Karlsruher Institut für Technologie

Algorithmen für Routenplanung

19. Vorlesung, Sommersemester 2018

Tobias Zündorf | 11. Juli 2018

Institut für Theoretische Informatik · Algorithmik · Prof. Dr. Dorothea Wagner

Bisher

Graph-basiert:

- Dijkstras Algorithmus auf zeitexpandiertem Graph
- Dijkstras Algorithmus auf zeitabhängigem Graph

Bisher

Graph-basiert:

- Dijkstras Algorithmus auf zeitexpandiertem Graph
- Dijkstras Algorithmus auf zeitabhängigem Graph

Näher an den Eingabedaten:

- RAPTOR

  Aber: Daten müssen pro Route zusammengefasst werden

Bisher

Graph-basiert:

- Dijkstras Algorithmus auf zeitexpandiertem Graph
- Dijkstras Algorithmus auf zeitabhängigem Graph

Näher an den Eingabedaten:

- RAPTOR

  Aber: Daten müssen pro Route zusammengefasst werden

Frage:

- Wie nahe können wir an den Eingabedaten bleiben?

Wiederholung: Connection

Zur Erinnerung:

- Eine *Connection* ist ein 5-tupel aus:
  - Abfahrtsstop: $s_{\text{dep}}(c)$
  - Zielstop: $s_{\text{arr}}(c)$
  - Abfahrtszeit: $\tau_{\text{dep}}(c)$
  - Ankunftszeit: $\tau_{\text{arr}}(c)$
  - Trip: $\text{trip}(c)$

Connection Scan

Idee:

- Vorberechung: Sortiere Connections nach Abfahrtszeit
- Anfrage: Relaxiere alle Connections in dieser Reihe

Connection Scan

Idee:

- Vorberechung: Sortiere Connections nach Abfahrtszeit
- Anfrage: Relaxiere alle Connections in dieser Reihe

Datenstruktur

- Tentative Ankunftszeiten d[x] pro Stop
- Erreichbarkeitsbit r[x] pro Trip

Connection Scan

Idee:

- Vorberechung: Sortiere Connections nach Abfahrtszeit
- Anfrage: Relaxiere alle Connections in dieser Reihe

Datenstruktur

- Tentative Ankunftszeiten d[x] pro Stop
- Erreichbarkeitsbit r[x] pro Trip

Connection-Relaxierung

- Teste ob Connection c erreichbar ist, d.h., teste ob man
- Wenn c erreichbar ist, dann

Connection Scan

Idee:

- Vorberechung: Sortiere Connections nach Abfahrtszeit
- Anfrage: Relaxiere alle Connections in dieser Reihe

Datenstruktur

- Tentative Ankunftszeiten d[x] pro Stop
- Erreichbarkeitsbit r[x] pro Trip

Connection-Relaxierung

- Teste ob Connection c erreichbar ist, d.h., teste ob man
  - einsteigen kann: $\text{d}[s_{\text{dep}}(c)] \leq \tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c))$
  - bereits im Trip sitzt: $\text{r}[\text{trip}(c)] = \text{true}$
- Wenn c erreichbar ist, dann

Connection Scan

Idee:

- Vorberechung: Sortiere Connections nach Abfahrtszeit
- Anfrage: Relaxiere alle Connections in dieser Reihe

Datenstruktur

- Tentative Ankunftszeiten d[x] pro Stop
- Erreichbarkeitsbit r[x] pro Trip

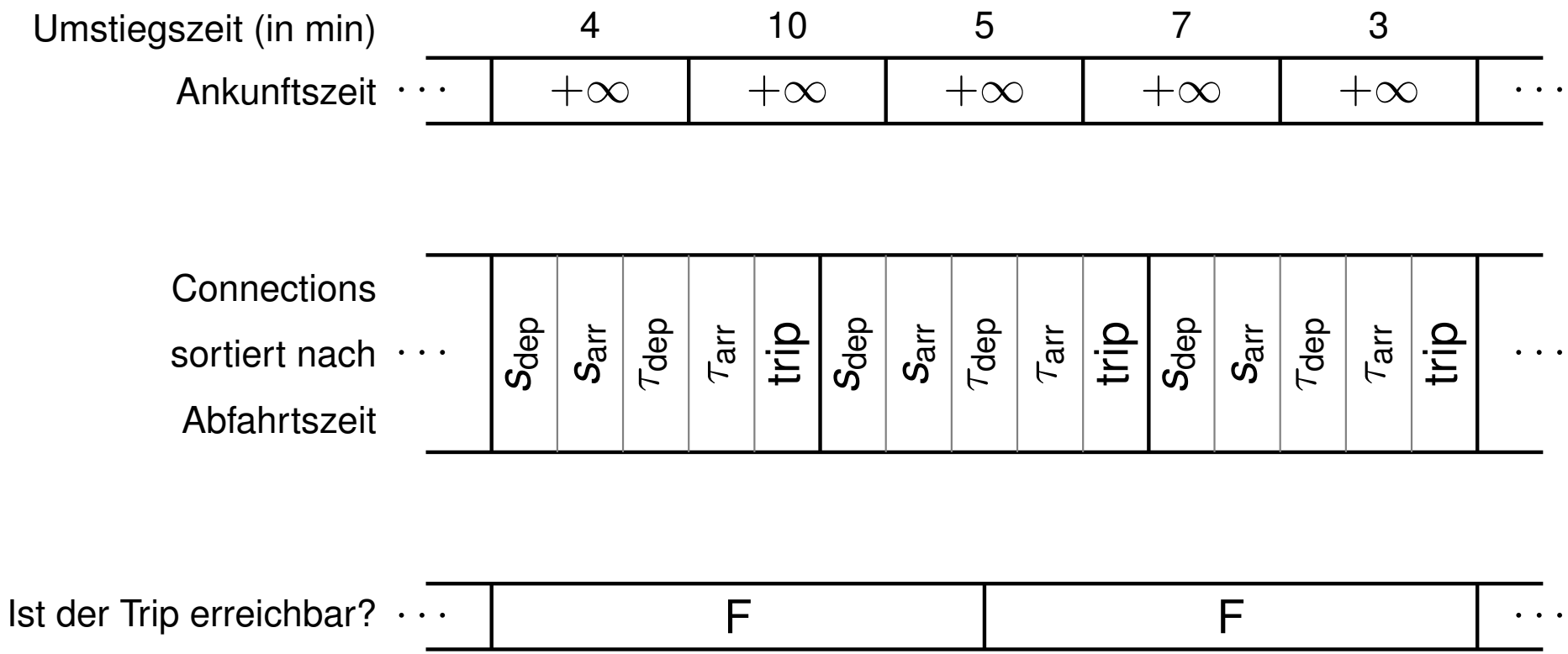
Connection-Relaxierung

- Teste ob Connection c erreichbar ist, d.h., teste ob man
  - einsteigen kann: $\text{d}[s_{\text{dep}}(c)] \leq \tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c))$
  - bereits im Trip sitzt: $\text{r}[\text{trip}(c)] = \text{true}$
- Wenn c erreichbar ist, dann
  - könnte man sitzen bleiben: $\text{r}[\text{trip}(c)] \leftarrow \text{true}$
  - könnte man aussteigen: $\text{d}[s_{\text{arr}}(c)] \leftarrow \min\{\text{d}[s_{\text{arr}}(c)], \tau_{\text{arr}}(c)\}$

Connection Scan

Problem der Frühesten Ankunft

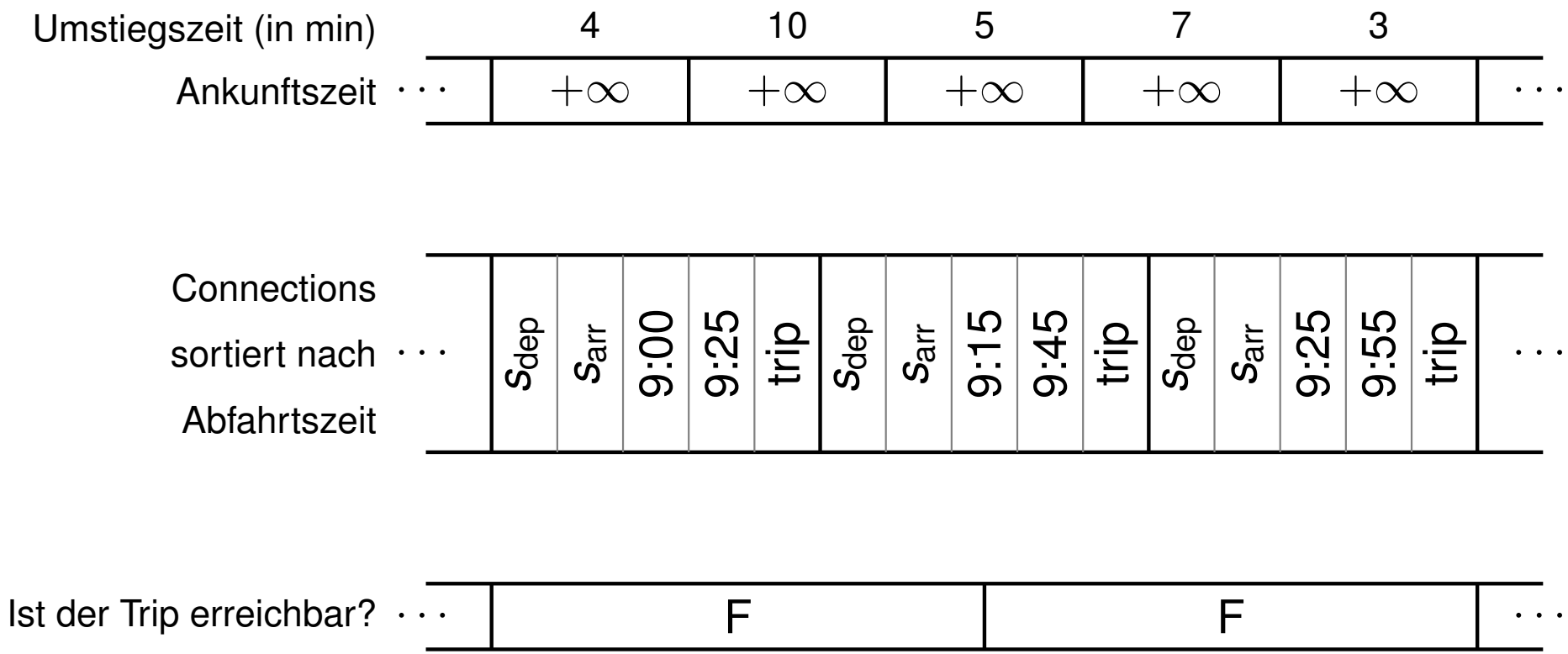
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

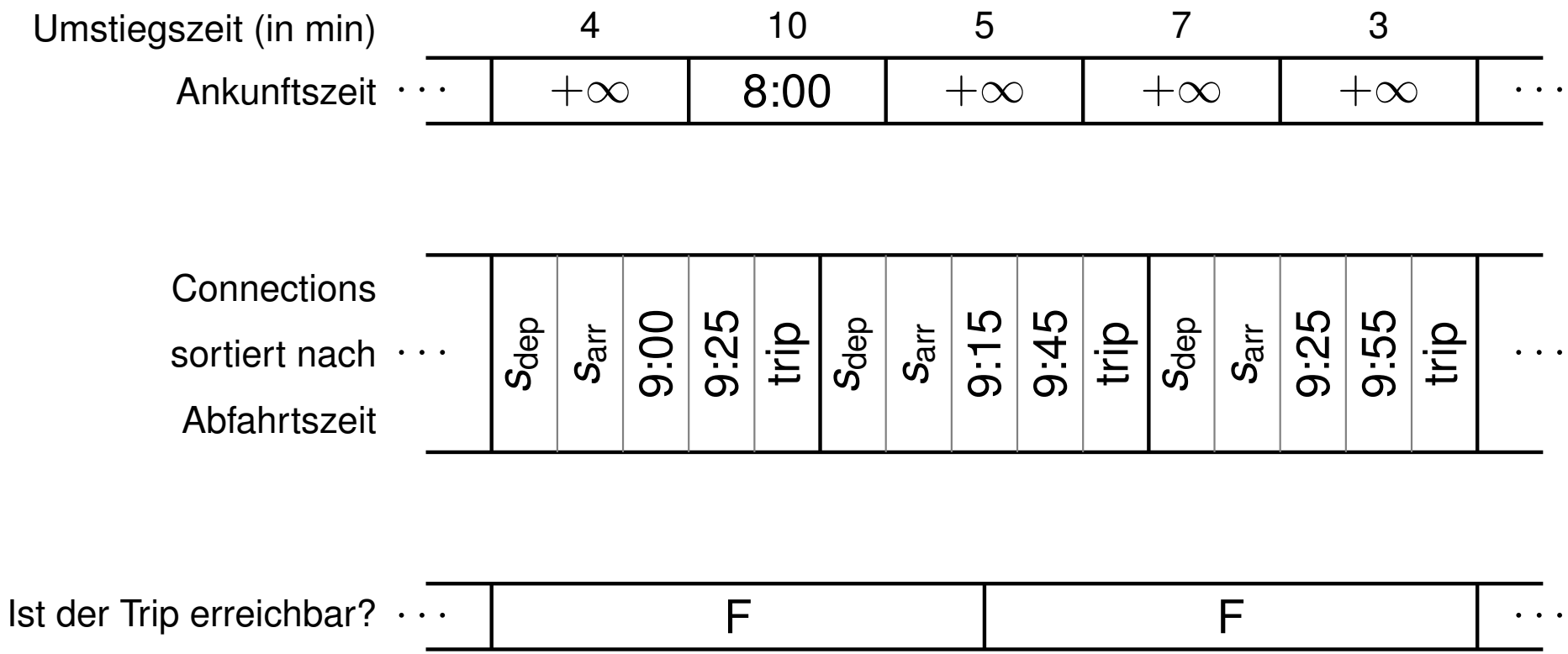
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

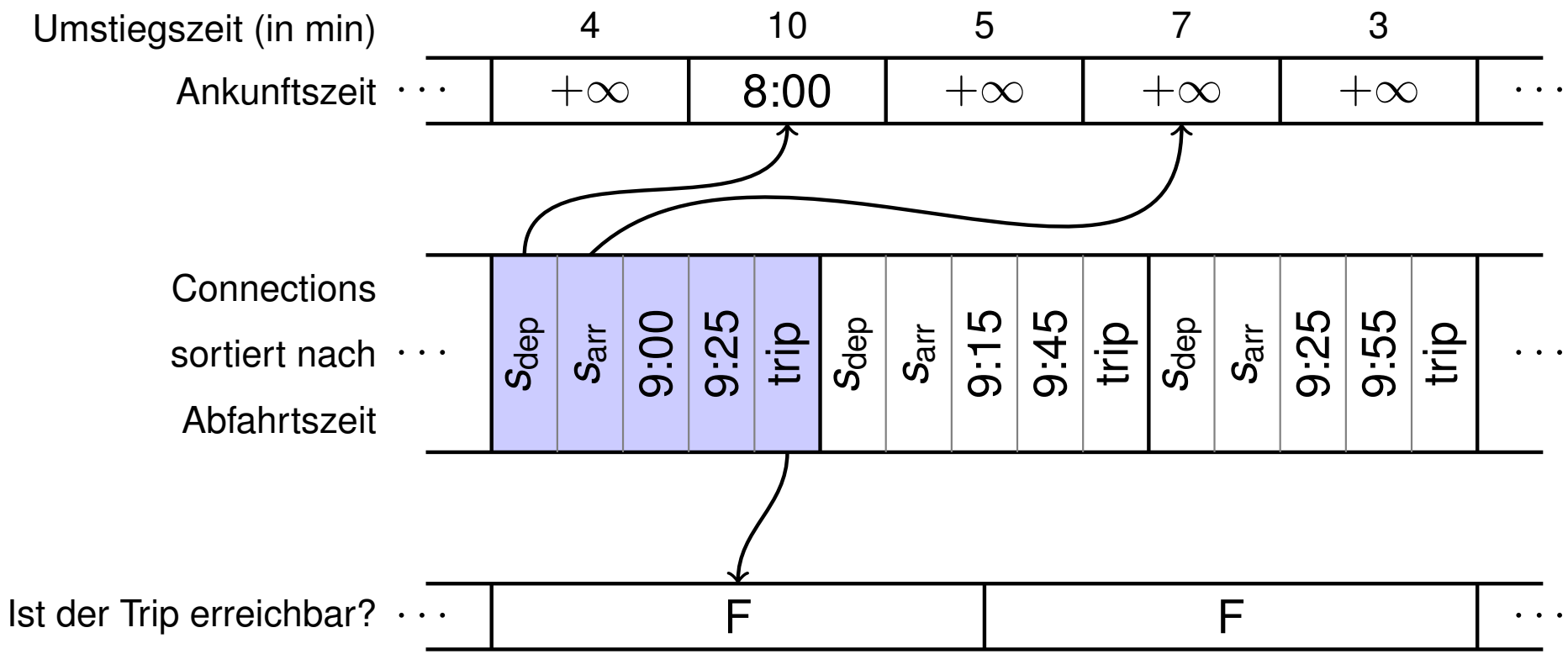
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

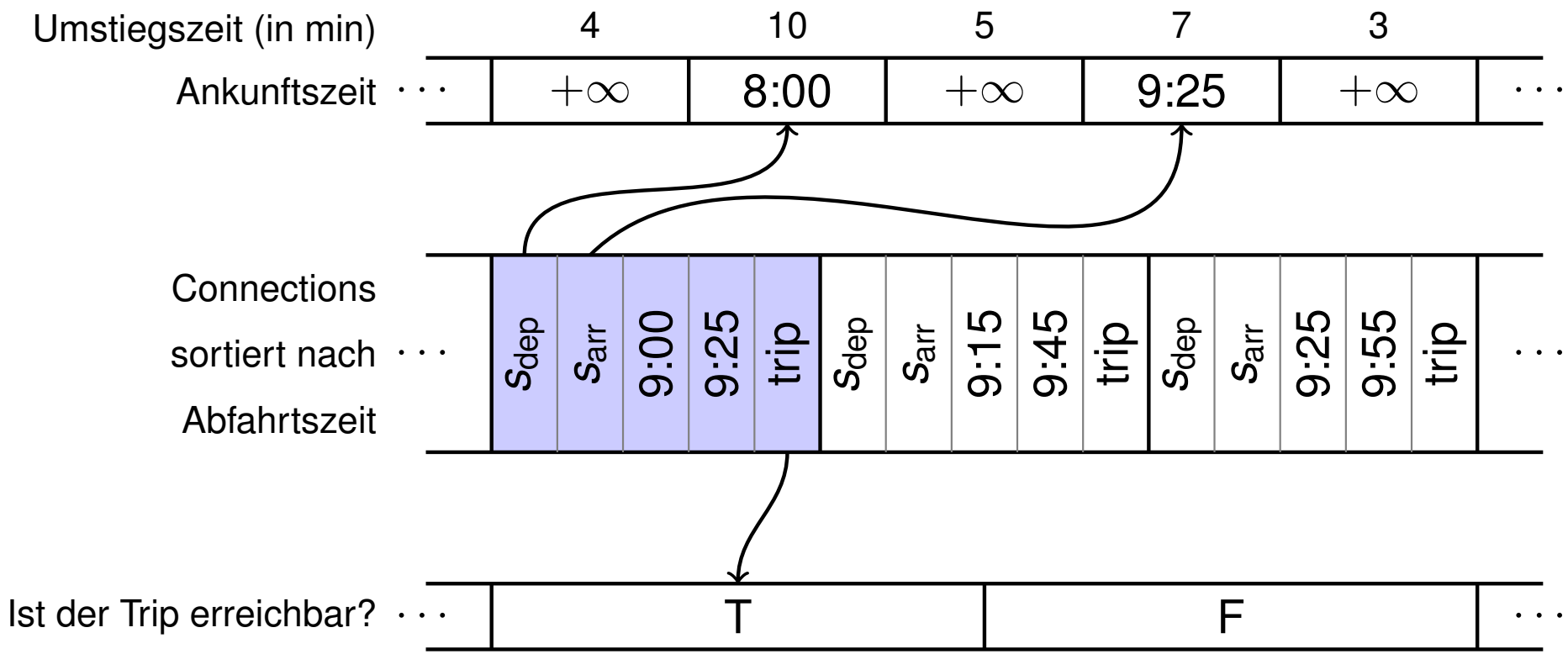
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

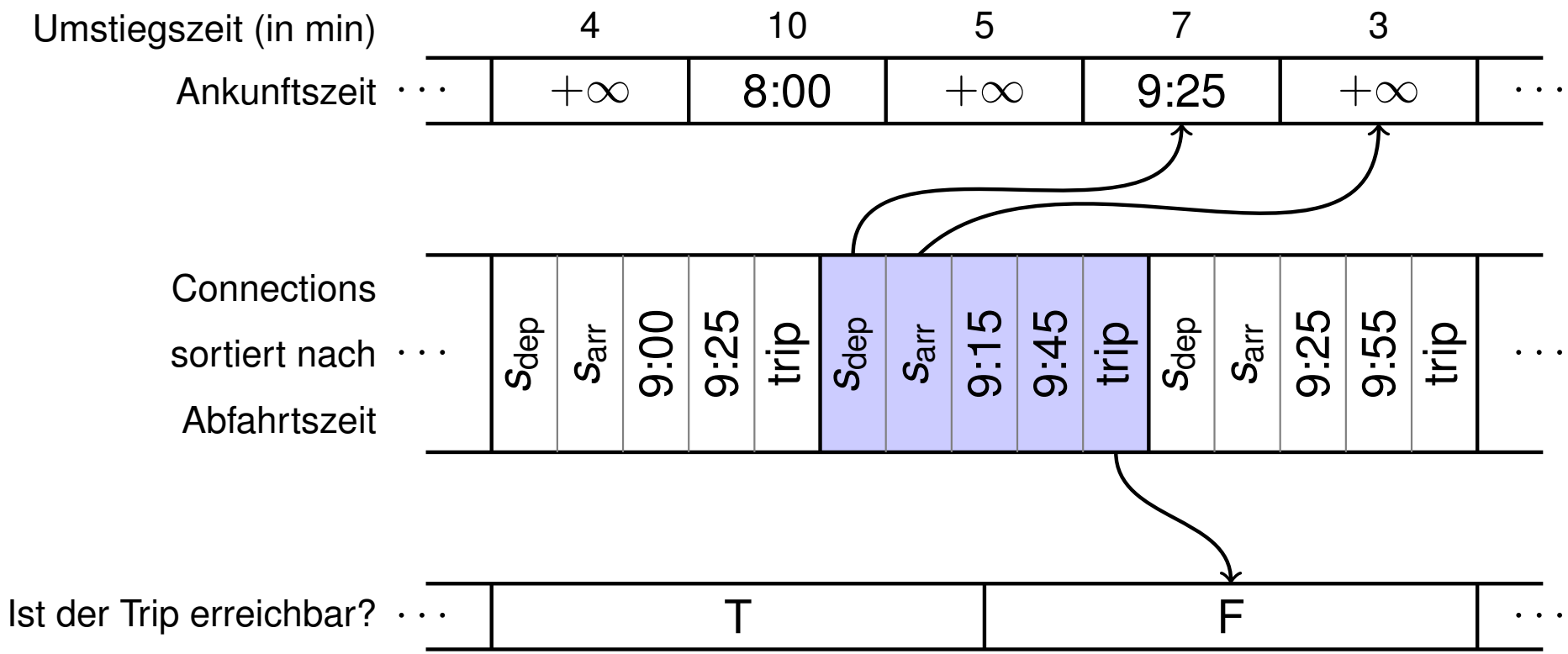
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

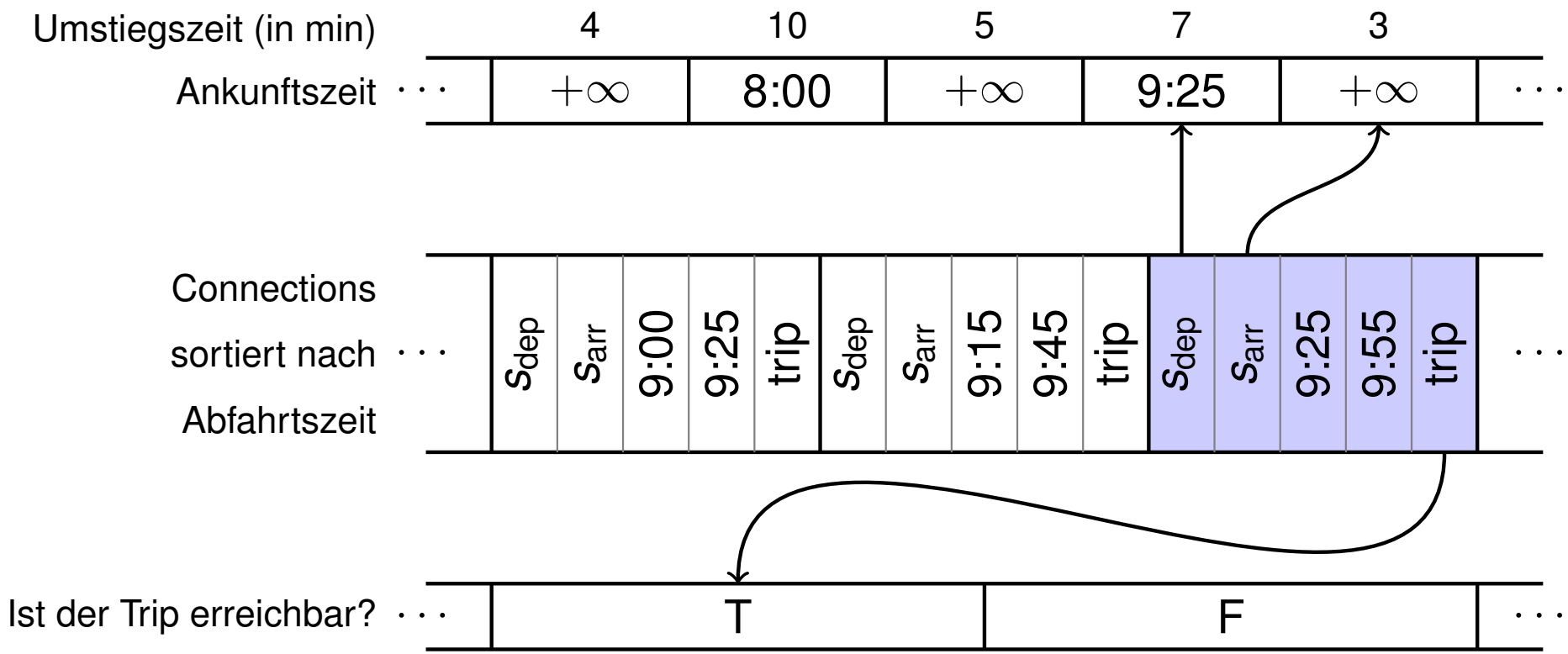
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

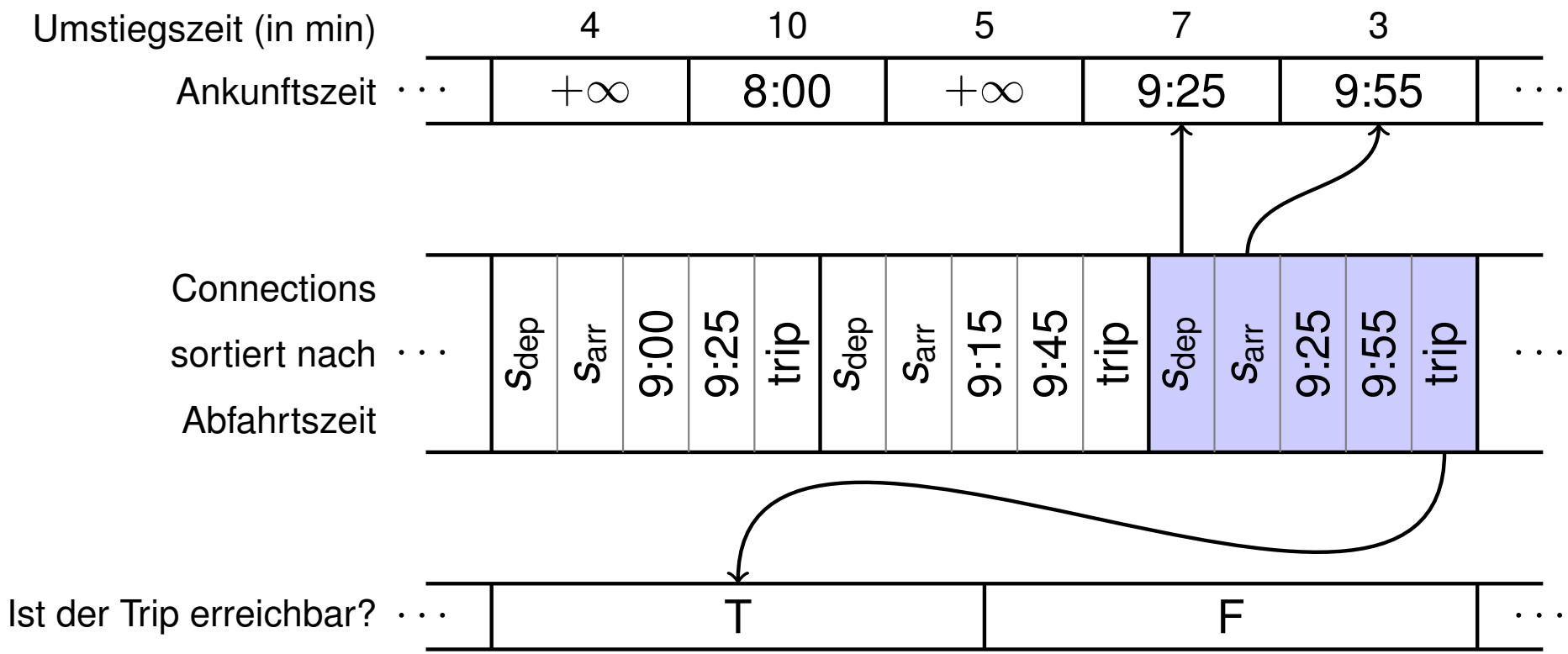
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

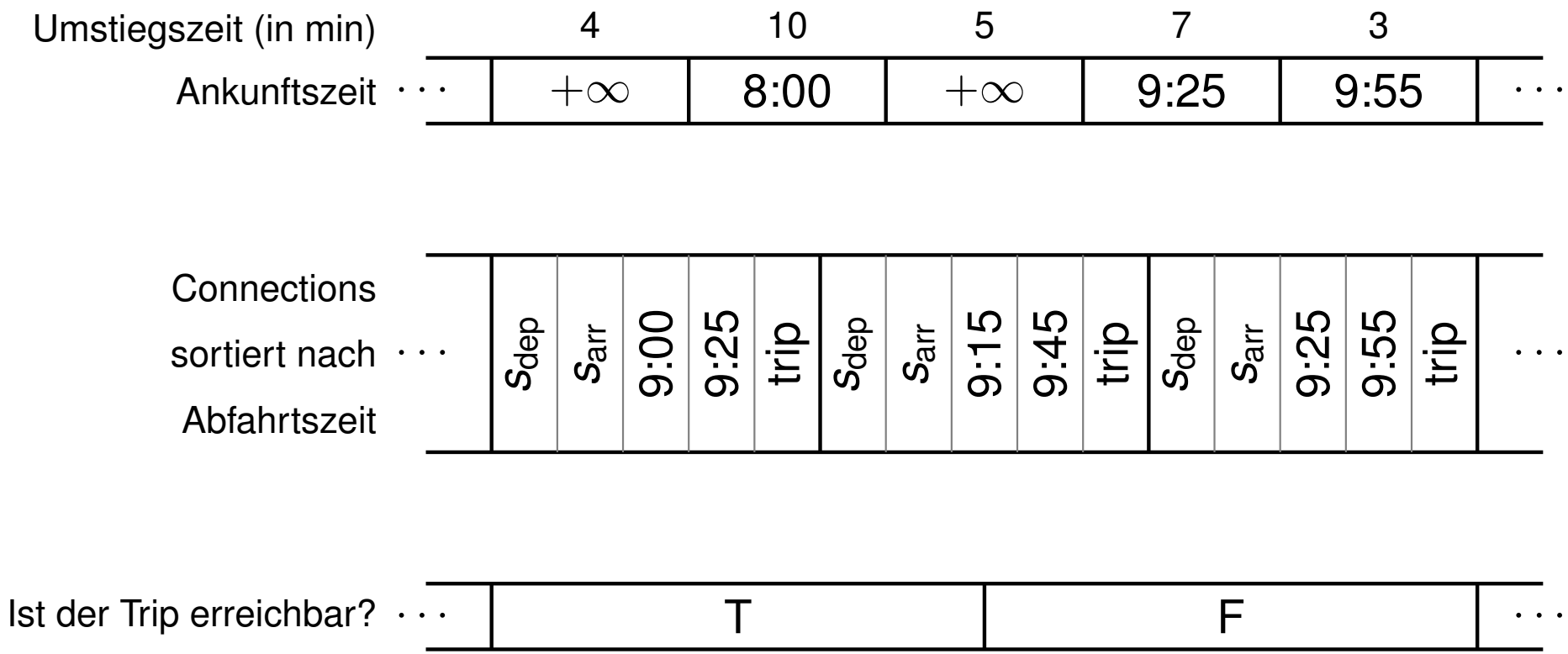
Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

Problem der Frühesten Ankunft

Eingabe: Sortiere Connectionliste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit



Connection Scan

$d[x] \leftarrow \infty$ für alle Stops x;
$d[s] \leftarrow \tau - \tau_{ch}(s)$;
$r[x] \leftarrow \text{false}$ für alle Trips x;

for *alle Connections c aufsteigend nach* $\tau_{dep}(c)$ **do**
  if $d[s_{dep}(c)] \leq \tau_{dep}(c) - \tau_{ch}(s_{dep}(c))$ ***or*** $r[\text{trip}(c)] = \text{true}$ **then**
    $r[\text{trip}(c)] \leftarrow \text{true}$;
    $d[s_{arr}(c)] \leftarrow \min\{d[s_{arr}(c)], \tau_{arr}(c)\}$;

Fußwege

Idee:

- Wenn man an Stop x aussteigt, dann relaxiert man alle von x ausgehenden Fußwege

Fußwege

Idee:

- Wenn man an Stop x aussteigt, dann relaxiert man alle von x ausgehenden Fußwege

Optimierung:

- Man braucht die Fußwege nur abzulaufen wenn $d[s_{arr}(c)]$ auch verbessert wird
- **Begründung**: Wenn $d[s_{arr}(c)]$ nicht verbessert wird, dann wurde bereits einen besseren Weg P nach $s_{arr}(c)$ gefunden. An P kann man diese Fußwege dran hängen und ist an allen per Fußweg erreichbaren Stops früher. Kein $d[x]$ wird also verbessert.
- **Achtung**: Die Optimierung benötigt, dass die Fußwege transitiv abgeschlossen sind und die Dreiecksungleichung erfüllen

Connection Scan

$\mathrm{d}[x] \leftarrow \infty$ für alle Stops x;
$\mathrm{d}[s] \leftarrow \tau - \tau_{\mathrm{ch}}(s)$;
$\mathrm{r}[x] \leftarrow \mathrm{false}$ für alle Trips x;
for *alle Fußwege* (s, x) *mit Länge* ℓ **do**
  $\mathrm{d}[x] \leftarrow \min\{\mathrm{d}[x], \tau + \ell\}$;

for *alle Connections* c *aufsteigend nach* $\tau_{\mathrm{dep}}(c)$ **do**
  if $\mathrm{d}[s_{\mathrm{dep}}(c)] \leq \tau_{\mathrm{dep}}(c) - \tau_{\mathrm{ch}}(s_{\mathrm{dep}}(c))$ ***or*** $\mathrm{r}[\mathrm{trip}(c)] = \mathrm{true}$ **then**
    $\mathrm{r}[\mathrm{trip}(c)] \leftarrow \mathrm{true}$;
    if $\mathrm{d}[s_{\mathrm{arr}}(c)] > \tau_{\mathrm{arr}}(c)$ **then**
      $\mathrm{d}[s_{\mathrm{arr}}(c)] \leftarrow \tau_{\mathrm{arr}}(c)$;
      for *alle Fußwege* $(s_{\mathrm{arr}}(c), x)$ *mit Länge* ℓ **do**
        $\mathrm{d}[x] \leftarrow \min\{\mathrm{d}[x], \tau_{\mathrm{arr}}(c) + \ell\}$;

Startkriterium

Beobachtung:

- Connections vor der Abfahrtszeit können nicht verwendet werden
- Per binärer Suche die erste connection bestimmen, die nicht vor der Startzeit abfährt und erst ab dieser scanen

Stopkriterium

Bisher: Wir lösen das one-to-all Problem

Frage: Geht es besser, wenn wir den Zielstop t kennen?

Stopkriterium

Bisher: Wir lösen das one-to-all Problem

Frage: Geht es besser, wenn wir den Zielstop t kennen?

Beobachtung: Züge die abfahren, nach der Ankunftszeit an t sind nie nützlich

$\Rightarrow$ Scan abbrechen, wenn die Zeit an t nicht mehr größer ist als die Abfahrtszeit der aktuell gescannten Connection

Profil Anfragen

Problem: Der Fahrgast kennt seine Abfahrts- und Ankunftszeit oft nicht.
Lösung: Journeys für eine Zeitspanne angeben.

Profil Anfragen

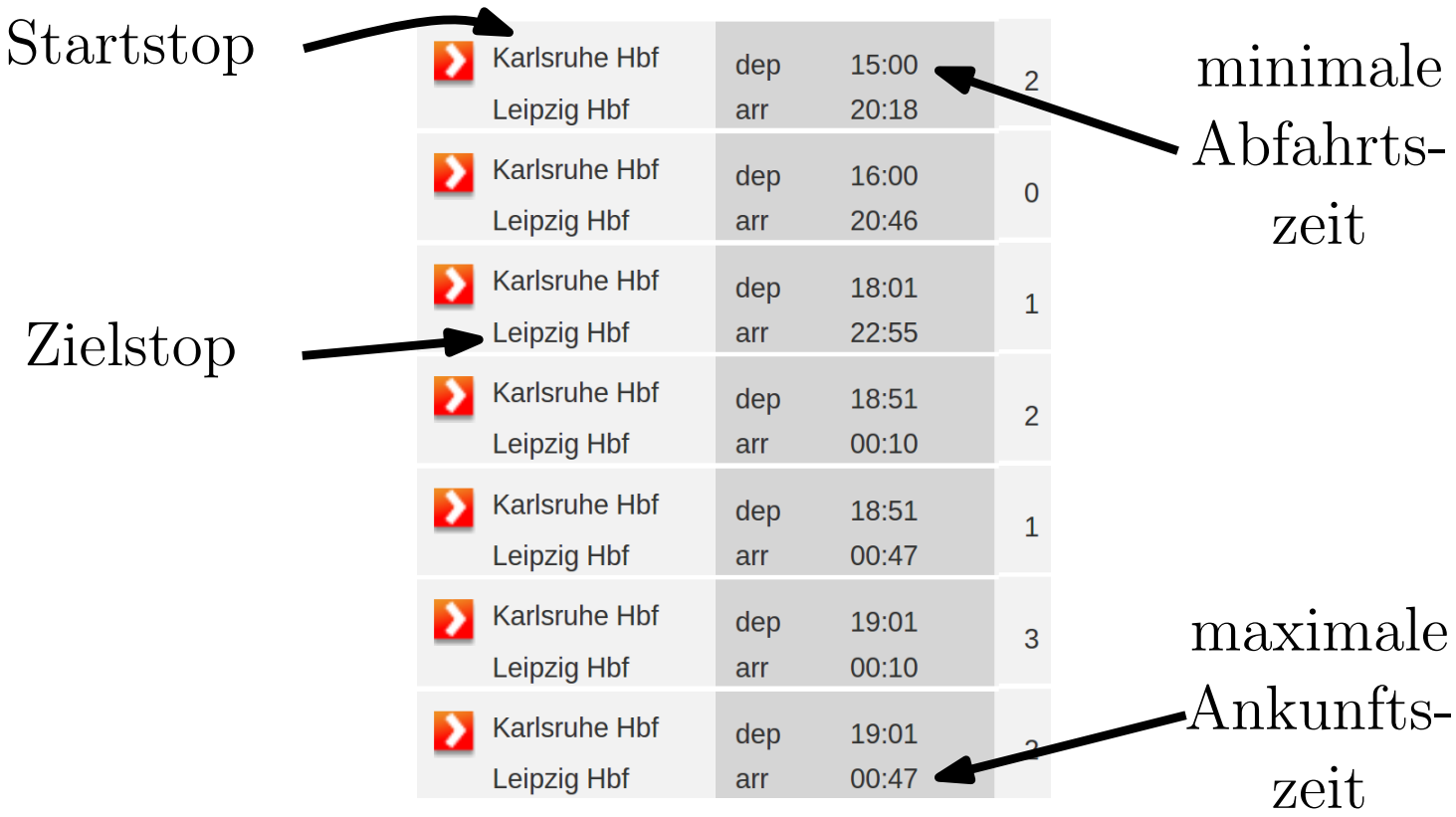
Problem: Der Fahrgast kennt seine Abfahrts- und Ankunftszeit oft nicht.
Lösung: Journeys für eine Zeitspanne angeben.

Karlsruhe Hbf	dep	15:00	2
Leipzig Hbf	arr	20:18	
Karlsruhe Hbf	dep	16:00	0
Leipzig Hbf	arr	20:46	
Karlsruhe Hbf	dep	18:01	1
Leipzig Hbf	arr	22:55	
Karlsruhe Hbf	dep	18:51	2
Leipzig Hbf	arr	00:10	
Karlsruhe Hbf	dep	18:51	1
Leipzig Hbf	arr	00:47	
Karlsruhe Hbf	dep	19:01	3
Leipzig Hbf	arr	00:10	
Karlsruhe Hbf	dep	19:01	2
Leipzig Hbf	arr	00:47	

Screenshot von bahn.de

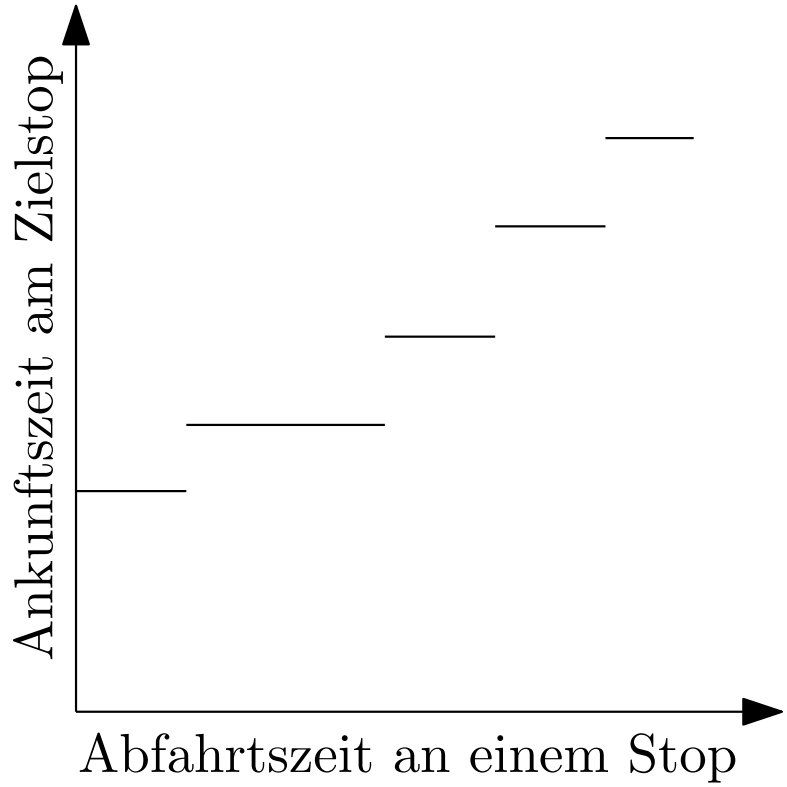
Profil Anfragen

Problem: Der Fahrgast kennt seine Abfahrts- und Ankunftszeit oft nicht.
Lösung: Journeys für eine Zeitspanne angeben.



Screenshot von bahn.de

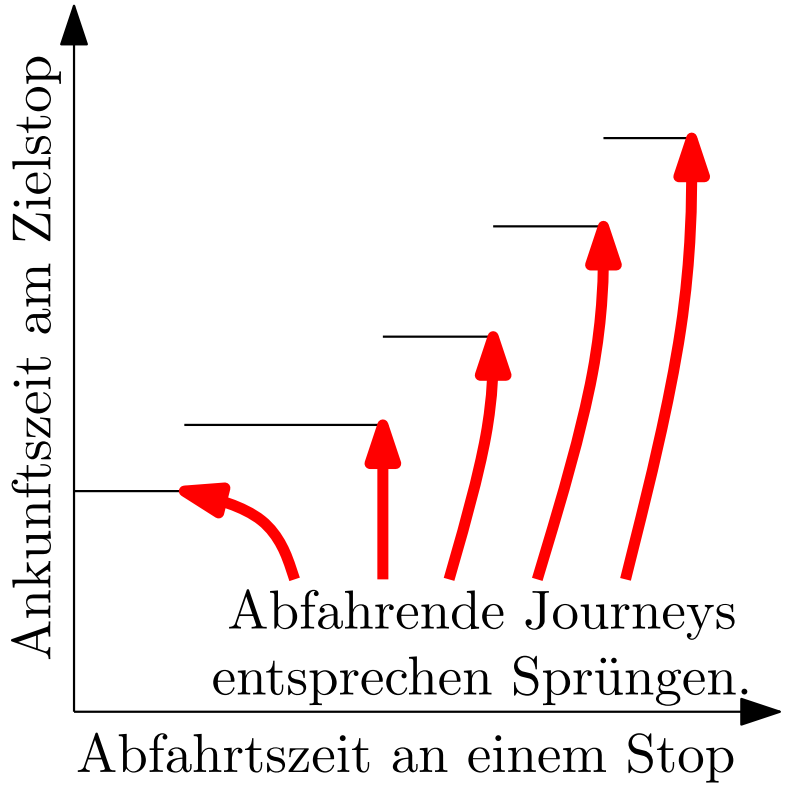
Profil Connection Scan



Problem der frühesten *Rückwärts*-Ankunftsprofile

Eingabe: Fahrplan, Zielstop t
Ausgabe: st-Profil für jeden Stop s (außer t)

Profil Connection Scan



Problem der frühesten *Rückwärts*-Ankunftsprofile

Eingabe: Fahrplan, Zielstop t
Ausgabe: st-Profil für jeden Stop s (außer t)

Grundgerüst

Initialisiere Profil an Stops;
Initialisiere optimale Ankunftszeit an Trips;

for *all Connections c absteigend nach* $\tau_{\text{dep}}(c)$ **do**

`// 1. Bestimme Ankunftszeit falls c benutzt wird`

$\tau_1 \leftarrow$ Ankunftszeit wenn man zum Ziel läuft;

$\tau_2 \leftarrow$ Ankunftszeit wenn Sitzenbleiben (Benötigt Profil vom Trip trip(c));

$\tau_3 \leftarrow$ Ankunftszeit wenn Umsteigen (Benötigt Profil vom Stop $s_{\text{arr}}(c)$);

`// τc Optimale Ankunftszeit wenn c benutzt wird`

$\tau_c \leftarrow \min\{\tau_1, \tau_2, \tau_3\}$;

`// 2. Passe die Stop und Trip Profile an`

Füge τ_c zum Profil von Stop $s_{\text{dep}}(c)$ hinzu;

Aktualisiere Ankunftszeit von trip(c) mit τ_c;

Grundgerüst

Vorgehen

- Initial gibt es keine Connection
- Connections werden absteigend nach Abfahrtszeit eingeführt
- Zum Zeitpunkt wo c eingeführt wird, gibt es keine frühere Connection

Grundgerüst

Vorgehen

- Initial gibt es keine Connection
- Connections werden absteigend nach Abfahrtszeit eingeführt
- Zum Zeitpunkt wo c eingeführt wird, gibt es keine frühere Connection

Idee

- Verwalte Profile bezüglich aller eingeführten Connections
- Initial gibt es keine Connections $\rightarrow$ triviale initial Lösung
- Wenn Connection c eingeführt wird, dann wird die Lösung angepasst

Grundgerüst

Vorgehen

- Initial gibt es keine Connection
- Connections werden absteigend nach Abfahrtszeit eingeführt
- Zum Zeitpunkt wo c eingeführt wird, gibt es keine frühere Connection

Idee

- Verwalte Profile bezüglich aller eingeführten Connections
- Initial gibt es keine Connections $\rightarrow$ triviale initial Lösung
- Wenn Connection c eingeführt wird, dann wird die Lösung angepasst

Beobachtung

- Wenn eine Connection c bearbeitet wird, gibt es keine frühere
  - Niemand der bereits unterwegs ist kann c verwenden
  - c kann nur am Anfang einer Journey vorkommen

Zum Ziel Laufen

Ohne Fußwege

- Man kann also “Laufen” wenn man am Ziel angekommen ist
- Aus

$\tau_1 \leftarrow$ Ankunftszeit wenn man zum Ziel läuft;

- wird

if $s_{\text{arr}}(c) = \text{target}$ **then**
  $\tau_1 \leftarrow \tau_{\text{arr}}(c)$;
else
  $\tau_1 \leftarrow \infty$;

Zum Ziel Laufen

Mit Fußwege

- Verwalte $L[x]$ Array das die Laufdistanz von x zu t enthält
- L wird in einem preprocessing Schritt berechnet

for *alle Stops* x **do**
  $L[x] \leftarrow \infty$;
for *alle Fußwege* (x, target) *mit Länge* ℓ **do**
  $L[x] \leftarrow \ell$;
$L[\text{target}] \leftarrow 0$;
...
$\tau_1 \leftarrow \tau_{\text{arr}}(c) + L[s_{\text{arr}}(c)]$;

Sitzenbleiben

Trip-Datenstruktur

- Verwalte in Array $T[x]$, mit einem Wert pro Trip x
- $T[x]$ ist die früheste Ankunftszeit, wenn man in x startet
- Initial gibt es keine Connections in x, $T[x] \leftarrow \infty$ für alle x

- Aus

  Initialisiere optimale Ankunftszeit an Trips;

- wird

  for *alle Trips x* **do**
  $T[x] \leftarrow \infty$;

Sitzenbleiben

Trip-Datenstruktur

- Verwalte in Array $T[x]$, mit einem Wert pro Trip x
- $T[x]$ ist die früheste Ankunftszeit, wenn man in x startet
- Initial gibt es keine Connections in x, $T[x] \leftarrow \infty$ für alle x

- Aus

  $\tau_2 \leftarrow$ Ankunftszeit wenn Sitzenbleiben;

- wird

  $\tau_2 \leftarrow T[\text{trip}(c)]$;

Sitzenbleiben

Trip-Datenstruktur

- Verwalte in Array $T[x]$, mit einem Wert pro Trip x
- $T[x]$ ist die früheste Ankunftszeit, wenn man in x startet
- Initial gibt es keine Connections in x, $T[x] \leftarrow \infty$ für alle x

- Aus

  Aktualisiere Ankunftszeit von trip(c) mit τ_c;

- wird

  $T[\text{trip}(c)] \leftarrow \tau_c$;

Umsteigen

Stop-Datenstruktur

- Verwalte Array $P[x]$ von Profilen
- $P[x]$ ist das Profil von Stop x zu target
  (Details zu Profilen auf nächster Folie)

- Aus

  Initialisiere Profil an Stops;

- wird

  for *alle Stops* x **do**
  $P[x] \leftarrow \{\forall \tau : \tau \mapsto \infty\}$;

Umsteigen

Stop-Datenstruktur

- Verwalte Array $P[x]$ von Profilen
- $P[x]$ ist das Profil von Stop x zu target

  (Details zu Profilen auf nächster Folie)

- Aus

  $\tau_3 \leftarrow$ Ankunftszeit wenn Umsteigen;

- wird

  $\tau_3 \leftarrow$ werte $P[s_{\text{arr}}(c)]$ zum Zeitpunkt $\tau_{\text{arr}}(c)$ aus;

Umsteigen

Stop-Datenstruktur

- Verwalte Array $P[x]$ von Profilen
- $P[x]$ ist das Profil von Stop x zu target
  (Details zu Profilen auf nächster Folie)

- Aus

  Füge τ_c zum Profil von Stop $s_{\text{dep}}(c)$ ein;

- wird

  Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)), \tau_c)$ in $P[s_{\text{dep}}(c)]$ ein;

Profile

- Speichere Ankunftszeitfunktionen P als Array von $(\tau_{\text{dep}}, \tau_{\text{arr}})$-Paaren
- Paare dominieren sich nicht
  - Es gibt keine zwei Paare (d_1, a_1) und (d_2, a_2) mit $d_1 < d_2$ und $a_1 > a_2$
- Array ist dynamisch und kann am Anfang wachsen
- Array ist sortiert, d.h., in $P[0]$ steht die früheste Abfahrt

Profile

- Jedes Array hat ein (∞, ∞)-Paar
- Damit ist die Ankunftszeitfunktion ∞ wenn alle Züge abgefahren sind
- (∞, ∞)-Paar wird bei der Initialisierung eingefügt
- Aus

for *alle Stops x* **do**
  $P[x] \leftarrow \{\forall \tau : \tau \mapsto \infty\}$;

- wird

for *alle Stops x* **do**
  $P[x] \leftarrow \{(\infty, \infty)\}$;

Profile

- Evaluierung kann mit binärer oder sequentiell durchgeführt werden
- Hier ist sequentiell besser (Begründung später)
- Aus

werte P zum Zeitpunkt τ aus;

- wird

for i *from* 0 *to* $\text{length}(P) - 1$ **do**
    $(d, a) \leftarrow P[i]$;
    if $\tau \leq d$ **then**
        Ergebnis ist a;
        breche Schleife ab;

Profile

- Erstmal ohne Fußwege (mit später)
- Alle bisherigen Connections fahren nicht vor $\tau_{\text{dep}}(c)$ ab
- Wenn $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)), \tau_c)$ in $P[s_{\text{dep}}(c)]$ eingefügt wird, dann an der Stelle $P[s_{\text{dep}}(c)][0]$
- Aus

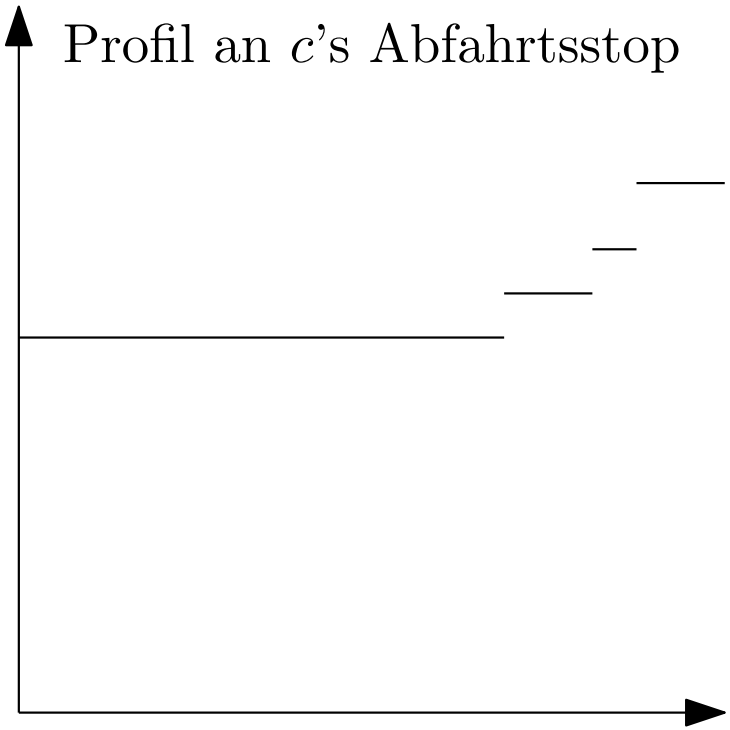
> Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)), \tau_c)$ in $P[s_{\text{dep}}(c)]$ ein;

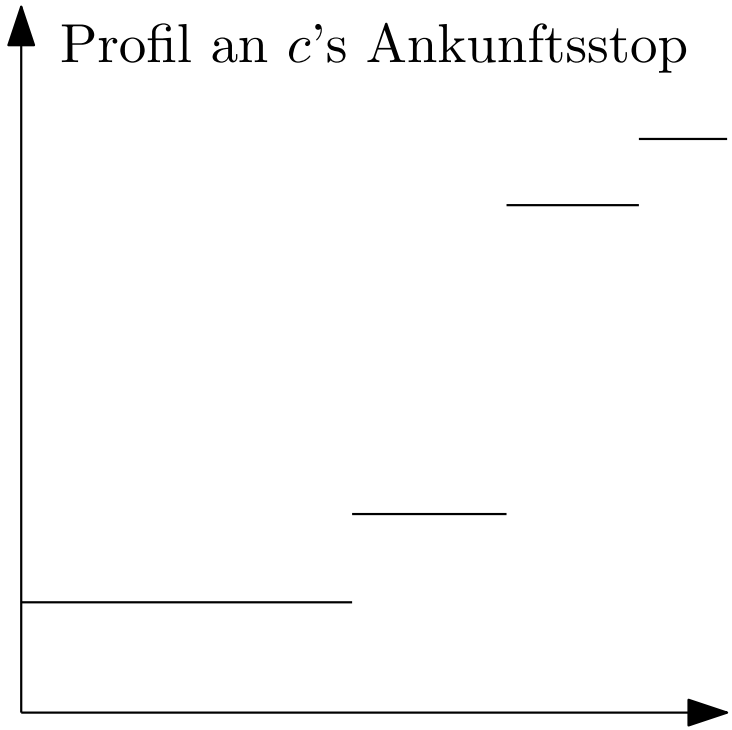
- wird

$d \leftarrow \tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)))$; $a \leftarrow \tau_c$;
if $a < \tau_{\text{arr}}(P[s_{\text{dep}}(c)][0])$ **then**
  if $d = \tau_{\text{dep}}(P[s_{\text{dep}}(c)][0])$ **then**
    $\tau_{\text{arr}}(P[s_{\text{dep}}(c)][0]) \leftarrow a$;
  else
    Füge (d, a) am Anfang von $P[s_{\text{dep}}(c)]$ ein;

Profil Connection Scan

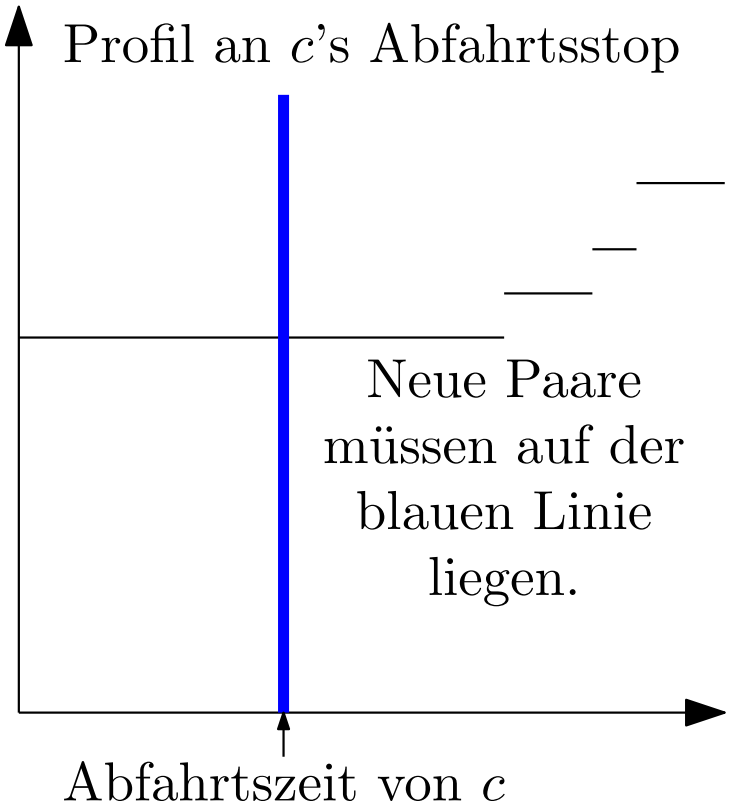
Für jede Connection c absteigend nach Abfahrtszeit:

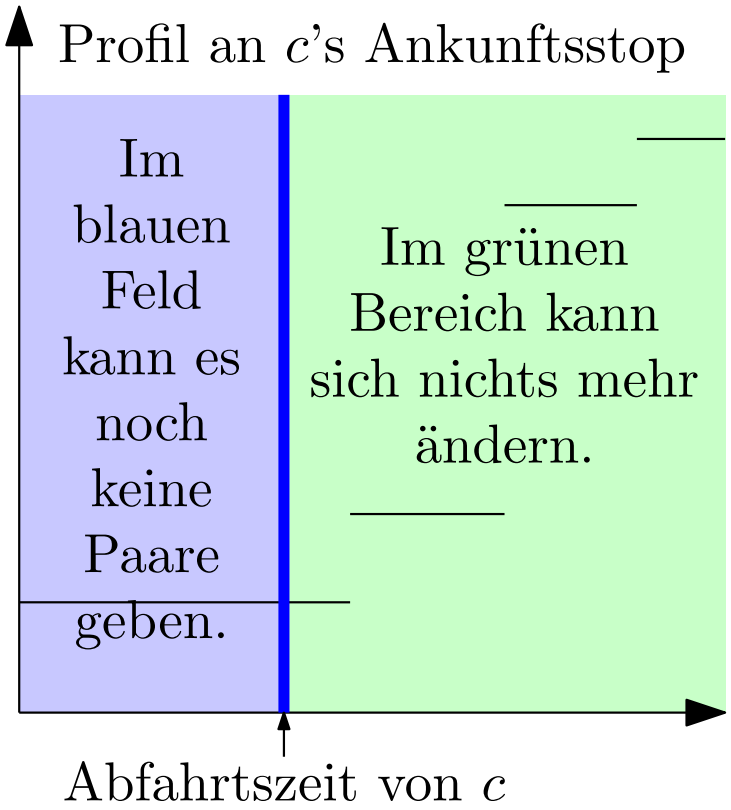




Profil Connection Scan

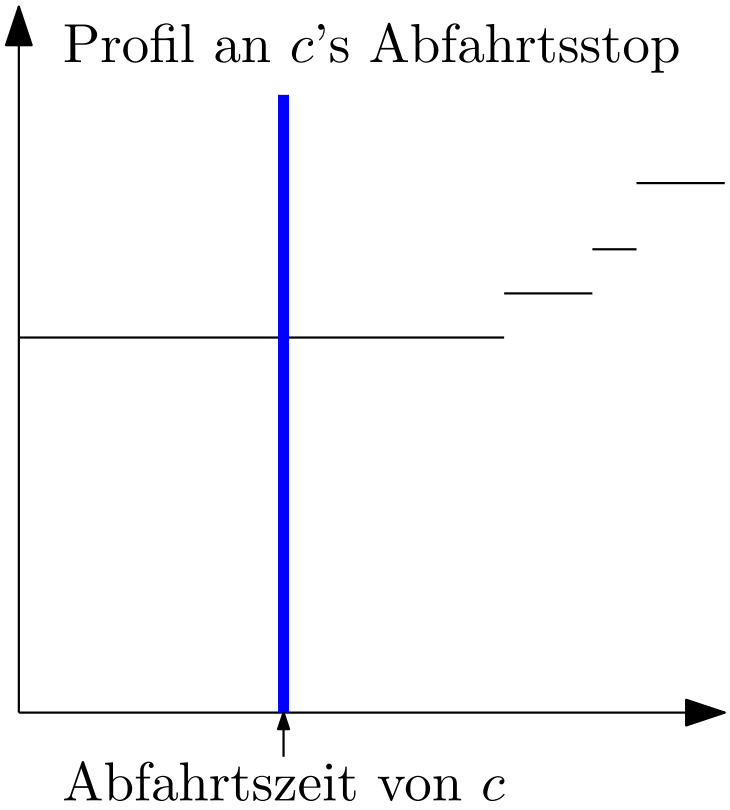
Für jede Connection c absteigend nach Abfahrtszeit:

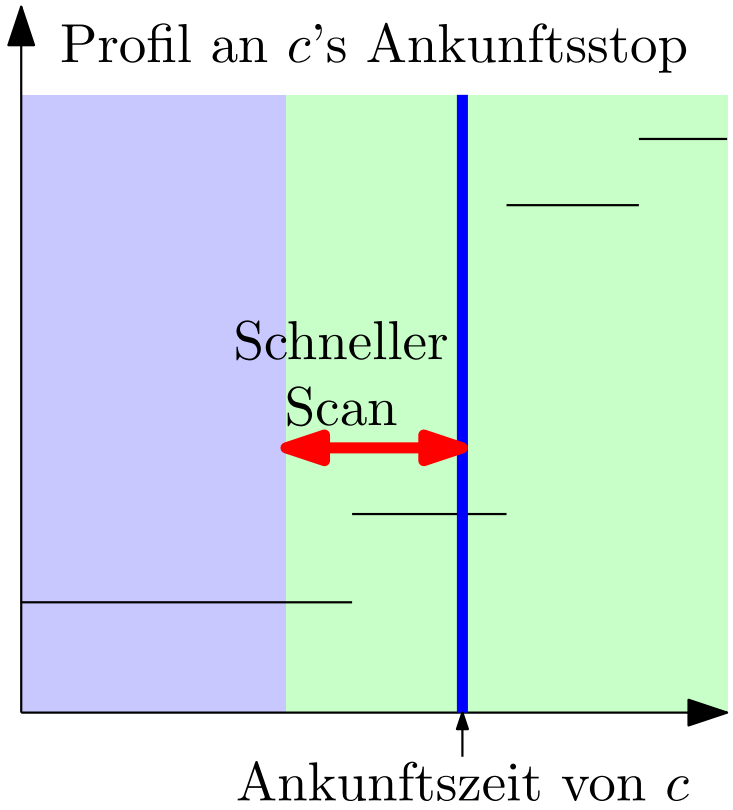




Profil Connection Scan

Für jede Connection c absteigend nach Abfahrtszeit:

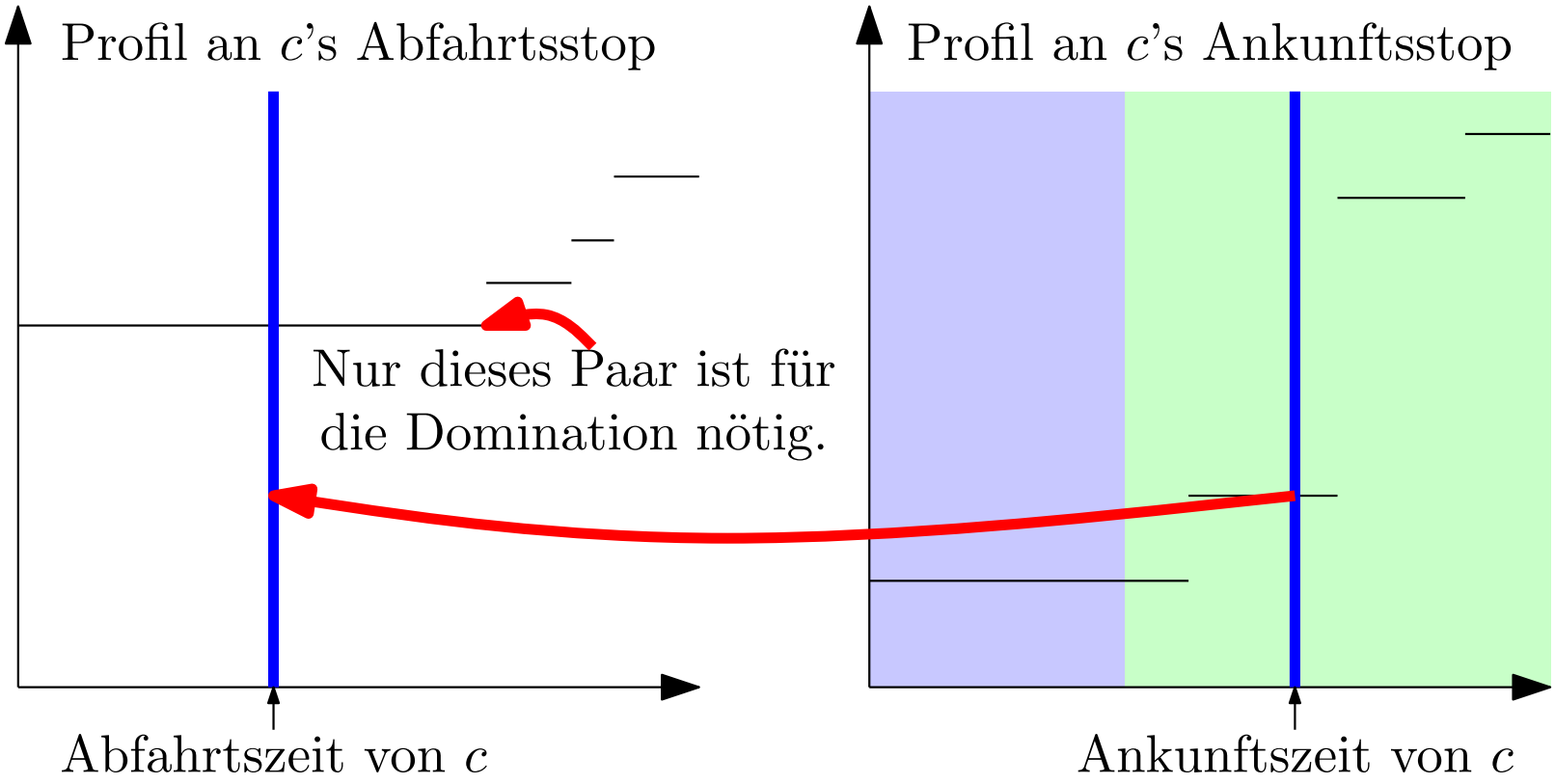




In der Praxis: sehr kurzer lineare Scan.
(Deswegen ist die sequentielle Suche besser.)

Profil Connection Scan

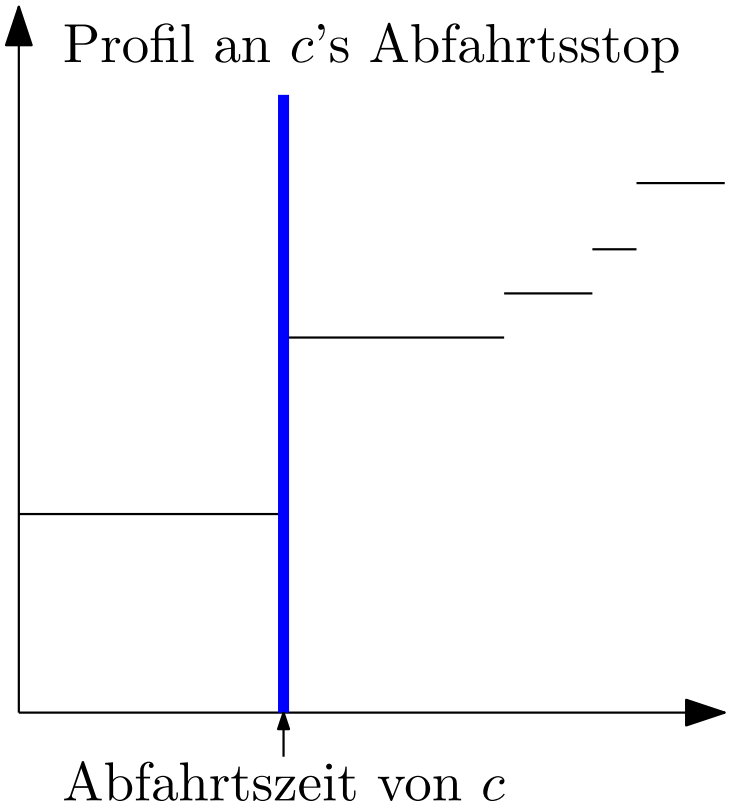
Für jede Connection c absteigend nach Abfahrtszeit:

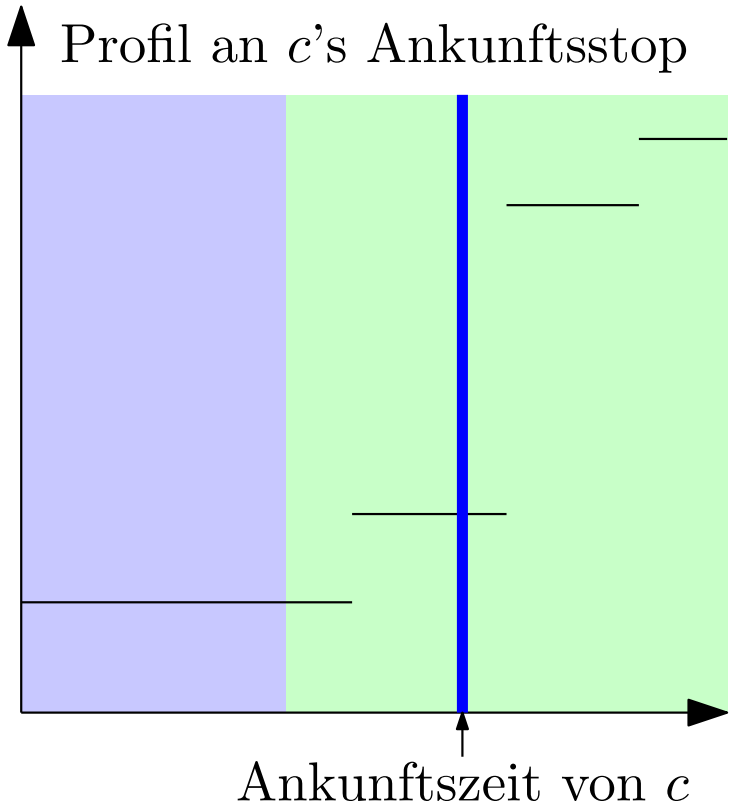


Teste ob das neue Paar über oder unter dem bereits existierenden ist.

Profil Connection Scan

Für jede Connection c absteigend nach Abfahrtszeit:





Neuen Sprung einfügen, wenn er unterhalb ist.
(Im Beispiel ist er unterhalb.)

τ_{min} und τ_{max} verwenden

- Profilanfrage enthält auch τ_{min} und τ_{max}
- Wie verwenden wir diese?

τ_{min} und τ_{max} verwenden

- Profilanfrage enthält auch τ_{min} und τ_{max}
- Wie verwenden wir diese?

τ_{min} **und** τ_{max} **verwenden**:

- Scanne nur Connections c mit $\tau_{\text{min}} \leq \tau_{\text{dep}}(c) \leq \tau_{\text{max}}$
- Braucht eine binäre Suche um die erste Connection c mit $\tau_{\text{dep}}(c) \leq \tau_{\text{max}}$ zu finden

Alternative Evaluierung

- Evaluierung in der Praxis schnell
- Geht das auch beweisbar in $O(1)$?

Alternative Evaluierung

- Evaluierung in der Praxis schnell
- Geht das auch beweisbar in $O(1)$?

- Ja, mit leichter Modifikation
- (Trick geht nicht mit Fußwegen)

Alternative Evaluierung

- **Idee:** Füge ein Paar für jede Connection ein
- Aus

> Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)), \tau_c)$ in $P[s_{\text{dep}}(c)]$ ein;

- wird

> $d \leftarrow \tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)))$;
> $x \leftarrow (d, \min\{\tau_c, \tau_{\text{arr}}(P[s_{\text{dep}}(c)][0])\})$;
> Füge x am Anfang von $P[s_{\text{dep}}(c)]$ ein;

Alternative Evaluierung

Warum?

- Aufeinander folgende Paare können nun die selbe Ankunftszeit haben
- Mehr Paare als nötig?
- Was bringt uns das?

Alternative Evaluierung

Warum?

- Aufeinander folgende Paare können nun die selbe Ankunftszeit haben
- Mehr Paare als nötig?
- Was bringt uns das?

Antwort

- Wann ein Paar eingefügt wird ist unabhängig von target
- Die Abfahrtszeit eines Paars ist unabhängig von target
- $\rightarrow$ Sequentieller Scan kann in die Vorberechnung verschoben werden
- $\rightarrow$ Schleifenrumpf in $O(1)$
- $\rightarrow$ Profilsuche beweisbar in Zeit linear in der Anzahl an Connections

Fußwege

- **Bisher**: Wir haben gesehen wie man finale Fußwege behandelt
- **Nun**: Wie man mit initialen und Fußwegen in der Mitte umgeht

Fußwege

- **Bisher**: Wir haben gesehen wie man finale Fußwege behandelt
- **Nun**: Wie man mit initialen und Fußwegen in der Mitte umgeht

Idee 1:

- Expandiere Fußwege zu Connections ähnlich der Fußwegexpansion beim zeitexpandiertem Graph
- Können sehr viele Connections werden

Fußwege

- **Bisher**: Wir haben gesehen wie man finale Fußwege behandelt
- **Nun**: Wie man mit initialen und Fußwegen in der Mitte umgeht

Idee 1:

- Expandiere Fußwege zu Connections ähnlich der Fußwegexpansion beim zeitexpandiertem Graph
- Können sehr viele Connections werden

Idee 2:

- Laufe Fußwege beim Einfügen von Paaren ab

Fußwege

- Aus

  Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)), \tau_c)$ in $P[s_{\text{dep}}(c)]$ ein;

- wird

  Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)), \tau_c)$ in $P[s_{\text{dep}}(c)]$ ein;
  for *alle Fußwege* $(x, s_{\text{dep}}(c))$ *mit Länge* ℓ **do**
  Füge $(\tau_{\text{dep}}(c) - \ell, \tau_c)$ in $P[x]$ ein;

Fußwege

Problem

- Wegen unterschiedlicher Fußweglängen werden Paar nicht mehr immer absteigend nach Abfahrtszeit eingefügt
- Einfügeoperation wird komplizierter

Fußwege

Problem

- Wegen unterschiedlicher Fußweglängen werden Paar nicht mehr immer absteigend nach Abfahrtszeit eingefügt
- Einfügeoperation wird komplizierter

Idee

- (d, a) soll eingefügt werden
- Verschiebe alle Paare (d', a') mit $d' < d$ in Hilfsarray Tmp
- Füge (d, a) wie gewohnt ein
- Füge danach alle Paare von Tmp wieder ein

Fußwege

Aus

Füge (d, a) in P ein;

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Fußwege

wird

$\mathrm{Tmp} \leftarrow \{\}$;
while $\tau_{\text{dep}}(P[\text{first}]) \leq d$ **do**
  Füge $P[\text{first}]$ in Tmp ein;
  Lösche $P[\text{first}]$ aus P;
if $a < \tau_{\text{arr}}(P[\text{first}])$ **then**
  if $\tau_{\text{dep}}(P[\text{first}]) = d$ **then**
    $\tau_{\text{arr}}(P[\text{first}]) \leftarrow a$;
  else
    Füge (d, a) am Anfang von P ein;
for *alle* (d', a') *in* Tmp *absteigend in* d' **do**
  if $a' < \tau_{\text{arr}}(P[0])$ **then**
    Füge (d', a') am Anfang von P ein;

Fußwege

Optimierung

- Wenn $P[s_{\text{dep}}(c)]$ nicht verändert wird, dann muss man die Fußwege nicht ablaufen
- Gültig wegen transitiv abgeschlossener Fußwege mit Dreiecksungleichung

Interpretation

- Wenn $P[s_{\text{dep}}(c)]$ nicht verändert wird, dann war jemand schon früher da. Diese Person kann von $s_{\text{dep}}(c)$ aus weiterlaufen und ist somit überall früher

Fußwege

Optimierung

- Wenn $P[s_{\text{dep}}(c)]$ nicht verändert wird, dann muss man die Fußwege nicht ablaufen
- Gültig wegen transitiv abgeschlossener Fußwege mit Dreiecksungleichung

Interpretation

- Wenn $P[s_{\text{dep}}(c)]$ nicht verändert wird, dann war jemand schon früher da. Diese Person kann von $s_{\text{dep}}(c)$ aus weiterlaufen und ist somit überall früher

Hinweis

Diese Optimierung macht meistens einen signifikanten Unterschied

Umstiege nachgelagert optimieren

Ziel:

- Ankunftszeit und Umstiege optimieren
- Ankunftszeit ist wichtiger
- D.h. finde Journey mit minimaler Anzahl an Umstiegen unter allen Journeys mit minimaler Ankunftszeit

Umstiege nachgelagert optimieren

- Aus

  $\tau_3 \leftarrow$ Ankunftszeit wenn Umsteigen;

- wird

  $\tau_3 \leftarrow$ Ankunftszeit wenn Umsteigen + ϵ;

- Für ein hinreichend kleines ϵ
- **Idee**: untere Bits kodieren Anzahl an Ausstiegen

Umstiege nachgelagert optimieren

Problem:

- Zwei Journeys *A* und *B*
  - *A* fährt um 7:00:00 los und kommt um 8:00:00 an braucht aber 20 Umstiege
  - *B* fährt um 7:00:00 los und kommt um 8:00:01 an braucht aber keinen Umstieg
- *A* wird gegenüber *B* vorgezogen

Umstiege nachgelagert optimieren

Problem:

- Zwei Journeys A und B
  - A fährt um 7:00:00 los und kommt um 8:00:00 an braucht aber 20 Umstiege
  - B fährt um 7:00:00 los und kommt um 8:00:01 an braucht aber keinen Umstieg
- A wird gegenüber B vorgezogen

- Problem kann heuristisch vermieden werden, indem man

$\tau_1 \leftarrow$ Ankunftszeit wenn man zum Ziel läuft;

durch

$\tau_1 \leftarrow \text{round}(\text{Ankunftszeit wenn man zum Ziel läuft});$

ersetzt

Umstiege Pareto-optimieren

Ziel:

- Ankunftszeit und Umstiege im Pareto-Sinn optimieren

Umstiege Pareto-optimieren

Ziel:

- Ankunftszeit und Umstiege im Pareto-Sinn optimieren

Idee:

- Ersetze skalare Ankunftszeit mit Vektor $v[i]$ mit konstanter Länge
  - In den Beispielen Länge 8
  - Geht mit beliebigen Längen
- $v[i]$ ist die Ankunftszeit am target wenn man höchstens i mal aussteigen darf

Vektoroperationen

Broadcast

- Eingabe: x
- Ausgabe: (x, x, x, x, x, x, x, x)

Vektoroperationen

Broadcast

- Eingabe: x
- Ausgabe: (x, x, x, x, x, x, x, x)

Minimum

- Eingabe: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ und $(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8)$
- Ausgabe: $(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8)$ mit $z_i = \min\{x_i, y_i\}$

Vektoroperationen

Broadcast

- Eingabe: x
- Ausgabe: (x, x, x, x, x, x, x, x)

Minimum

- Eingabe: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ und $(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8)$
- Ausgabe: $(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8)$ mit $z_i = \min\{x_i, y_i\}$

Shift

- Eingabe: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$
- Ausgabe: $(\infty, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$

Vektoroperationen

Broadcast

- Eingabe: x
- Ausgabe: (x, x, x, x, x, x, x, x)

Minimum

- Eingabe: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ und $(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8)$
- Ausgabe: $(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8)$ mit $z_i = \min\{x_i, y_i\}$

Shift

- Eingabe: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$
- Ausgabe: $(\infty, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$

- Geht alles mit SIMD/SSE/AVX

Grundgerüst

Initialisiere Datenstruktur an Stops;
Initialisiere Datenstruktur an Trips;

for *alle Connections c absteigend nach* $\tau_{\text{dep}}(c)$ **do**

```
/* 1. Bestimme Ankunftszeit von man in c startet   */
```
$\tau_1 \leftarrow$ Ankunftszeit wenn man zum Ziel läuft;
$\tau_2 \leftarrow$ Ankunftszeit wenn Sitzenbleiben, braucht Daten von Trip trip(c);
$\tau_3 \leftarrow$ Ankunftszeit wenn Umsteigen, braucht Daten vom Stop $s_{\text{arr}}(c)$;

```
/* τ_c Ankunftszeit wenn man in c beginnt          */
```
$\tau_c \leftarrow \min\{\tau_1, \tau_2, \tau_3\}$;

```
/* 2. Passe die Stop / Trip Datenstrukturen an     */
```
Füge τ_c in Daten von Stop $s_{\text{dep}}(c)$ ein;
Füge τ_c in Daten von Trip trip(c) ein;

- τ_1, τ_2, τ_3 und τ_c sind nun Vektoren
- Abfahrtszeiten bleiben Skalare

Zum Ziel Laufen

- Aus

  $\tau_1 \leftarrow$ Ankunftszeit wenn man zum Ziel läuft;

- wird

  if $s_{\text{arr}}(c) = \text{target}$ **then**
  $\quad \tau_1 \leftarrow \text{broadcast}(\tau_{\text{arr}}(c))$;
  else
  $\quad \tau_1 \leftarrow \text{broadcast}(\infty)$;

- (Finale Fußwege gehen genauso wie bisher)

Sitzenbleiben

Trip-Datenstruktur

- $T[x]$ ist eine Ankunftszeit
- $\rightarrow$ $T[x]$ wird Vektor

- Aus

  Initialisiere Datenstruktur an Trips;

- wird

  for *alle Trips x* **do**
  $T[x] \leftarrow \text{broadcast}(\infty)$;

Sitzenbleiben

Trip-Datenstruktur

- $T[x]$ ist eine Ankunftszeit
- $\rightarrow$ $T[x]$ wird Vektor

- Aus

  $\tau_2 \leftarrow$ Ankunftszeit wenn Sitzenbleiben;

- wird

  $\tau_2 \leftarrow T[\text{trip}(c)]$;

- (Diesmal sind die Variablen aber Vektoren)

Sitzenbleiben

Trip-Datenstruktur

- $T[x]$ ist eine Ankunftszeit
- $\rightarrow$ $T[x]$ wird Vektor

- Aus

  Füge τ_c in Daten von Trip trip(c) ein;

- wird

  $T[\text{trip}(c)] \leftarrow \tau_c$;

- (Diesmal sind die Variablen aber Vektoren)

Umsteigen

Stop-Datenstruktur

- Ankunftszeitfunktion bildet auf Vektor ab
- Aus

  Initialisiere Datenstruktur an Stop;

- wird

  for *alle Stops x* **do**
  $P[x] \leftarrow \{\forall \tau : \tau \mapsto \text{broadcast}(\infty)\}$;

Ankunftszeitfunktionen

- **Bisher**: P ist Array von $(\text{dep\_time}, \text{arr\_time})$-Paaren
- **Nun**: P ist Array von $(\text{dep\_time}, \text{vector})$-Paaren
- Array ist sortiert nach Abfahrtszeit

Ankunftszeitfunktionen

- **Bisher**: P ist Array von $(\text{dep\_time}, \text{arr\_time})$-Paaren
- **Nun**: P ist Array von $(\text{dep\_time}, \text{vector})$-Paaren
- Array ist sortiert nach Abfahrtszeit

Interpretation:

- P ist Profil von Stop x nach target
- P am Zeitpunkt τ auswerten ergibt Vektor v
- In $v[i]$ steht wann ich an target ankomme wenn ich
  - um τ
  - an x losfahre
  - und höchstens i mal aussteigen darf

Umstiege Pareto-optimieren

- Jedes Array hat ein $(\infty, \text{broadcast}(\infty))$-Paar
- Aus

for *alle Stops x* **do**
  $P[x] \leftarrow \{\forall \tau : \tau \mapsto \text{broadcast}(\infty)\}$;

- wird

for *alle Stops x* **do**
  $P[x] \leftarrow \{(\infty, \text{broadcast}(\infty))\}$;

Umstiege Pareto-optimieren

- Evaluierungs-Pseudo-Code bleibt unverändert
- **Aber**: a ist nun ein Vektor
- Aus

werte P zum Zeitpunkt τ aus;

- wird

for *i from* 0 *to* $\text{length}(P) - 1$ **do**
  $(d, a) \leftarrow P[i]$;
  if $\tau \leq d$ **then**
    Ergebnis ist a;
    breche Schleife ab;

Umstiege Pareto-optimieren

- Füge Paar ein, wenn es in mindestens einer Komponente besser ist
- Aus

> Füge $(\tau_{\text{dep}}(c), \tau_c - \tau_{\text{ch}}(s_{\text{dep}}(c)))$ in $P[s_{\text{dep}}(c)]$ ein;

- wird

$d \leftarrow \tau_{\text{dep}}(c) - \tau_{\text{ch}}(s_{\text{dep}}(c)))$; $x \leftarrow \min\{\tau_c, P[s_{\text{dep}}(c)][0]_{\text{vector}}\}$;
if $x \neq P[s_{\text{dep}}(c)][0]_{\text{vector}}$ **then**
  if $d = P[s_{\text{dep}}(c)][0]_{\text{dep\_time}}$ **then**
    $P[s_{\text{dep}}(c)][0]_{\text{vector}} \leftarrow x$;
  else
    Füge (d, x) am Anfang von $P[s_{\text{dep}}(c)]$ ein;

Umstiege Pareto-optimieren

- Vektor v der Länge n hat die Komponenten:

$$(v_1, v_2 \dots v_n)$$

- Journeys werden gefunden bis maximal n Mal einsteigen
- $\rightarrow$ bis $n - 1$ Umstiege

Umstiege Pareto-optimieren

- Vektor v der Länge n hat die Komponenten:

$$(v_1, v_2 \dots v_n)$$

- Journeys werden gefunden bis maximal n Mal einsteigen
- $\rightarrow$ bis $n - 1$ Umstiege

- Vektorlänge in der Regel 8
- 7 Umstiege ist für die meisten Anwendungen gut genug

Umstiege Pareto-optimieren

Beobachtung:

- Man kann die Shift-Operation so modifizieren, dass v_n die früheste Ankunftszeit ohne beschränkte Umstiege ist.
- Die Bedeutung von v_i für $i < n$ bleibt erhalten: höchstens i Ausstiege
- Ist in manchen Anwendungen nützlich

Modifiziertes Shift

- Eingabe: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$
- Ausgabe: $(\infty, x_1, x_2, x_3, x_5, x_6, \min\{x_7, x_8\})$

Zeit Kompression

Problem: Algorithmus ist memory-bound
Der Speicher ist fast vollständig mit Ankunftszeiten gefüllt
$\rightarrow$ Komprimiere Ankunftszeiten

Beobachtung: Nicht an jedem Zeitpunkt fährt ein Zug

Idee: Berechne für jeden Stop ein geordnetes Array von Zeitpunkten $t_0, t_1, \ldots, t_n$ an denen ein Zug abfährt oder ankommt

- Indizes respektieren die zeitliche Ordnung, d.h., $t_i < t_j \iff i < j$
- Indizes passen oft in 16 Bit
- Kopiere Indizes anstatt von Zeitpunkten

Zeit Kompression

Problem: Algorithmus ist memory-bound
Der Speicher ist fast vollständig mit Ankunftszeiten gefüllt
$\rightarrow$ Komprimiere Ankunftszeiten

Beobachtung: Nicht an jedem Zeitpunkt fährt ein Zug

Idee: Berechne für jeden Stop ein geordnetes Array von Zeitpunkten $t_0, t_1, \ldots, t_n$ an denen ein Zug abfährt oder ankommt

- Indizes respektieren die zeitliche Ordnung, d.h., $t_i < t_j \iff i < j$
- Indizes passen oft in 16 Bit
- Kopiere Indizes anstatt von Zeitpunkten

Fußwege

Nicht triviale Interaktion mit Fußwegen (nicht in der Vorlesung)

Pfadentpackung

Option 1:

- Speichere an jeder Ankunftszeit das erste Leg einer entsprechenden Journey
- Entpacke Journeys rekursiv

Pfadentpackung

Option 1:

- Speichere an jeder Ankunftszeit das erste Leg einer entsprechenden Journey
- Entpacke Journeys rekursiv

Option 2:

- Traversiere Fahrplan DFS-mässig von source in der Zeit vorwärts
- Benutze Profile um frühzeitig zu prunen
- Kann genutzt werden um eine Journey zu finden
- Kann auch genutzt werden um alle optimalen Journeys zu finden

Laufzeiten

London Instanz mit 4 850 431 Connections.

Früheste Ankunft One-to-One:

- Time-Expanded: 64.4 ms
- Time-Dependent: 10.9 ms
- Connection Scan: 2.0 ms

Früheste Ankunft One-to-All:

- Time-Expanded: 876.2 ms
- Time-Dependent: 18.9 ms
- Connection Scan: 9.7 ms

(Time-Dependent kriegt man etwas schneller, mit Ideeen die nicht im Kurs vorkommen.)

Laufzeiten

Non-Pareto Profil All-to-One:

- Self-Pruning-Connection-Setting : 1 262 ms
- Connection Scan: 177 ms
- + constant eval: 134 ms
- + time compress: 104 ms

Pareto Profil All-to-One (mit höchstens 8 Zügen pro Journey):

- RAPTOR : 1 179 ms
- Connection Scan: 255 ms
- + SSE: 221 ms

- Connection Scan mit expandierten Fußwegen

Literatur I

Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner.

Intriguingly simple and fast transit routing.

In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013.

Ben Strasser and Dorothea Wagner.

Connection scan accelerated.

In *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX'14)*, pages 125–137. SIAM, 2014.