

Algorithmen für Routenplanung

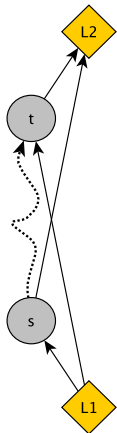
Letzte Sitzung, Sommersemester 2010

Thomas Pajor | 09. Juli 2010

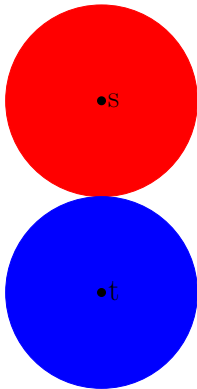
INSTITUT FÜR THEORETISCHE INFORMATIK · ALGORITHMIK I · PROF. DR. DOROTHEA WAGNER



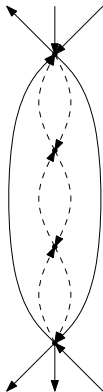
Landmarken



Bidirektionale Suche



Kontraktion



Arc-Flags

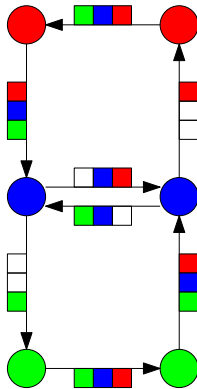
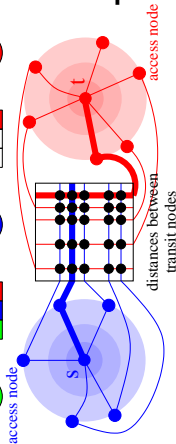


Table-Lookups



Vorbereitung:

- benutze gleiche Knotenordnung
- kontrahiere zeitabhängig
- erzeugt Suchgraphen $G' = (V, \uparrow E \cup \downarrow E)$

Vorbereitung:

- benutze gleiche Knotenordnung
- kontrahiere zeitabhängig
- erzeugt Suchgraphen $G' = (V, \uparrow E \cup \downarrow E)$

Anfrage

- Rückwärts aufwärts mittels min-max Suche
- markiere alle Kanten (u, v) aus $\downarrow E$ mit $\underline{d}(u, v) + \underline{d}(v, t) \leq \overline{d}(u, v)$
- diese Menge sei $\downarrow E'$
- zeitabhängige Vorwärtssuche in $(V, \uparrow E \cup \downarrow E')$

| input | type of ordering | Contr. | | | Queries | |
|----------|------------------|----------------|--------------|-------------|-----------|----------|
| | | ordering [h:m] | const. [h:m] | space [B/n] | time [ms] | speed up |
| Monday | static min | 0:05 | 0:20 | 1 035 | 1.19 | 1 240 |
| | timed | 1:47 | 0:14 | 750 | 1.19 | 1 244 |
| midweek | static min | 0:05 | 0:20 | 1 029 | 1.22 | 1 212 |
| | timed | 1:48 | 0:14 | 743 | 1.19 | 1 242 |
| Friday | static min | 0:05 | 0:16 | 856 | 1.11 | 1 381 |
| | timed | 1:30 | 0:12 | 620 | 1.13 | 1 362 |
| Saturday | static min | 0:05 | 0:08 | 391 | 0.81 | 1 763 |
| | timed | 0:52 | 0:08 | 282 | 1.09 | 1 313 |
| Sunday | static min | 0:05 | 0:06 | 248 | 0.71 | 1 980 |
| | timed | 0:38 | 0:07 | 177 | 1.07 | 1 321 |

Approximation

Problem:

(Lange) Shortcuts verbrauchen sehr viel Platz.

Idee:

- Speichere Approximationen der Funktionen an Shortcuts
- Upper-Bound-Funktion: $\text{len}^\uparrow(u, v)$ mit

$$\text{len}(u, v)(\tau) \leq \text{len}^\uparrow(u, v)(\tau) \leq (1 + \varepsilon) \text{len}(u, v)(\tau)$$

- Lower-Bound-Funktion implizit gegeben durch

$$\text{len}^\downarrow(u, v) : \tau \mapsto \text{len}^\uparrow(u, v) / (1 + \varepsilon)$$

- Für ursprüngliche Kanten behalte vollständige Funktionen

Approximationen der Funktionen benötigen deutlich weniger Interpolationspunkte!

Exakte Time-Queries

Problem: Finde *exakten* kürzesten s - t -Weg zu Zeit τ .

Problem: Finde *exakten* kürzesten s - t -Weg zu Zeit τ .

Phase 1

- Bidirektionale Intervallsuche mit Labels $\vec{d}^\uparrow[u]$, $\vec{d}^\downarrow[u]$, $\overleftarrow{d}^\uparrow[v]$, $\overleftarrow{d}^\downarrow[v]$ die nur aufsteigende Kanten relaxiert.
- Initialisiere Vorwärtssuche mit $\vec{d}^\uparrow[s] = \vec{d}^\downarrow[s] = \tau$.
- Initialisiere Rückwärtssuche mit $\overleftarrow{d}^\uparrow[t] = \overleftarrow{d}^\downarrow[t] = 0$.
- Sei $C \subset V$ Knoten an denen sich die Suchen treffen.

Problem: Finde *exakten* kürzesten s - t -Weg zu Zeit τ .

Phase 1

- Bidirektionale Intervallsuche mit Labels $\vec{d}^\uparrow[u]$, $\vec{d}^\downarrow[u]$, $\overleftarrow{d}^\uparrow[v]$, $\overleftarrow{d}^\downarrow[v]$ die nur aufsteigende Kanten relaxiert.
- Initialisiere Vorwärtssuche mit $\vec{d}^\uparrow[s] = \vec{d}^\downarrow[s] = \tau$.
- Initialisiere Rückwärtssuche mit $\overleftarrow{d}^\uparrow[t] = \overleftarrow{d}^\downarrow[t] = 0$.
- Sei $C \subset V$ Knoten an denen sich die Suchen treffen.

Phase 2

- Führe von allen $v \in C$ *Abwärtssuchen* zu t durch, die nur Kanten relaxiert, die in Phase 1 von der Rückwärtssuche besucht wurden.
- Führt zu Labels $\vec{d}_v^\uparrow[t]$ und $\vec{d}_v^\downarrow[t]$ für alle $v \in C$.

Problem: Finde *exakten* kürzesten s - t -Weg zu Zeit τ .

Phase 3

- Führe von t aus Rückwärts-Intervallsuche durch mit Labels

$$\max_{v \in C} \{ \vec{d}_v^\uparrow[t] \} \quad \text{und} \quad \min_{v \in C} \{ \vec{d}_v^\downarrow[t] \}$$

- Sei $C' \subset V$ Knoten an denen sich die Suche mit der Vorwärtssuche aus Phase 1 trifft.

Problem: Finde *exakten* kürzesten s - t -Weg zu Zeit τ .

Phase 3

- Führe von t aus Rückwärts-Intervallsuche durch mit Labels

$$\max_{v \in C} \{ \vec{d}_v^\uparrow[t] \} \quad \text{und} \quad \min_{v \in C} \{ \vec{d}_v^\downarrow[t] \}$$

- Sei $C' \subset V$ Knoten an denen sich die Suche mit der Vorwärtssuche aus Phase 1 trifft.

Phase 4

- Führe von allen $v \in C'$ Vorwärts- und Rückwärtsbreitensuche durch
- Relaxiere ausschließlich Kanten die in Phase 1 (Vorwärts-) und Phase 3 (Rückwärtssuche) betrachtet wurden
- Entpacke währenddessen Shortcuts. \rightsquigarrow Korridor C''

Problem: Finde *exakten* kürzesten *s-t*-Weg zu Zeit τ .

Phase 5

- Führe in C'' normalen zeitabhängigen Dijkstra mit Init.-Label $d[s] = \tau$ durch.

Problem: Finde *exakten* kürzesten *s-t*-Weg zu Zeit τ .

Phase 5

- Führe in C'' normalen zeitabhängigen Dijkstra mit Init.-Label $d[s] = \tau$ durch.

Ergebnisse:

| Algorithm | ϵ [%] | Prepro | | Query | |
|-----------------|-------------------|--------|------|-------|--------|
| | | [B/n] | GRO | [ms] | Spd-Up |
| Germany midweek | | | | | |
| TCH | – | 994 | 10.4 | 0.72 | 1 440 |
| ATCH | 0.1 | 308 | 3.2 | 1.10 | 942 |
| | 1.0 | 239 | 2.5 | 1.27 | 816 |
| | 10.0 | 163 | 1.7 | 2.40 | 432 |
| | ∞ | 118 | 1.2 | 1.45 | 714 |

Inexact TCHs:

- Approximiere $\text{len}(u, v)$ für jede Kante durch $\text{len}^\uparrow(u, v)$ mit

$$\frac{1}{1 + \varepsilon} \text{len}(u, v)(\tau) \leq \text{len}^\uparrow(u, v)(\tau) \leq (1 + \varepsilon) \text{len}(u, v)(\tau)$$

- Benutze TCH-Query mit Approximation der Funktionen
- Schneller als exakte ATCH ($\approx 2-3$ Mal)
- Relativer Fehler $< 10\%$

| Algorithm | ϵ [%] | prepro. [h:m] | ovh. [B/n] | time-q spd-up | profile-q spd-up | err. [%] |
|-----------------|-------------------|------------------|---------------|------------------|---------------------|-------------|
| Germany midweek | | | | | | |
| CALT | – | 0:09 | 50 | 280 | – | 0.00 |
| SH | – | 1:16 | 155 | 60 | 0.02 | 0.00 |
| heu SH | – | 3:26 | 137 | 2 164 | 1.40 | 0.61 |
| spc-eff heu SH | – | 3:48 | 14 | 491 | – | 0.61 |
| TCH | – | 0:37 | 899 | 1 440 | 11.66 | 0.00 |
| ATCH | 1 | 0:37 | 144 | 816 | 31.65 | 0.00 |
| ATCH | ∞ | 0:37 | 23 | 714 | 13.49 | 0.00 |
| inex TCH | 1 | 0:37 | 119 | 1 440 | 352.59 | 1.03 |
| inex TCH | 10 | 0:37 | 18 | 1 006 | 417.99 | 9.75 |

Bemerkung: Profile-Query Speed-Up bezieht sich auf normale Time-Dijkstra-Query

Gegeben (Fahrplan):

- Menge \mathcal{B} von Bahnhöfen,
- Menge \mathcal{Z} von Zügen
- Menge \mathcal{C} von elementaren Verbindungen
- Mindestumstiegszeiten transfer : $\mathcal{B} \rightarrow \mathbb{N}$.

Elementare Verbindung: Tupel bestehend aus

- Zug $Z \in \mathcal{Z}$
- Abfahrtsbahnhof $S_{\text{dep}} \in \mathcal{B}$
- Zielbahnhof $S_{\text{arr}} \in \mathcal{B}$
- Abfahrtszeit $\tau_{\text{dep}} \in \Pi$
- Ankunftszeit $\tau_{\text{arr}} \in \Pi$

Interpretation: Zug Z fährt von S_{dep} nach S_{arr} ohne Zwischenhalt von $\tau_{\text{dep}} - \tau_{\text{arr}}$ Uhr.

Beispiel für einen Fahrplan:

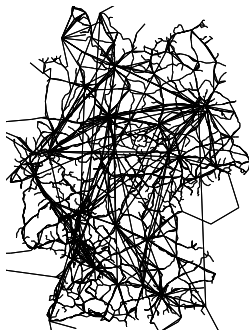
| | | | |
|---------------------------|-----------------|--------|--------|
| (IR 2269, Karlsruhe Hbf, | Pforzheim Hbf, | 10:05, | 10:23) |
| (IR 2269, Pforzheim Hbf, | Mühlacker, | 10:25, | 10:33) |
| (IR 2269, Mühlacker, | Vaihingen(Enz), | 10:34, | 10:40) |
| (IR 2269, Vaihingen(Enz), | Stuttgart Hbf, | 10:41, | 10:57) |
| ... | | | |
| (ICE 791, Stuttgart Hbf, | Ulm Hbf, | 11:12, | 12:06) |
| (ICE 791, Ulm Hbf, | Augsburg Hbf, | 12:08, | 12:47) |
| (ICE 791, Augsburg Hbf, | München Hbf, | 12:49, | 13:21) |

Ziel:

- Modellierung des Fahrplans als Graphen
- Fahrplanauskunft durch kürzeste-Wege Suche

Modellierung:

- Für jeden Bahnhof $S \in \mathcal{B}$: Ein Knoten
- Kante (S_i, S_j) gdw. ex. elementare Verbindung von S_i nach S_j
- Kantengewichte: Lower-Bound der Reisezeit zw. Bahnhöfen



Nachteile:

- Unrealistische Anfragen (keine Zeitabhängigkeit)
- Mit Zeitabhängigkeit: Unrealistische Umstiege

1. Zeitexpandiert

- Zeitabhängigkeiten ausrollen
- Knoten entsprechen Ereignissen im Fahrplan
- Kanten verbinden Ereignisse miteinander
 - Zugfahrt eines Zuges,
 - Umstieg zwischen Zügen,
 - Warten
- Großer Graph (viele Knoten und Kanten)
- + Einfacher Anfragealgorithmus (Dijkstra)

1. Zeitexpandiert

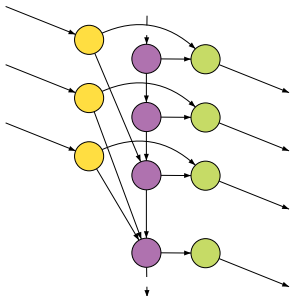
- Zeitabhängigkeiten ausrollen
- Knoten entsprechen Ereignissen im Fahrplan
- Kanten verbinden Ereignisse miteinander
 - Zugfahrt eines Zuges,
 - Umstieg zwischen Zügen,
 - Warten
- Großer Graph (viele Knoten und Kanten)
- + Einfacher Anfragealgorithmus (Dijkstra)

2. Zeitabhängig

- Zeitabhängigkeit an den Kanten
- Knoten entsprechen Bahnhöfen
- Kante \leftrightarrow Zug verbindet Bahnhöfe
 - Transferzeiten?
- + Kleiner Graph
- Zeitabhängige Routenplanung

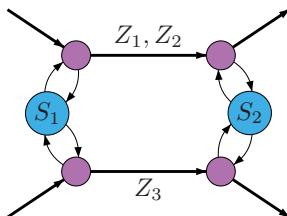
Modellierung: Zwei Ansätze

1. Zeitexpandiert



- Arrival-, Transfer- und Departure-Ereignisse
- Für jeden Zug
- Kantengewicht = Zeitdifferenz

2. Zeitabhängig



- Partitioniere Züge in Routen
- Pro Bahnhof: Stationsknoten
- Pro Route: Route-Knoten
- Routenkanten: Mehrere Züge
- Stationskanten: Transferzeit

Zeit-Anfragen

Gegeben: Startbahnhof S , Zielbahnhof T und Abfahrtszeit τ_S

Gegeben: Startbahnhof S , Zielbahnhof T und Abfahrtszeit τ_S

1. Zeitexpandiert

Startknoten:

- *Erstes* Transferevent in S mit Zeit $\tau \geq \tau_S$.

Zielknoten:

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

2. Zeitabhängig

Startknoten:

- Bahnhofsknoten S

Zielknoten:

- Bahnhofsknoten T

Anfrage:

- Time-Dependent Dijkstra mit Zeit τ_S
- Hier: Ankunftszeit im Voraus unbekannt

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

| 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

Jetzt: Erster Ansatz zur Beschleunigung von Profil-Anfragen

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

Wiederholung: Szenarien

- One-to-All Query (Vorberechnung) vs. One-to-One Query (Anfrage)
- Time-Query vs. Profile-Query

Zur Erinnerung:

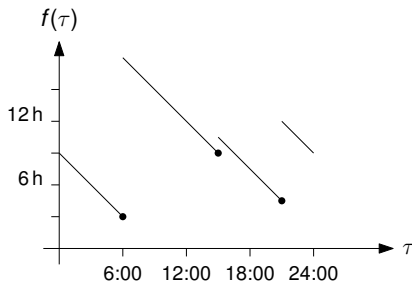
- Dijkstra's Algorithmus: One-to-All Query
- Stoppkriterium: One-to-One Query
- Grundlage für (fast) alle Beschleunigungstechniken
- Profilsuchen: Label-Correcting statt Label-Setting

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** (τ, w)
 $\tau \hat{=}$ Abfahrtszeit, $w \hat{=}$ 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 τ an S für **alle** $\tau \in \Pi$ und $v \in V$ ist.

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 τ 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 Factor 50)

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

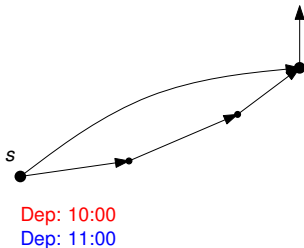
Für jede ausgehende Verbindung c_i an S :
Separate Zeitanfrage mit Abfahrtszeit $\tau_{\text{dep}}(c_i)$.

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

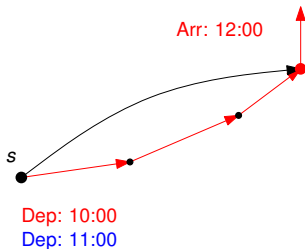
Beobachtung:

Verbindungen können sich **dominieren**.



Beobachtung:

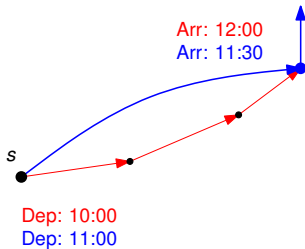
Verbindungen können sich **dominieren**.



Self-Pruning

Beobachtung:

Verbindungen können sich **dominieren**.

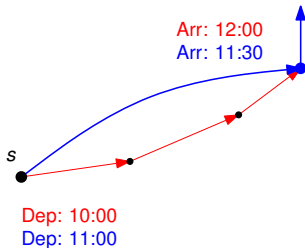


Beobachtung:

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)

Connection Reduction

Problem:

Dennoch: Nicht jede Verbindung ist 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 |

Connection Reduction

Problem:

Dennoch: Nicht jede Verbindung ist 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 |

Connection Reduction

Problem:

Dennoch: Nicht jede Verbindung ist 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:

Dennoch: Nicht jede Verbindung ist 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 |

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 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

Choice of Partitioning

Frage:

Gute Strategie zur Partitionierung der Verbindungen an S

Choice of Partitioning

Frage:

Gute Strategie zur Partitionierung der Verbindungen an S

- EQUICONN:
Gleiche **Anzahl** Verbindungen für jeden Thread

Choice of Partitioning

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.
Berechnungs-overhead durch Partitionierung

Inter-Thread-Pruning

Problem:

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

| | | | | | | | | | | | |
|------------------|------------------|------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|--------------------|--------------------|
| 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 |
|------------------|------------------|------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|--------------------|--------------------|

Inter-Thread-Pruning

Problem:

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

| | | | | | | | | | | | |
|------------------|------------------|------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|--------------------|--------------------|
| 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 |
|------------------|------------------|------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|--------------------|--------------------|

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 $\min_{j \in \text{Thread } l} \{arr(v, j)\} < arr(v, i)$.

Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten v und Thread k : Verwalte Label $\text{minarr}_k(v)$
Beschreibt Verbindungsindex mit minimaler Ankunftszeit an v
- 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 Verbindungsindex mit minimaler Ankunftszeit an v
- 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 $\min_{j \in \text{Thread } l} \{\text{arr}(v, j)\} < \text{arr}(v, i)$.

Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten v und Thread k : Verwalte Label $\text{minarr}_k(v)$
Beschreibt Verbindungsindex mit minimaler Ankunftszeit an v
- 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 $\text{minarr}_l(v) < \text{arr}(v, i)$.

Um Inter-Thread-Pruning zu ermöglichen:

- Pro Knoten v und Thread k : Verwalte Label $\text{minarr}_k(v)$
Beschreibt Verbindungsindex mit minimaler Ankunftszeit an v
- 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 $\text{minarr}_l(v) < \text{arr}(v, i)$.

- 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 Verbindungsindex mit minimaler Ankunftszeit an v
- 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 $\text{minarr}_l(v) < \text{arr}(v, i)$.

- 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 .

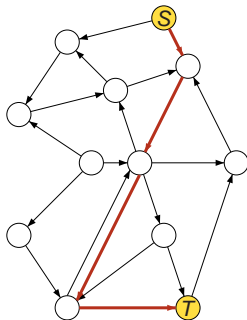
Ziel:

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 .

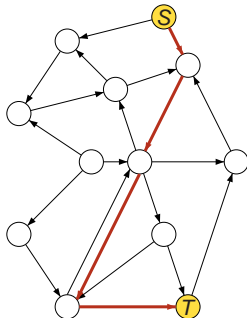
Ziel:

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) :-)

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

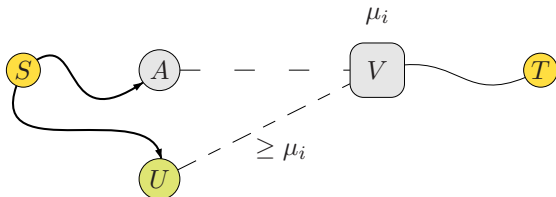
Parallel Self-Pruning Connection-Setting:

- 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 $S \in \mathcal{S}_{\text{trans}}$



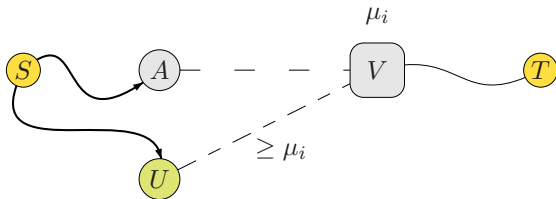
Idee:

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

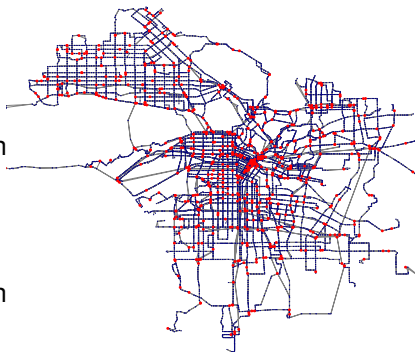
$$\text{dist}_S(U) + \mathcal{D}[U, V] \geq \mu_i$$

Netzwerk von **Los Angeles**:

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

Zugnetz von **Europa**:

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

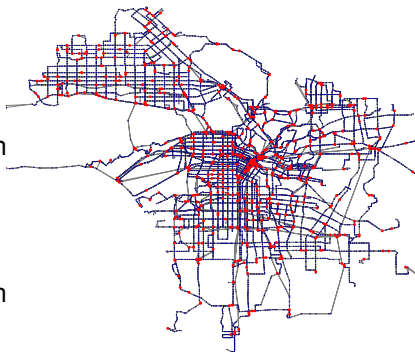


Netzwerk von **Los Angeles**:

- 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 |
| PSPCS: | 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 % |
| LC: | 1 | 18.9 M | 1482.1 | — | — | 17.4 M | 2497.1 | — | — |

One-to-All Anfragen

| | | 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 | — |
| 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 % |
| LC: | 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 |
| PSPCS: | 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 % |
| LC: | 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 |
| PSPCS: | 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 % |
| LC: | 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

Basismodule:

- 0 Bidirektionale Suche
- + Landmarken
- + Kontraktion
- + Arc-Flags
- Table Lookups

Somit sind folgende Algorithmen gut in zeitabhängigen Szenarien verwendbar

- ALT
- Core-ALT
- SHARC
- Contraction Hierarchies

Fahrplanauskunft

- Zeitexpandierte vs. zeitabhängige Modellierung
- Beschleunigungstechniken auf Straßennetzwerken schlecht adaptierbar
- Profilsuchen sinnvoller als Zeitanfragen
- PSPCS: Ersetzt Label-Correcting Algorithmus für Profil-Suchen
 - Connection-Setting Eigenschaft
 - Self-Pruning von dominierten Verbindungen
 - Gut Parallelisierbar

Offenes Forschungsgebiet!

Literatur (Contraction Hierarchies):

- **Gernot Veit Batz, Daniel Delling, Peter Sanders, Christian Vetter**
Time-Dependent Contraction Hierarchies
In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, April 2009.
- **Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, Peter Sanders**
Time-Dependent Contraction Hierarchies and Approximation
In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, May 2010.

Username: `routePlanning`

Password: `ss10`

Literatur (Fahrplanauskunft):

- **Evangelia Pyrga, Frank Schulz, Dorothea Wagner, Christos Zaroliagis**
Efficient Models for Timetable Information in Public Transportation Systems In: *ACM Journal of Experimental Algorithmics*, 2007
- **Daniel Delling, Bastian Katz, Thomas Pajor**
Parallel Computation of Best Connections in Public Transportation Networks In: *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society, 2010

Username: `routePlanning`

Password: `ss10`