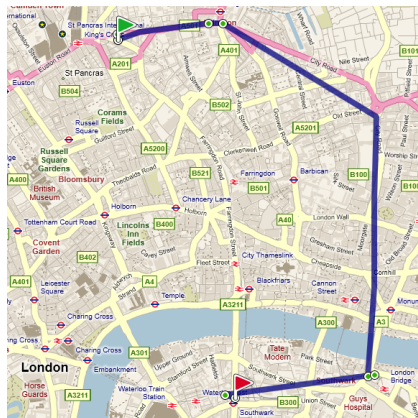
- Journey zu  $t$ , die an  $s$  nicht früher als  $\tau$  abfährt
- und an  $t$  frühestmöglich ankommt.



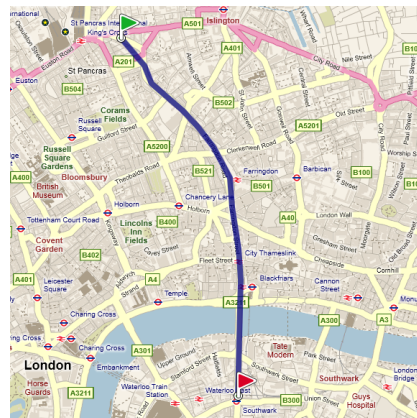
Reicht uns das?

# Einbeziehen von Umstiegen

Umstiege zu betrachten ist wichtig!



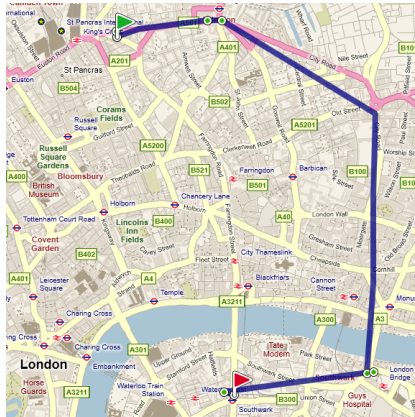
Ankunft 11:08 Uhr, 2 Umstiege



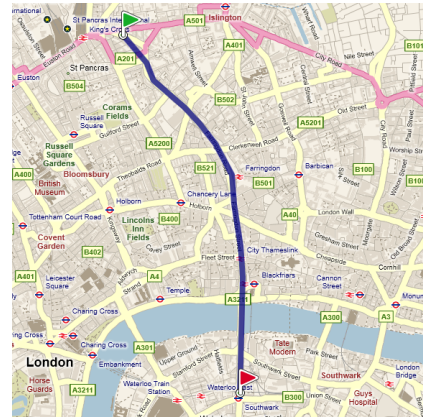
Ankunft 11:09 Uhr, 0 Umstiege

# Einbeziehen von Umstiegen

Umstiege zu betrachten ist wichtig!



Ankunft 11:08 Uhr, 2 Umstiege



Ankunft 11:09 Uhr, 0 Umstiege

Berechne „gute“ Journeys für Ankunftszeit und Anzahl Umstiege.

## Definition (Pareto-Optimum)

Gegeben eine Menge  $M$  von  $n$ -Tupeln:

Tupel  $m_i = (x_1, \dots, x_n) \in M$  heißt Pareto-Optimum

$\Leftrightarrow$

$\nexists m_j \in M: m_j \neq m_i$  und  $m_i$  ist in keinem Wert besser als  $m_j$   
( $m_j$  dominiert  $m_i$ )

## Definition (Pareto-Optimum)

Gegeben eine Menge  $M$  von  $n$ -Tupeln:

Tupel  $m_i = (x_1, \dots, x_n) \in M$  heißt Pareto-Optimum

$\Leftrightarrow$

$\nexists m_j \in M: m_j \neq m_i$  und  $m_i$  ist in keinem Wert besser als  $m_j$   
( $m_j$  dominiert  $m_i$ )

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

## Definition (Pareto-Optimum)

Gegeben eine Menge  $M$  von  $n$ -Tupeln:

Tupel  $m_i = (x_1, \dots, x_n) \in M$  heißt Pareto-Optimum

$\Leftrightarrow$

$\nexists m_j \in M: m_j \neq m_i$  und  $m_i$  ist in keinem Wert besser als  $m_j$   
( $m_j$  dominiert  $m_i$ )

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)

Gegeben eine Menge  $M$  von  $n$ -Tupeln:

Tupel  $m_i = (x_1, \dots, x_n) \in M$  heißt Pareto-Optimum

$\Leftrightarrow$

$\nexists m_j \in M: m_j \neq m_i$  und  $m_i$  ist in keinem Wert besser als  $m_j$   
( $m_j$  dominiert  $m_i$ )

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)

Gegeben eine Menge  $M$  von  $n$ -Tupeln:

Tupel  $m_i = (x_1, \dots, x_n) \in M$  heißt Pareto-Optimum

$\Leftrightarrow$

$\nexists m_j \in M: m_j \neq m_i$  und  $m_i$  ist in keinem Wert besser als  $m_j$   
( $m_j$  dominiert  $m_i$ )

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

**Beispiel:** Betrachte Tupel aus Ankunftszeit und # Umstiege.

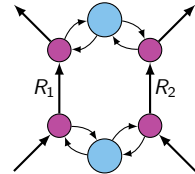
$M = \{(14:00 \text{ Uhr}, 5), (15:13 \text{ Uhr}, 3), (13:45 \text{ Uhr}, 4), (15:15 \text{ Uhr}, 0)\}$

Wie effizient berechnen?

# Multi-Label-Correcting – Ansatz

## Idee:

- Benutze (zeitabhängiges) Graphmodell
- Grundlage: Dijkstras Algorithmus

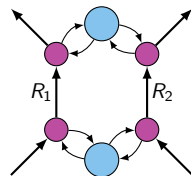


## Idee:

- Benutze (zeitabhängiges) Graphmodell
- Grundlage: Dijkstras Algorithmus

## Aber:

- Label  $\ell$  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



---

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

---

```
1  $B_v \leftarrow \{\}$  for each  $v \in V$ ;  $B_s \leftarrow \{(\tau_{\text{dep}}, 0)\}$ 
2  $Q.clear()$ ,  $Q.insert(s, (\tau_{\text{dep}}, 0))$ 
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  $B_v$  and  $Q$ 
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  $\ell'$  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.