

# Algorithmen für Routenplanung

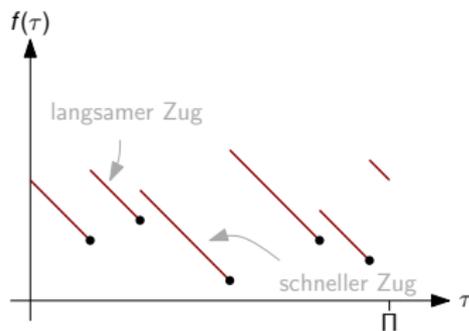
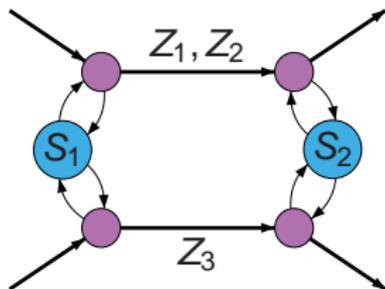
15. Sitzung, Sommersemester 2013

Thomas Pajor | 26. Juni 2013

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



## Einführung in die Fahrplanauskunft



Funktion  $f$  durch Punkte:  $I^f := \{(t_1^f, w_1^f), \dots, (t_k^f, w_k^f)\}$

**Gegeben:** Startbahnhof  $S$ , Zielbahnhof  $T$

**Gegeben:** Startbahnhof  $S$ , Zielbahnhof  $T$

## 1. Zeitexpandiert

?

## 2. Zeitabhängig

### Startknoten:

- Bahnhofsknoten  $S$

### Zielknoten:

- Bahnhofsknoten  $T$

### Anfrage:

- Label-Correcting Algorithmus von  $S$

---

**Algorithm 1:** Profile-Search( $G = (V, E), s$ )

---

```
1  $d_*[s] = 0$ 
2  $Q.clear(), Q.add(s, 0)$ 
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
10      else  $Q.insert(v, \underline{d}[v])$ 
```

---

## Beobachtungen:

- Operationen auf Funktionen
- Priorität im Prinzip frei wählbar  
( $d[u]$  ist das Minimum der Funktion  $d_*[u]$ )
- Knoten können mehrfach besucht werden  $\Rightarrow$  **label-correcting**

Technik	Preprocessing			Time-Queries		Profile-Queries	
	time [h:m]	space [B/n]	edge inc.	time [ms]	speed up	time [ms]	speed up
TIME-DEPENDENT APPROACH							
Dijkstra	0:00	0	0 %	125.2	1.0	5 327	1.0
uALT	0:02	128	0 %	75.3	1.7	4 384	1.2
eco-SHARC	1:30	113	74 %	17.5	7.2	988	5.4
gen-SHARC	12:15	120	74 %	4.7	26.6	273	19.5
CH*	0:08	87	86 %	0.6	209	9	591
TIME-EXPANDED APPROACH							
Dijkstra	0:00	0	0 %	534.7	1.0	—	—
uALT**	0:01	4	0 %	52.8	10.1	—	—
uALT+AF**	83:58	128	0 %	9.5	56.3	—	—

\* Auf Stationsgraph (Transfers durch Algo sichergestellt)  
 Kontraktion auf real. Graph nicht möglich (Kantenexplosion)

\*\* Optimiertes Routen-Modell

## Beobachtung:

Techniken für Straßennetzwerke funktionieren nicht gut

## Gründe (Intuition):

- Weniger Hierarchie (insbes. Busnetze)
  - Kontraktion lässt Knotengrad explodieren
  - Lokale Anfragen sehr “teuer”
- ↪ “15-Stunden zum nächsten Dorf Problem”
- Lower- und Upper-Bounds weit auseinander

## Beobachtung:

Techniken für Straßennetzwerke funktionieren nicht gut

## Gründe (Intuition):

- Weniger Hierarchie (insbes. Busnetze)
  - Kontraktion lässt Knotengrad explodieren
  - Lokale Anfragen sehr “teuer”
- ↪ “15-Stunden zum nächsten Dorf Problem”
- Lower- und Upper-Bounds weit auseinander

## Jetzt: Erster Ansatz zur Beschleunigung von Profil-Anfragen

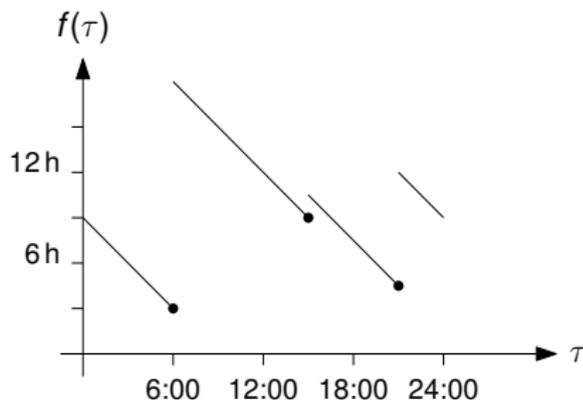
- Ausnutzen der Struktur von Schienenfunktionen
- Parallelisierung
- Stoppkriterium
- Distanztabelle

Verbindungen modelliert durch stückweise lineare Funktionen

Verbindungen zw.  $S_i$  und  $S_j$ :

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 Verbindung: **Connection Point**  $(\tau, w)$   
 $\tau \triangleq$  Abfahrtszeit,  $w \triangleq$  Reisezeit
- Zwischen Verbindungen: Lineares Warten

# Profil-Anfragen (One-to-All)

**Gegeben:**

Zeitabhängiges Netzwerk  $G = (V, E)$  und Startbahnhof  $S$ .

# Profil-Anfragen (One-to-All)

## Gegeben:

Zeitabhängiges Netzwerk  $G = (V, E)$  und Startbahnhof  $S$ .

## Problem (Profil-Anfrage):

Berechne die *Reisezeitfunktion*  $\text{dist}_S(v, \tau)$ , so dass  $\text{dist}_S(v, \tau)$  die Länge des **kürzesten Weges** von  $S$  nach  $v$  in  $G$  zur Abfahrtszeit  $\tau$  an  $S$  für **alle**  $\tau \in \Pi$  und  $v \in V$  ist.

## Gegeben:

Zeitabhängiges Netzwerk  $G = (V, E)$  und Startbahnhof  $S$ .

## Problem (Profil-Anfrage):

Berechne die *Reisezeitfunktion*  $\text{dist}_S(v, \tau)$ , so dass  $\text{dist}_S(v, \tau)$  die Länge des **kürzesten Weges** von  $S$  nach  $v$  in  $G$  zur Abfahrtszeit  $\tau$  an  $S$  für **alle**  $\tau \in \Pi$  und  $v \in V$  ist.

## Bisheriger Ansatz:

Erweitere Dijkstra's Algorithmus zu **Label-Correcting Algorithmus**

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

# Hauptidee

## Beobachtung:

Jeder Reiseplan ab  $S$  (irgendwohin) beginnt mit einer Verb. an  $S$ .

## Beobachtung:

Jeder Reiseplan ab  $S$  (irgendwohin) beginnt mit einer Verb. an  $S$ .

## Naiver Ansatz

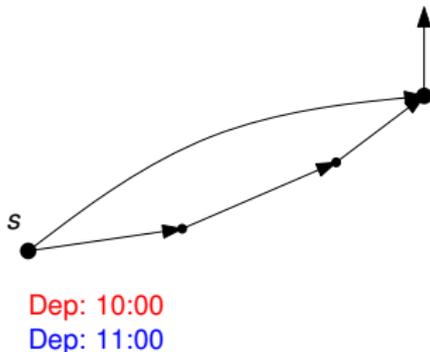
Für jede ausgehende Verbindung  $c_i$  an  $S$ :  
Separate Zeitanfrage mit Abfahrtszeit  $\tau_{\text{dep}}(c_i)$ .

## Nachteile

- Zu viele **redundante** Berechnungen  
Periodische Natur der Fahrpläne
- Nicht jede Verbindung ab  $S$  **trägt zu**  $\text{dist}_S(v, \cdot)$  **bei**  
Langsame Züge für weite Reisen machen wenig Sinn

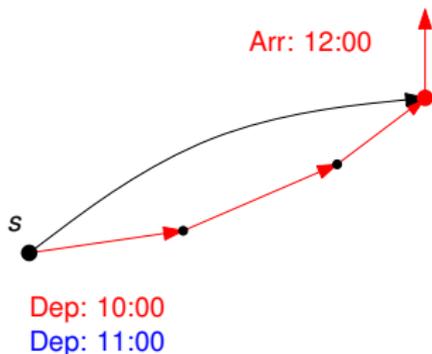
## Beobachtung:

Verbindungen können sich **dominieren**.



## Beobachtung:

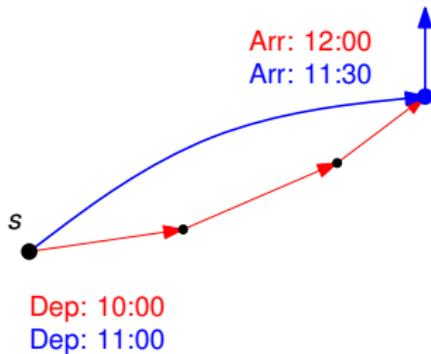
Verbindungen können sich **dominieren**.



# Self-Pruning

## Beobachtung:

Verbindungen können sich **dominieren**.

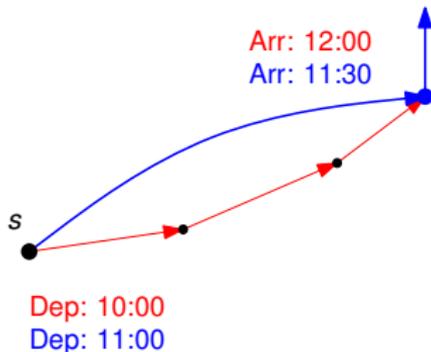


## Beobachtung:

Verbindungen können sich **dominieren**.

Einführung: **Self-Pruning** (SP):

1. Benutze **eine gemeinsame** Queue
2. Keys sind **Ankunftszeiten**
3. **Sortiere Verb.  $c_i$**  an  $S$  nach **Abfahrtszeit**



Beim Settlen von **Knoten  $v$**  und **Verb.-Index  $i$** :  
Prüfe ob  $v$  bereits gesettled mit **Verbindung  $j > i$** ; Dann **Prune  $i$**  an  $v$

## Integration von Self-Pruning (SP):

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

## Integration von Self-Pruning (SP):

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

Beim Settlen von Knoten  $v$  und Verb.-Index  $i$ :  
Prüfe ob  $v$  bereits gesettled mit Verbindung  $j > i$ ; Dann **Prune**  $i$  an  $v$

## Integration von Self-Pruning (SP):

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

Beim Settlen von Knoten  $v$  und Verb.-Index  $i$ :  
Prüfe ob  $\text{maxconn}(v) > i$ ; Dann **Prune**  $i$  an  $v$

## Integration von Self-Pruning (SP):

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

Beim Settlen von Knoten  $v$  und Verb.-Index  $i$ :  
Prüfe ob  $\text{maxconn}(v) > i$ ; Dann **Prune**  $i$  an  $v$

**Wiederherstellung von Dijkstra's Label-Setting Eigenschaft pro Verbindung**

## Integration von Self-Pruning (SP):

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

Beim Settlen von Knoten  $v$  und Verb.-Index  $i$ :  
Prüfe ob  $\text{maxconn}(v) > i$ ; Dann **Prune**  $i$  an  $v$

**Wiederherstellung von Dijkstra's Label-Setting Eigenschaft pro Verbindung**

⇒ Self-Pruning Connection-Setting Algorithmus (SPCS)

**Problem:** Falls nicht jede Verbindung optimal für  $\text{dist}_S(v, \cdot)$ .

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

**Problem:** Falls nicht jede Verbindung optimal für  $\text{dist}_S(v, \cdot)$ .

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

**Problem:** Falls nicht jede Verbindung optimal für  $\text{dist}_S(v, \cdot)$ .

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

**Problem:** Falls nicht jede Verbindung optimal für  $\text{dist}_S(v, \cdot)$ .

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

**Lösung:**

Connection-Reduction auf den Connection Points von  $\text{dist}_S(v, \cdot)$

- Wird nach dem Algorithmus durchgeführt
- Linearer Sweep von rechts nach links in  $\text{dist}_S(v, \cdot)$

# Parallelisierung: Idee

**Gegeben:**

Shared Memory Processing mit  $p$  Cores

# Parallelisierung: Idee

## Gegeben:

Shared Memory Processing mit  $p$  Cores

## Idee:

Verteile Verbindungen  $c_i$  von  $S$  auf verschiedene Threads

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 Verbindungen aus
- **Ergebnisse** werden im Anschluss zu  $\text{dist}_S(v, \cdot)$  zusammengeführt
- Führe **Connection Reduction** auf gemergtem Ergebnis durch

# Wahl der Partition

## Frage:

Gute Strategie zur Partitionierung der Verbindungen an  $S$

## Frage:

Gute Strategie zur Partitionierung der Verbindungen an  $S$

- EQUICONN:  
Gleiche Anzahl Verbindungen für jeden Thread

## Frage:

Gute Strategie zur Partitionierung der Verbindungen an  $S$

- EQUICONN:  
Gleiche **Anzahl** Verbindungen für jeden Thread
- EQUITIME:  
Gleiches **Zeitintervall** für jeden Thread

## Frage:

Gute Strategie zur Partitionierung der Verbindungen an  $S$

- EQUICONN:  
Gleiche **Anzahl** Verbindungen für jeden Thread
- EQUITIME:  
Gleiches **Zeitintervall** für jeden Thread
- K-MEANS  
Techniken aus dem Gebiet der Clusterung

## Frage:

Gute Strategie zur Partitionierung der Verbindungen an  $S$

- EQUICONN:  
Gleiche **Anzahl** Verbindungen für jeden Thread
- EQUITIME:  
Gleiches **Zeitintervall** für jeden Thread
- K-MEANS  
Techniken aus dem Gebiet der Clusterung

## Trade-off

Speed-up durch bessere Qualität der Partition

vs.

Berechnungsoverhead durch Partitionierung

## Problem:

Mit **zunehmender** Anzahl **Threads**, nimmt der Vorteil von **Self-Pruning** ab.

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

## Problem:

Mit **zunehmender** Anzahl **Threads**, nimmt der Vorteil von **Self-Pruning** ab.

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

## Problem:

Mit **zunehmender** Anzahl **Threads**, nimmt der Vorteil von **Self-Pruning** ab.

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

## Voraussetzung:

Jeder Thread berechnet nur aufeinanderfolgende Verbindungen

## Problem:

Mit **zunehmender** Anzahl **Threads**, nimmt der Vorteil von **Self-Pruning** ab.

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

## Voraussetzung:

Jeder Thread berechnet nur aufeinanderfolgende Verbindungen

## Inter-Thread-Pruning Regel:

Prune Verbindung  $i$  an Knoten  $v$  und Thread  $k$  wenn:

$\exists$  Thread  $l > k$  mit  $\min_{j \in \text{Thread } l} \{arr(v, j)\} \leq arr(v, i)$ .

## Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten  $v$  und Thread  $k$ : Verwalte Label  $\text{minarr}_k(v)$   
Beschreibt minimale Ankunftszeit an  $v$  (von Thread  $k$ )
- Update  $\text{minarr}_k(v)$  beim Settlen von  $v$  in Thread  $k$

## Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten  $v$  und Thread  $k$ : Verwalte Label  $\text{minarr}_k(v)$   
Beschreibt minimale Ankunftszeit an  $v$  (von Thread  $k$ )
- Update  $\text{minarr}_k(v)$  beim Settlen von  $v$  in Thread  $k$

## Inter-Thread-Pruning Rule:

Prune Verbindung  $i$  an Knoten  $v$  und Thread  $k$  wenn:  
 $\exists$  Thread  $l > k$  mit  $\min_{j \in \text{Thread } l} \{\text{arr}(v, j)\} \leq \text{arr}(v, i)$ .

## Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten  $v$  und Thread  $k$ : Verwalte Label  $\text{minarr}_k(v)$   
Beschreibt minimale Ankunftszeit an  $v$  (von Thread  $k$ )
- Update  $\text{minarr}_k(v)$  beim Settlen von  $v$  in Thread  $k$

## Inter-Thread-Pruning Rule:

Prune Verbindung  $i$  an Knoten  $v$  und Thread  $k$  wenn:  
 $\exists$  Thread  $l > k$  mit  $\text{minarr}_l(v) \leq \text{arr}(v, i)$ .

## Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten  $v$  und Thread  $k$ : Verwalte Label  $\text{minarr}_k(v)$   
Beschreibt minimale Ankunftszeit an  $v$  (von Thread  $k$ )
- Update  $\text{minarr}_k(v)$  beim Settlen von  $v$  in Thread  $k$

## Inter-Thread-Pruning Rule:

Prune Verbindung  $i$  an Knoten  $v$  und Thread  $k$  wenn:  
 $\exists$  Thread  $l > k$  mit  $\text{minarr}_l(v) \leq \text{arr}(v, i)$ .

- Verallgemeinerung von Self-Pruning auf mehrere Threads

## Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten  $v$  und Thread  $k$ : Verwalte Label  $\text{minarr}_k(v)$   
Beschreibt minimale Ankunftszeit an  $v$  (von Thread  $k$ )
- Update  $\text{minarr}_k(v)$  beim Settlen von  $v$  in Thread  $k$

## Inter-Thread-Pruning Rule:

Prune Verbindung  $i$  an Knoten  $v$  und Thread  $k$  wenn:  
 $\exists$  Thread  $l > k$  mit  $\text{minarr}_l(v) \leq \text{arr}(v, i)$ .

- Verallgemeinerung von Self-Pruning auf mehrere Threads
- Prüfen einer konstanten Anzahl von  $l > k$  ausreichend

# Station-to-Station Anfragen

## Eingabe:

Zeitabh. Netzwerk  $G = (V, E)$ , Start- und Zielbahnhöfe  $S$  und  $T$ .

# Station-to-Station Anfragen

## Eingabe:

Zeitabh. Netzwerk  $G = (V, E)$ , Start- und Zielbahnhöfe  $S$  und  $T$ .

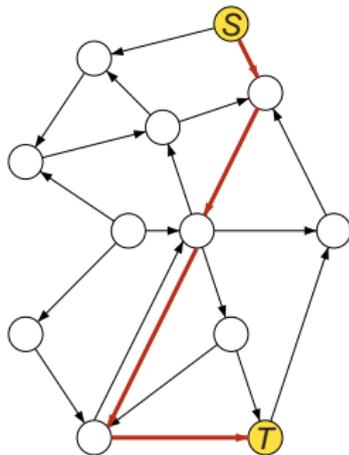
## Ziel:

Berechne  $\text{dist}_S(T, \cdot)$  nur für  $T$

## Intuition:

Weniger Berechnungsaufwand als für  $\text{dist}_S(\cdot, \cdot)$

⇒ **Beschleunigungstechniken**



# Station-to-Station Anfragen

## Eingabe:

Zeitabh. Netzwerk  $G = (V, E)$ , Start- und Zielbahnhöfe  $S$  und  $T$ .

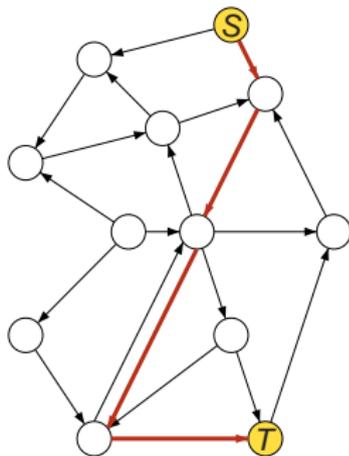
## Ziel:

Berechne  $\text{dist}_S(T, \cdot)$  nur für  $T$

## Intuition:

Weniger Berechnungsaufwand als für  $\text{dist}_S(\cdot, \cdot)$

⇒ **Beschleunigungstechniken**



Nicht-trivial für Dijkstra's Algorithmus (Diese Vorlesung) :-)

# Stoppkriterium

## Dijkstra's Algorithmus:

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

## Dijkstra's Algorithmus:

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

kann adaptiert werden durch

## Parallel Self-Pruning Connection-Setting:

- Verwalte globales Label  $T_m := -\infty$

## Dijkstra's Algorithmus:

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

kann adaptiert werden durch

## Parallel Self-Pruning Connection-Setting:

- Verwalte globales Label  $T_m := -\infty$
- Wenn Verbindung  $i$  an  $T$  abgearbeitet wird, setze  $T_m := \max\{T_m, i\}$

## Dijkstra's Algorithmus:

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

kann adaptiert werden durch

## Parallel Self-Pruning Connection-Setting:

- Verwalte globales Label  $T_m := -\infty$
- Wenn Verbindung  $i$  an  $T$  abgearbeitet wird, setze  $T_m := \max\{T_m, i\}$
- Prune alle Verbindungen  $j < T_m$  (an jedem Knoten)

## Dijkstra's Algorithmus:

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

kann adaptiert werden durch

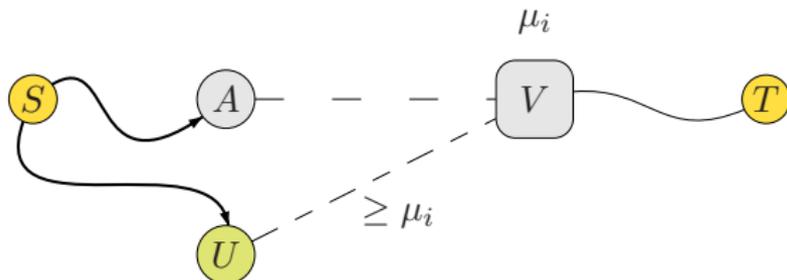
## Parallel Self-Pruning Connection-Setting:

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

# Pruning durch Distanztabellen (Idee)

## Vorbereitung:

- Wähle Teilmenge  $\mathcal{S}_{\text{trans}}$  von **Transfer-Stationen**
- Berechne komplette Distanztabelle zwischen allen  $S \in \mathcal{S}_{\text{trans}}$



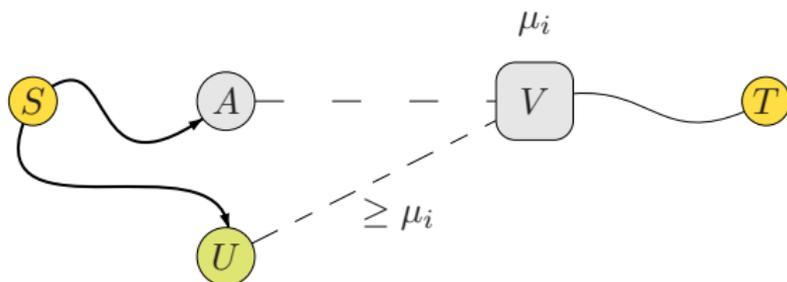
## Idee:

- Verwalte **vorl. Distanz**  $\mu_i$  zu Transferstationen  $V$  “nahe“ Ziel  $T$   
 $\mu_i$  ist **obere Schranke** bzgl. echtem Abstand zu  $V$

# Pruning durch Distanztabelle (Idee)

## Vorbereitung:

- Wähle Teilmenge  $\mathcal{S}_{\text{trans}}$  von **Transfer-Stationen**
- Berechne komplette Distanztabelle zwischen allen  $S \in \mathcal{S}_{\text{trans}}$



## Idee:

- Verwalte **vorl. Distanz**  $\mu_i$  zu Transferstationen  $V$  “nahe“ Ziel  $T$   
 $\mu_i$  ist **obere Schranke** bzgl. echtem Abstand zu  $V$
- **Prune** Verbindung  $i$  an **beliebiger** Transferstation  $U$  wenn

$$\text{dist}_S(U) + \mathcal{D}[U, V] \geq \mu_i$$

## Netzwerk von **Los Angeles**:

- 15 581 Stationen,
- 1 046 580 elem. Verbindungen

## Zugnetz von **Europa**:

- 30 517 Stationen,
- 1 775 533 elem. Verbindungen



## Netzwerk von **Los Angeles**:

- 15 581 Stationen,
- 1 046 580 elem. Verbindungen

## Zugnetz von **Europa**:

- 30 517 Stationen,
- 1 775 533 elem. Verbindungen



Auswertung durch 1 000 Anfragen wobei Start- und Zielbahnhöfe gleichverteilt zufällig gewählt.

# One-to-All Anfragen

	$p$	Los Angeles				Europe			
		Settled 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	—
EQUICONN	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	—	—

	$p$	Los Angeles				Europe			
		Settled 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	—
EQUICONN	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 deutlich weniger Verbindungen als LC

	$p$	Los Angeles				Europe			
		Settled 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	—
EQUICONN	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 deutlich weniger Verbindungen als LC
- PSPCS skaliert sehr gut mit zunehmender Anzahl Cores

	$p$	Los Angeles				Europe			
		Settled 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	—
EQUICONN	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 deutlich weniger Verbindungen als LC
- PSPCS skaliert sehr gut mit zunehmender Anzahl Cores
- Einfache Partitionierung liefert bereits gute Ergebnisse

# Station-to-Station Anfragen

	Los Angeles				Europe			
	PREPRO		QUERY		PREPRO		QUERY	
	Time [m:s]	Space [MiB]	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Time [ms]	Spd Up
0.0%	—	—	188.2	1.0	—	—	412.4	1.0
5.0%	4:19	240.7	59.1	3.2	20:13	214.3	186.5	2.2
10.0%	8:07	832.2	59.0	3.2	39:05	794.4	151.2	2.7
20.0%	16:21	3006.0	57.7	3.3	75:35	2986.7	132.1	2.9
deg > 2	18:01	3263	51.2	3.7	—	—	—	—

	Los Angeles				Europe			
	PREPRO		QUERY		PREPRO		QUERY	
	Time [m:s]	Space [MiB]	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Time [ms]	Spd Up
0.0%	—	—	188.2	1.0	—	—	412.4	1.0
5.0%	4:19	240.7	59.1	3.2	20:13	214.3	186.5	2.2
10.0%	8:07	832.2	59.0	3.2	39:05	794.4	151.2	2.7
20.0%	16:21	3006.0	57.7	3.3	75:35	2986.7	132.1	2.9
deg > 2	18:01	3263	51.2	3.7	—	—	—	—

- Speed-Up durch Distanztabelle bis 3.7

	Los Angeles				Europe			
	PREPRO		QUERY		PREPRO		QUERY	
	Time [m:s]	Space [MiB]	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Time [ms]	Spd Up
0.0%	—	—	188.2	1.0	—	—	412.4	1.0
5.0%	4:19	240.7	59.1	3.2	20:13	214.3	186.5	2.2
10.0%	8:07	832.2	59.0	3.2	39:05	794.4	151.2	2.7
20.0%	16:21	3006.0	57.7	3.3	75:35	2986.7	132.1	2.9
deg > 2	18:01	3263	51.2	3.7	—	—	—	—

- Speed-Up durch Distanztabelle bis 3.7
- 10 % Transferstationen sind guter Kompromiss

-  Reinhard Bauer, Daniel Delling, and Dorothea Wagner.  
Experimental Study on Speed-Up Techniques for Timetable Information Systems.  
*Networks*, 57(1):38–52, January 2011.
-  Daniel Delling.  
Time-Dependent SHARC-Routing.  
*Algorithmica*, 60(1):60–94, May 2011.  
Special Issue: European Symposium on Algorithms 2008.
-  Daniel Delling, Bastian Katz, and Thomas Pajor.  
Parallel Computation of Best Connections in Public Transportation Networks.  
*ACM Journal of Experimental Algorithmics*, 2012.  
To appear.
-  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.



Robert Geisberger.

Contraction of Timetable Networks with Realistic Transfers.

In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, May 2010.



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.

## Montag, 1.7.2013 (!)

Montag, 8.7.2013  
Mittwoch, 10.7.2013