Karlsruher Institut für Technologie

Algorithmen für Routenplanung

20. Vorlesung, Sommersemester 2020

Jonas Sauer | 13./15. Juli 2020

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 nah können wir an den Eingabedaten bleiben?

Wiederholung: Connection

Zur Erinnerung:

- Eine Connection ist ein 5-Tupel aus:
  - Abfahrtsstop: $v_{\text{dep}}(c)$
  - Ankunftsstop: $v_{\text{arr}}(c)$
  - Abfahrtszeit: $\tau_{\text{dep}}(c)$
  - Ankunftszeit: $\tau_{\text{arr}}(c)$
  - Trip: $\texttt{trip}(c)$

Earliest Arrival Connection Scan

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Connection Scan

Idee:

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

Connection Scan

Idee:

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

Datenstruktur:

- Tentative Ankunftszeit $d[v]$ pro Stop v
- Erreichbarkeitsbit $r[T]$ pro Trip T

Connection Scan

Idee:

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

Datenstruktur:

- Tentative Ankunftszeit $d[v]$ pro Stop v
- Erreichbarkeitsbit $r[T]$ pro Trip T

Connection-Relaxierung:

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

- Wenn c erreichbar ist, dann

Connection Scan

Idee:

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

Datenstruktur:

- Tentative Ankunftszeit $d[v]$ pro Stop v
- Erreichbarkeitsbit $r[T]$ pro Trip T

Connection-Relaxierung:

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

Connection Scan

Idee:

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

Datenstruktur:

- Tentative Ankunftszeit $d[v]$ pro Stop v
- Erreichbarkeitsbit $r[T]$ pro Trip T

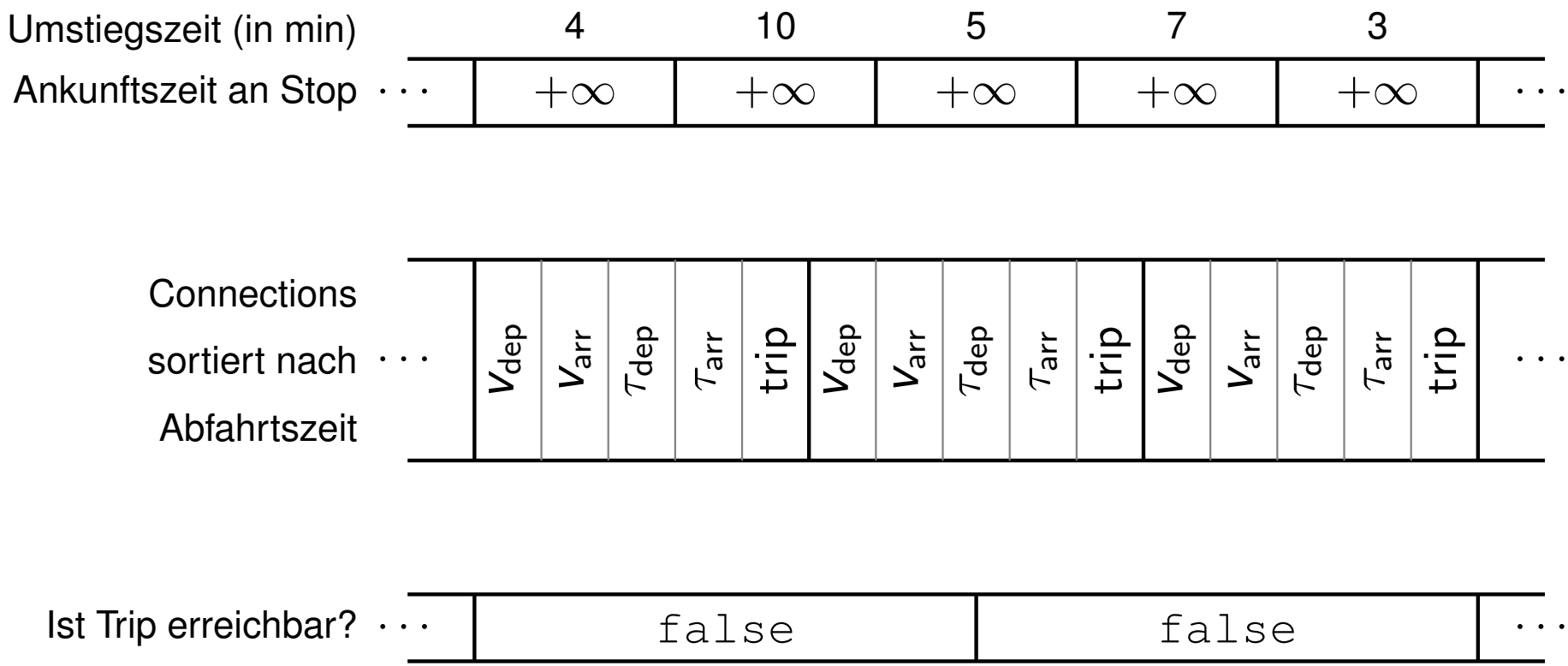
Connection-Relaxierung:

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

Connection Scan

Problem der frühesten Ankunft

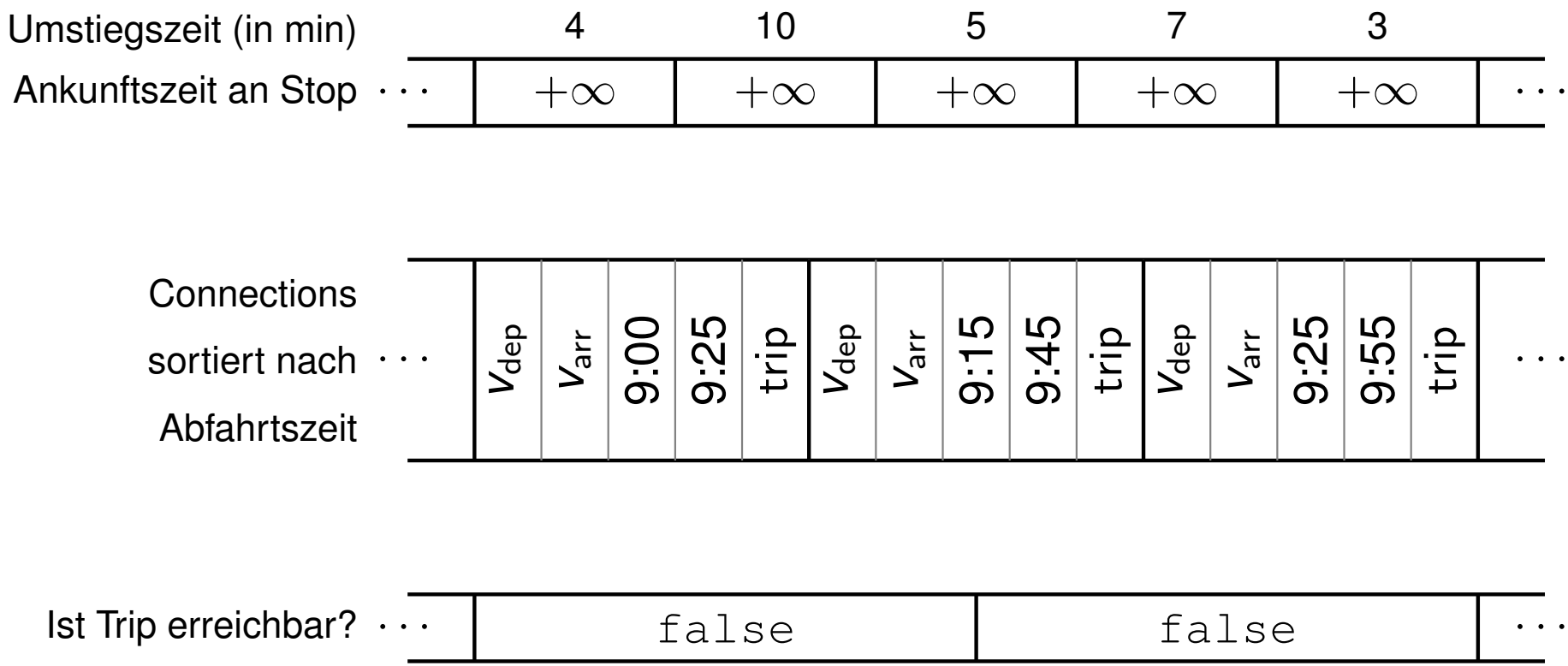
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

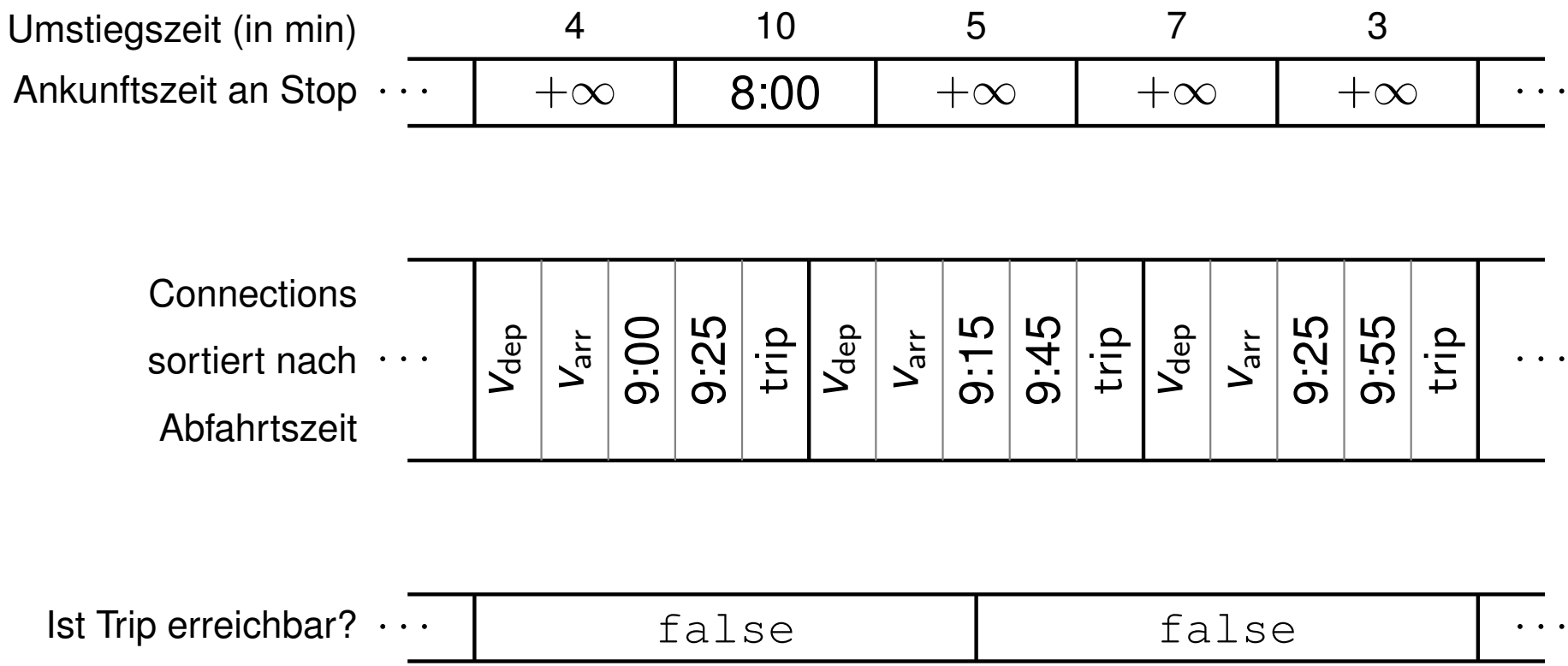
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

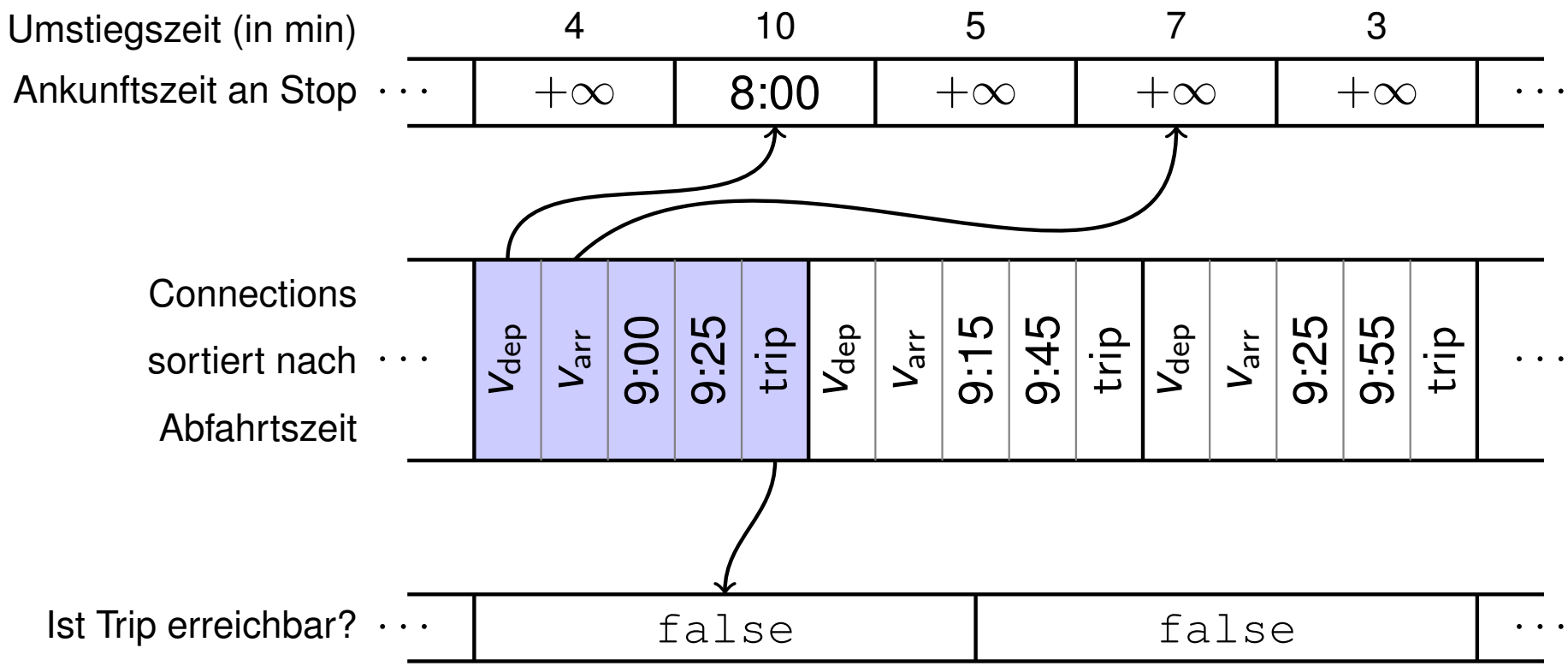
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

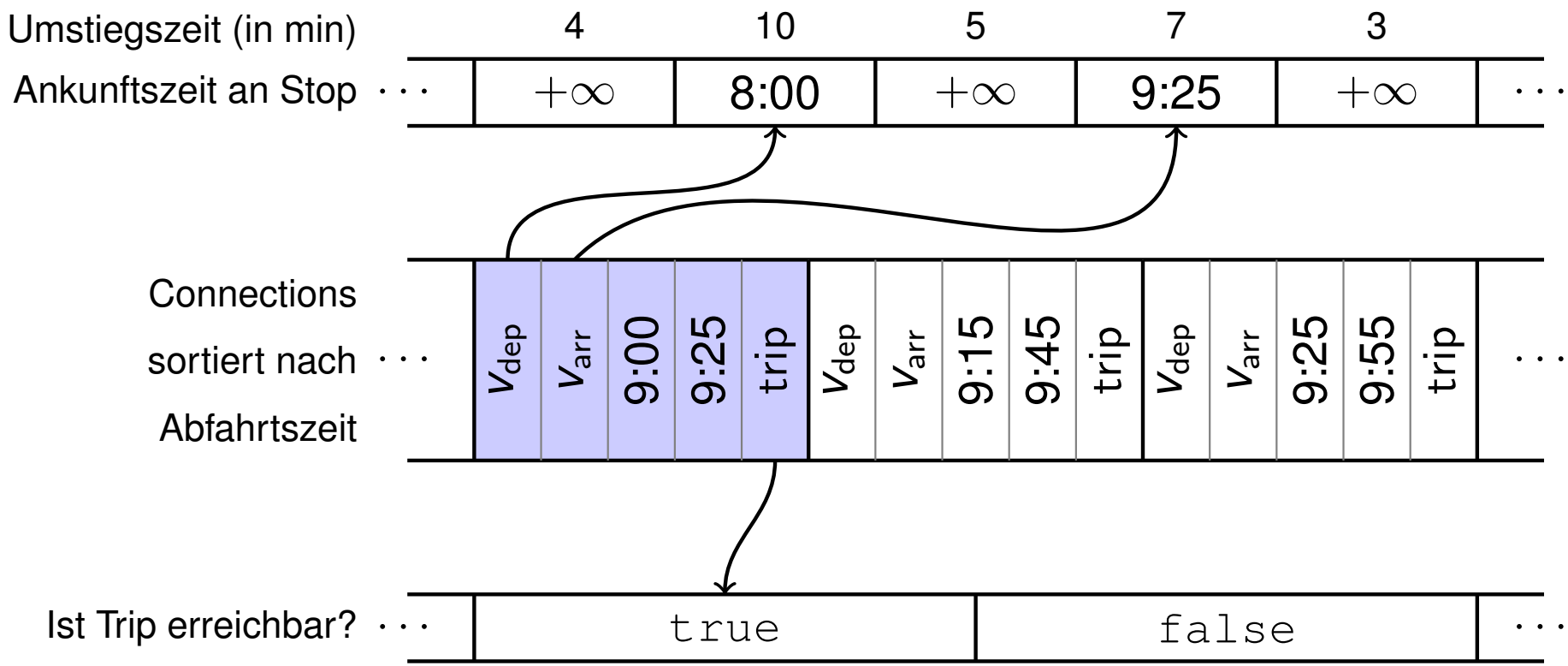
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

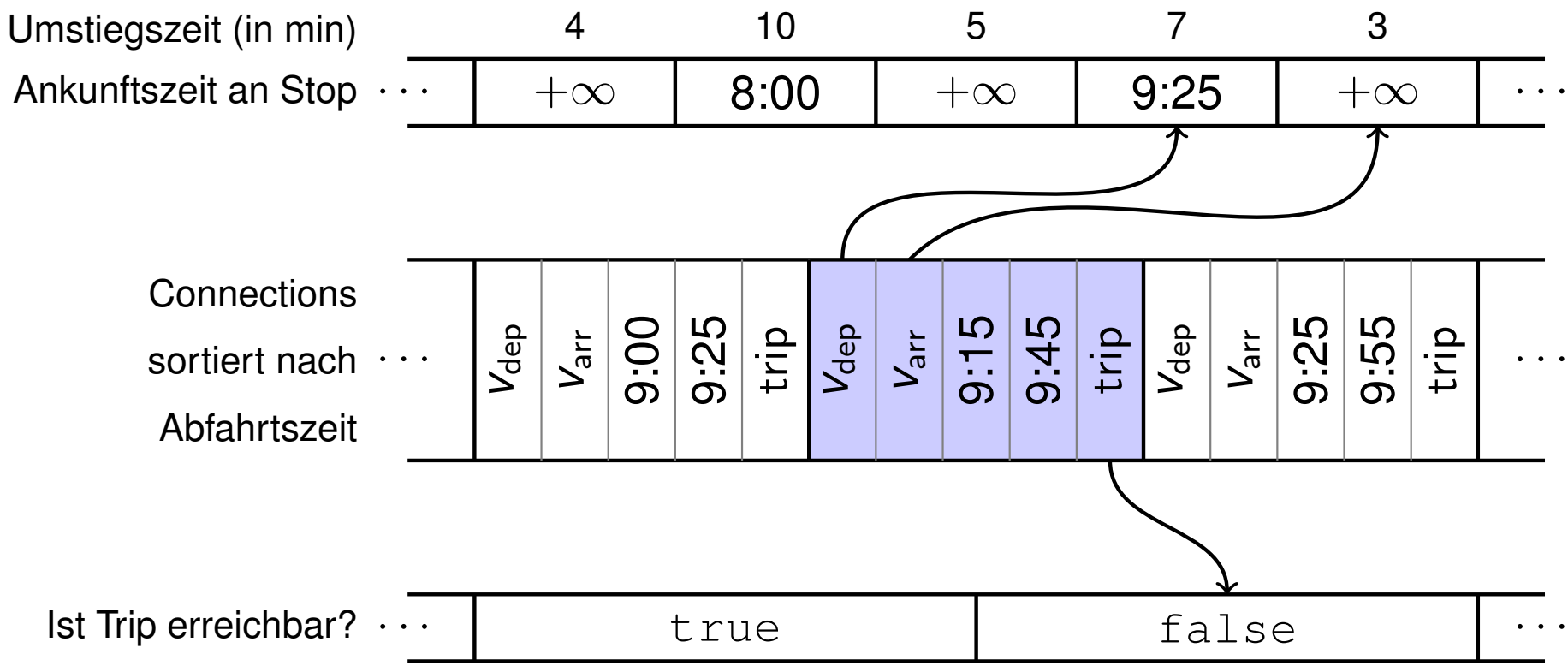
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

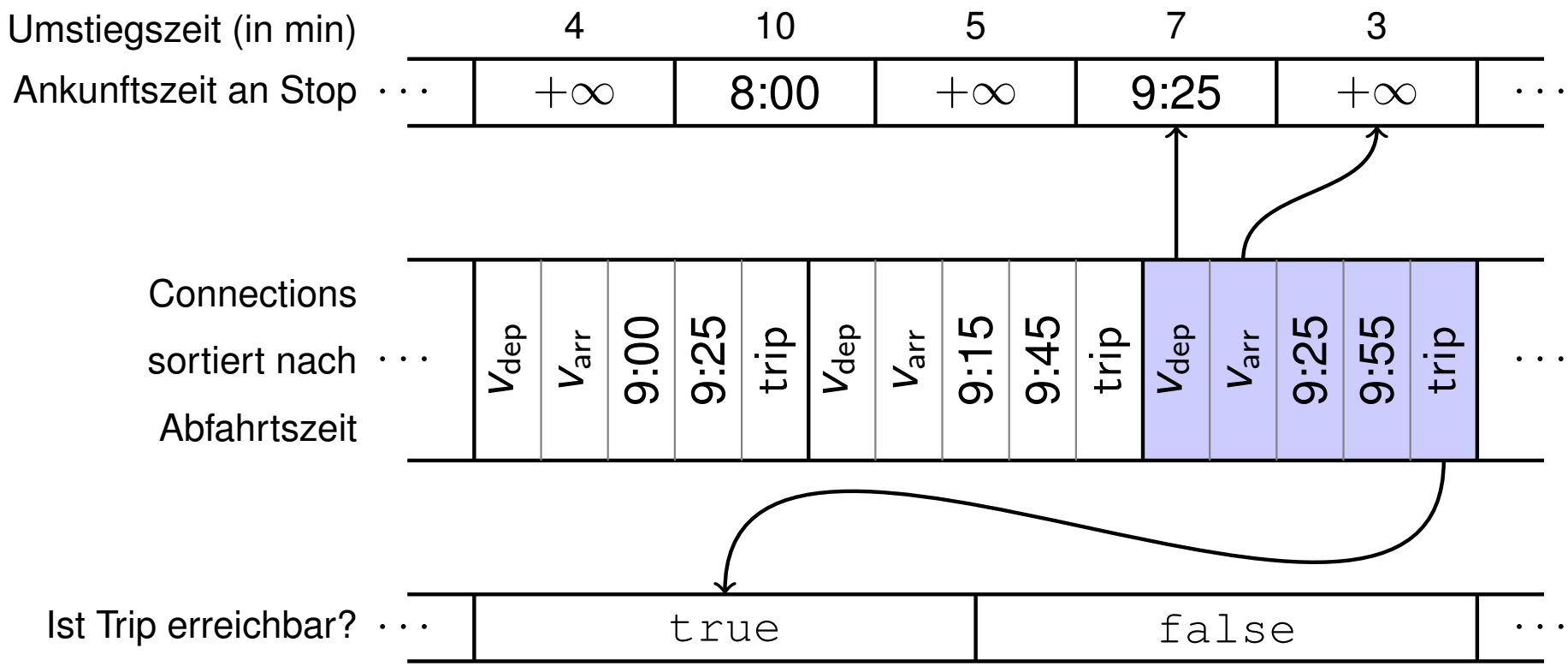
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

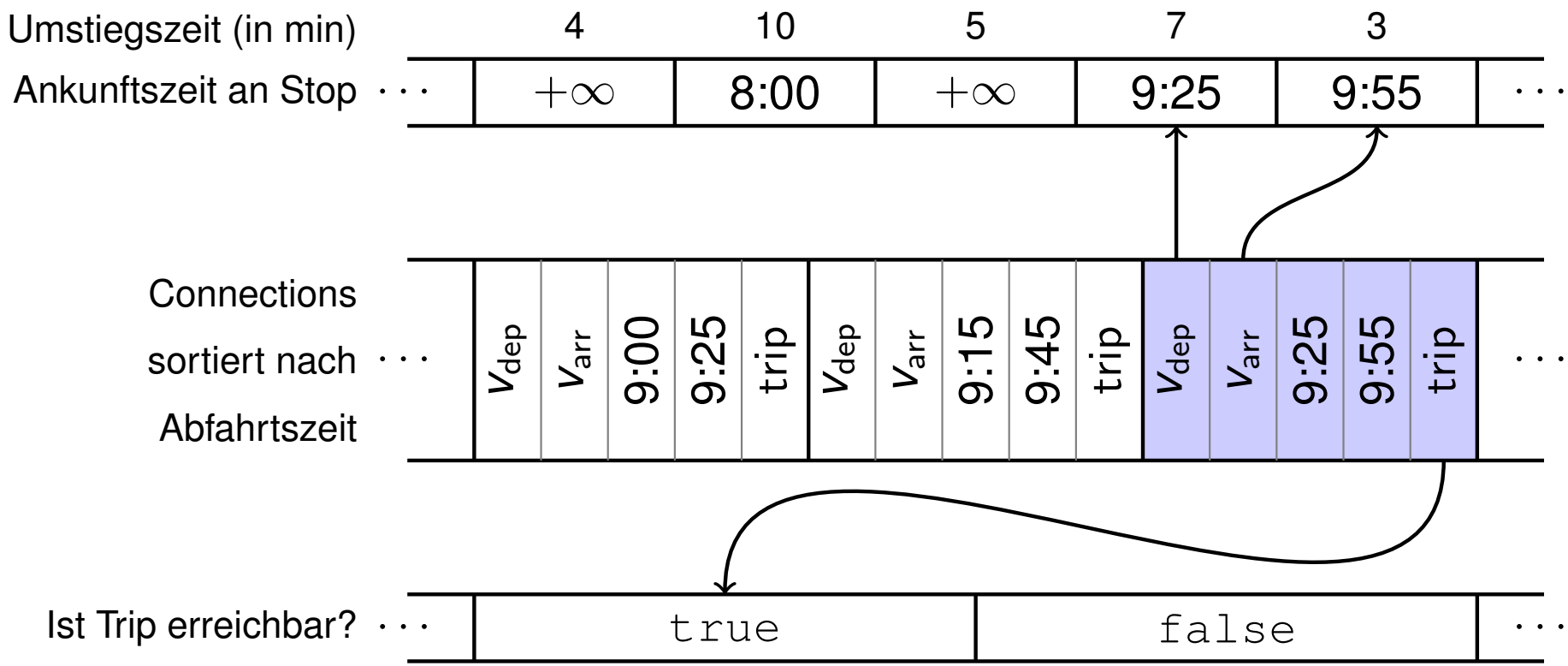
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

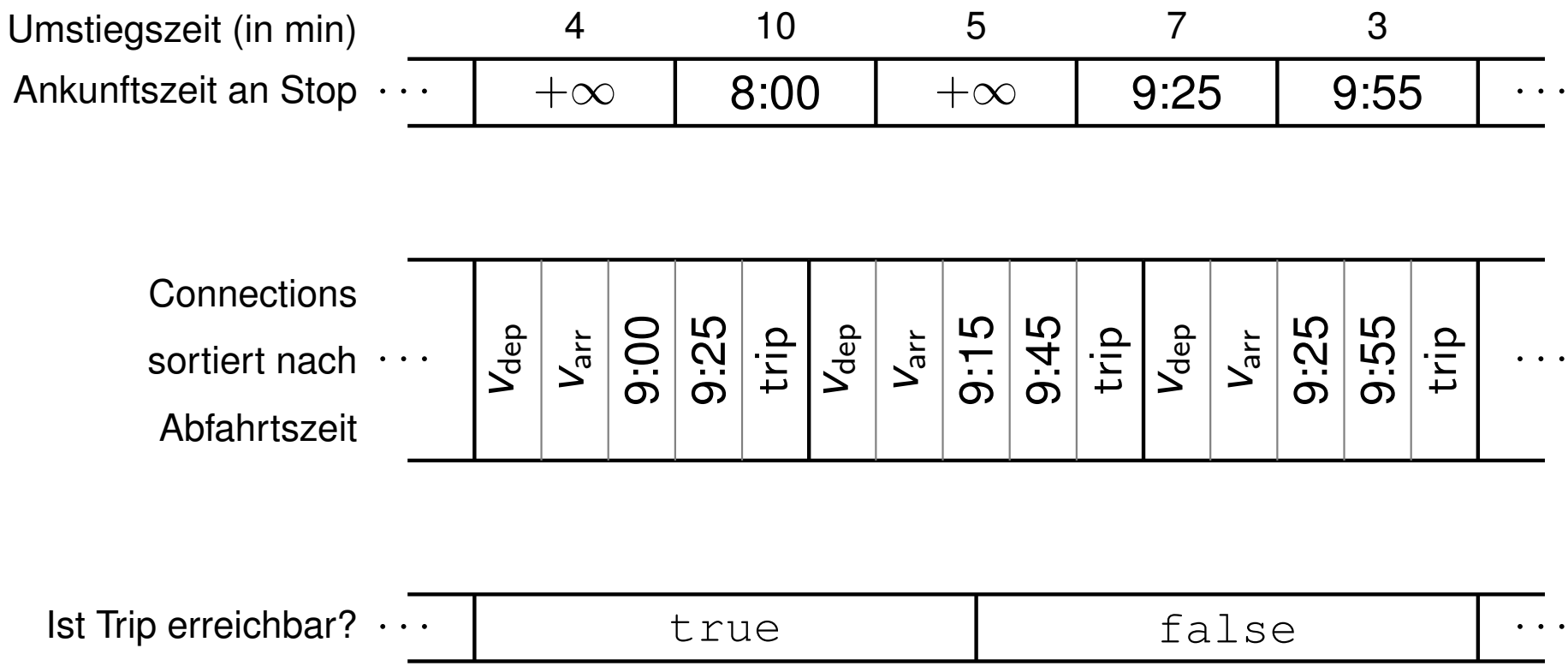
Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

Problem der frühesten Ankunft

Eingabe: Sortierte Connection-Liste, Startstop, Startzeit, Zielstop
Ausgabe: Früheste Ankunftszeit an Zielstop



Connection Scan

$d[v] \leftarrow \infty$ für alle Stops v
$d[s] \leftarrow \tau_{\text{dep}} - \tau_{\text{ch}}(s)$
$r[T] \leftarrow \texttt{false}$ für alle Trips T

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

Fußwege

Idee:

- Nach Scannen von Connection c:
  Relaxiere alle von $v_{\text{arr}}(c)$ ausgehenden Fußwege
- Aber nur, falls $d[v_{\text{arr}}(c)]$ durch c verbessert wurde

Fußwege

Idee:

- Nach Scannen von Connection c:
  Relaxiere alle von $v_{\text{arr}}(c)$ ausgehenden Fußwege
- Aber nur, falls $d[v_{\text{arr}}(c)]$ durch c verbessert wurde

Begründung:

- Angenommen, c verbessert $d[v_{\text{arr}}(c)]$ nicht
- Dann wurde bereits eine bessere Journey J nach $v_{\text{arr}}(c)$ gefunden
- Für jeden Fußweg $(v_{\text{arr}}(c), v)$: Hänge Fußweg an J dran
- Ankunftszeit an v ist dann besser, als wenn man Fußweg an c hängt

Voraussetzung:

Fußwege sind transitiv abgeschlossen und erfüllen Dreiecksungleichung!

Connection Scan

$d[v] \leftarrow \infty$ für alle Stops v
$d[s] \leftarrow \tau_{\text{dep}} - \tau_{\text{ch}}(s)$
$r[T] \leftarrow \texttt{false}$ für alle Trips T
for *alle Fußwege* (s, v) *mit Länge* ℓ **do**
  $d[v] \leftarrow \min\{d[v], \tau_{\text{dep}} + \ell\}$

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

Startkriterium

Beobachtung:

- Connections vor Startzeit τ_{dep} können nicht verwendet werden
- Finde mit binärer Suche die erste Connection c mit $\tau_{\text{dep}}(c) \geq \tau_{\text{dep}}$
- Scanne erst ab c

Stoppkriterium

Bisher: Wir lösen das One-to-All-Problem.

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

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Stoppkriterium

Bisher: Wir lösen das One-to-All-Problem.

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

Beobachtung:

- Connections, die nach der Ankunftszeit an t abfahren, sind nie nützlich

$\Rightarrow$ Scan abbrechen, sobald für aktuelle Connection c gilt: $\tau_{\text{dep}}(c) \geq d[t]$

Profile Connection Scan

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

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.

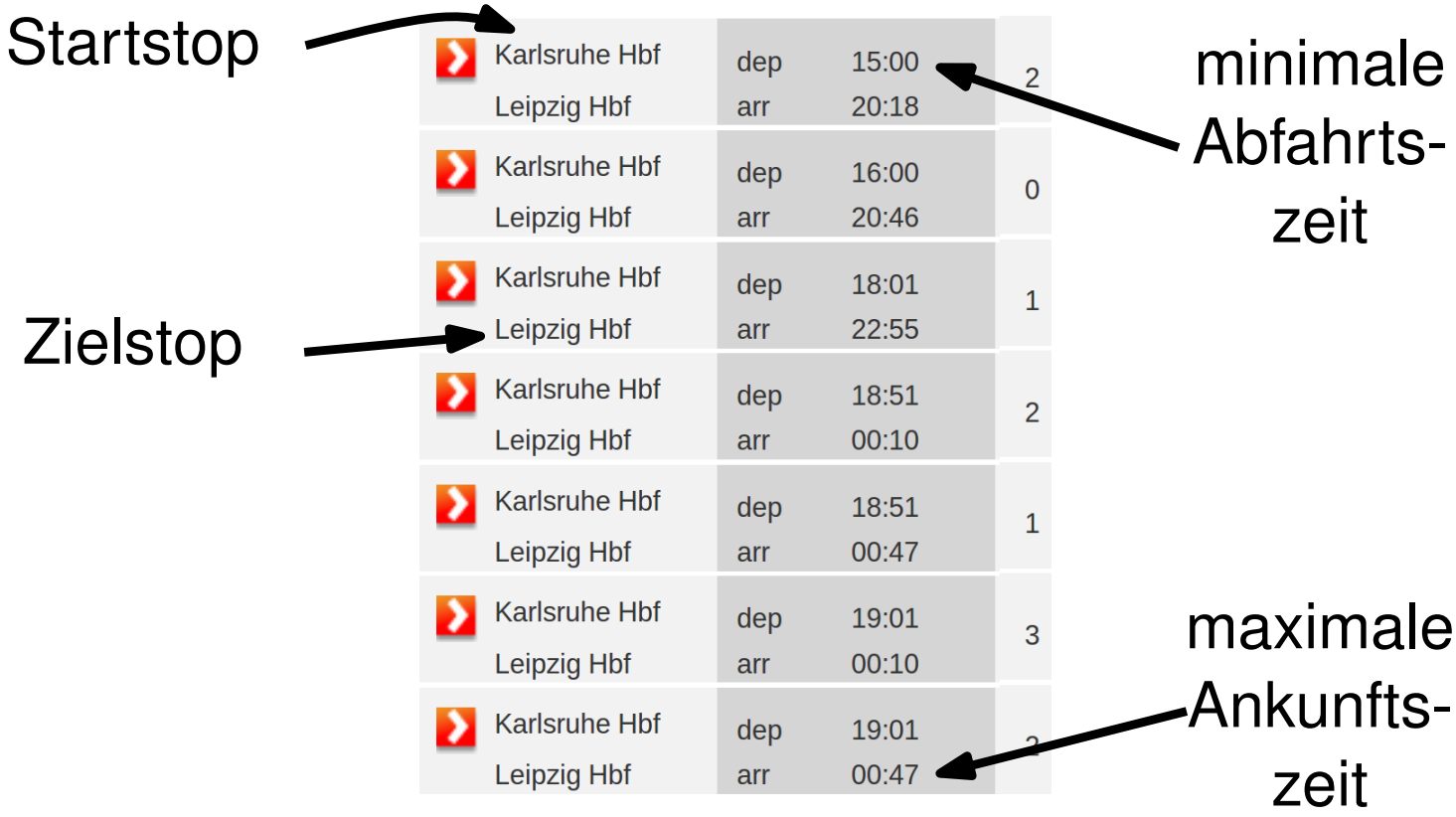
Karlsruhe Hbf	dep	15:00	2
Leipzig Hbf	arr	20:18	
Karlsruhe Hbf	dep	16:00	0
Leipzig Hbf	arr	20:46	
Karlsruhe Hbf	dep	18:01	1
Leipzig Hbf	arr	22:55	
Karlsruhe Hbf	dep	18:51	2
Leipzig Hbf	arr	00:10	
Karlsruhe Hbf	dep	18:51	1
Leipzig Hbf	arr	00:47	
Karlsruhe Hbf	dep	19:01	3
Leipzig Hbf	arr	00:10	
Karlsruhe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

Screenshot von bahn.de

Profil-Anfragen

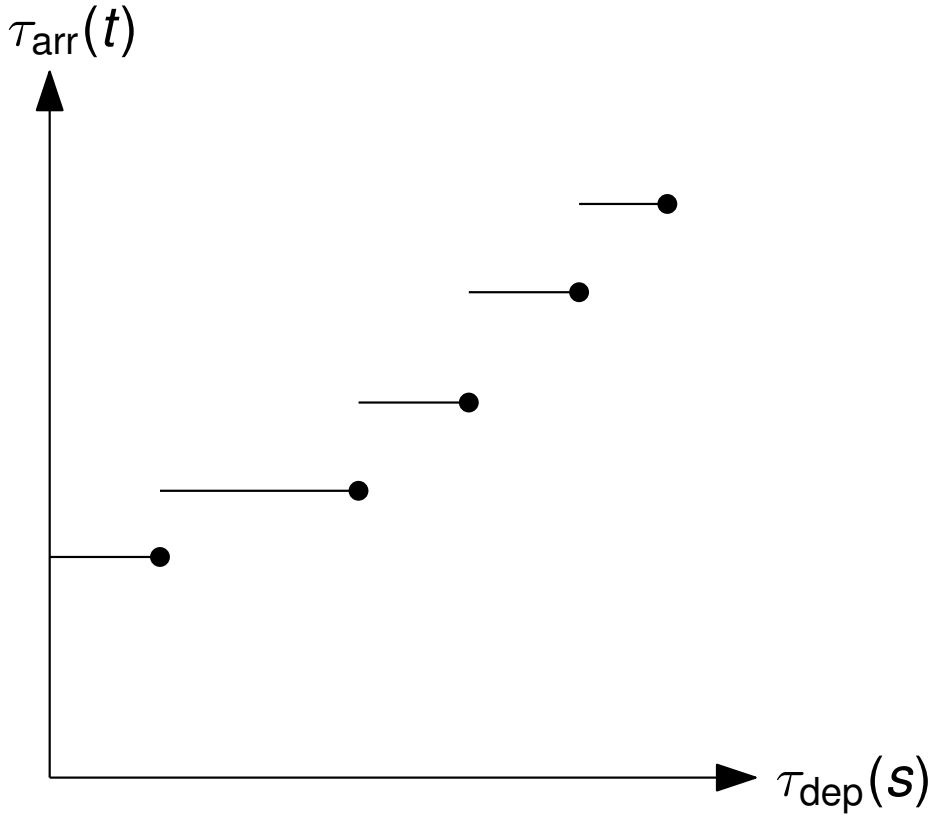
Problem: Der Fahrgast kennt seine Abfahrts- und Ankunftszeit oft nicht.

Lösung: Journeys für eine Zeitspanne angeben.



Screenshot von bahn.de

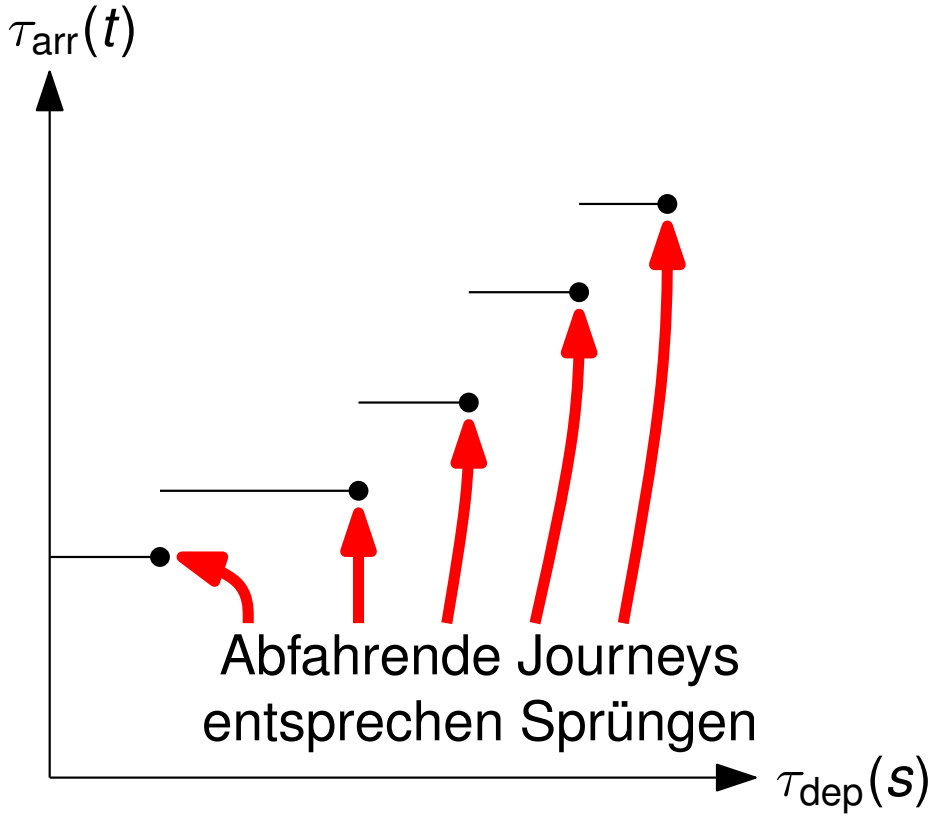
Profile Connection Scan



Problem der frühesten *Rückwärts*-Ankunftsprofile

Eingabe: Fahrplan, Zielstop t, Zeitspanne $[\tau_{\min}, \tau_{\max}]$
Ausgabe: s–t-Profil für jeden Stop s (außer t)

Profile Connection Scan



Problem der frühesten *Rückwärts*-Ankunftsprofile

Eingabe: Fahrplan, Zielstop t, Zeitspanne $[\tau_{\min}, \tau_{\max}]$
Ausgabe: s–t-Profil für jeden Stop s (außer t)

Grundgerüst

Initialisiere Profil $P[v]$ pro Stop v
Initialisiere optimale Ankunftszeit $d_{\text{trip}}[T]$ pro Trip T

for *alle 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 man sitzenbleibt (benutzt $d_{\text{trip}}[\text{trip}(c)]$)

$\tau_3 \leftarrow$ Ankunftszeit, wenn man umsteigt (benutzt $P[v_{\text{arr}}(c)]$)

`// τc: Optimale Ankunftszeit, falls c benutzt wird`

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

`// 2. Aktualisiere Stop- und Trip-Daten`

Füge τ_c zu $P[v_{\text{dep}}(c)]$ hinzu

Aktualisiere $d_{\text{trip}}[\text{trip}(c)]$ mit τ_c

Grundgerüst

Idee:

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

Grundgerüst

Idee:

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

Beobachtung:

- Wenn eine Connection c eingeführt wird, gibt es keine frühere

$\Rightarrow$ Niemand, der bereits unterwegs ist, kann c verwenden

$\Rightarrow$ c kann nur am Anfang einer Journey vorkommen

Zum Ziel laufen

- Verwalte Array $d_t[v]$, das die Laufdistanz von v zu t enthält
- $d_t[\cdot]$ wird in einem Preprocessing-Schritt berechnet
- Aus

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

- wird

  for *alle Stops* v **do**
  $d_t[v] \leftarrow \infty$
  for *alle Fußwege* (v, t) *mit Länge* ℓ **do**
  $d_t[v] \leftarrow \ell$
  $d_t[t] \leftarrow 0$
  ...
  $\tau_1 \leftarrow \tau_{\text{arr}}(c) + d_t[v_{\text{arr}}(c)]$

Sitzenbleiben

Trip-Datenstruktur

- Verwalte Array $d_{\text{trip}}[T]$, mit einem Wert pro Trip T
- $d_{\text{trip}}[T]$ ist die früheste Ankunftszeit an t, wenn man in T startet
- Initial gibt es keine Connections in T

$\Rightarrow$ $d_{\text{trip}}[T] \leftarrow \infty$ für alle T

- Aus

> Initialisiere optimale Ankunftszeit $d_{\text{trip}}[T]$ pro Trip T

- wird

> **for** *alle Trips* T **do**
>
> $\quad d_{\text{trip}}[T] \leftarrow \infty$

Sitzenbleiben

Trip-Datenstruktur

- Verwalte Array $d_{\text{trip}}[T]$, mit einem Wert pro Trip T
- $d_{\text{trip}}[T]$ ist die früheste Ankunftszeit an t, wenn man in T startet
- Initial gibt es keine Connections in T

$\Rightarrow$ $d_{\text{trip}}[T] \leftarrow \infty$ für alle T

- Aus

  $\tau_2 \leftarrow$ Ankunftszeit, wenn man sitzenbleibt

- wird

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

Sitzenbleiben

Trip-Datenstruktur

- Verwalte Array $d_{\text{trip}}[T]$, mit einem Wert pro Trip T
- $d_{\text{trip}}[T]$ ist die früheste Ankunftszeit an t, wenn man in T startet
- Initial gibt es keine Connections in T

$\Rightarrow$ $d_{\text{trip}}[T] \leftarrow \infty$ für alle T

- Aus

  Aktualisiere $d_{\text{trip}}[\text{trip}(c)]$ mit τ_c

- wird

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

Umsteigen

Stop-Datenstruktur

- Verwalte Array $P[\cdot]$ von Profilen
- $P[v]$ ist das Profil von Stop v zu t

  (Details zu Profilen auf nächster Folie)

- Aus

  Initialisiere Profil $P[v]$ pro Stop v

- wird

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

Umsteigen

Stop-Datenstruktur

- Verwalte Array $P[\cdot]$ von Profilen
- $P[v]$ ist das Profil von Stop v zu t

  (Details zu Profilen auf nächster Folie)

- Aus

  $\tau_3 \leftarrow$ Ankunftszeit, wenn man umsteigt

- wird

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

Umsteigen

Stop-Datenstruktur

- Verwalte Array $P[\cdot]$ von Profilen
- $P[v]$ ist das Profil von Stop v zu t

  (Details zu Profilen auf nächster Folie)

- Aus

  Füge τ_c zu $P[v_{\text{dep}}(c)]$ hinzu

- wird

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

Profile – Datenstruktur

- $P[v]$ ist stückweise lineare Ankunftszeit-Funktion

$\Rightarrow$ Speichere $P[v]$ als Array von Breakpoints (d, a)

$d \mathrel{\hat{=}}$ Abfahrtszeit, $a \mathrel{\hat{=}}$ Ankunftszeit

- Breakpoints dominieren sich nicht
  - Es gibt keine zwei Breakpoints (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 $P[v][0]$ steht die früheste Abfahrt

Profile – Initialisierung

- Jedes Array endet mit Breakpoint (∞, ∞)
- Also ist die Ankunftszeit ∞, wenn alle Connections abgefahren sind
- (∞, ∞)-Breakpoint wird bei der Initialisierung eingefügt
- Aus

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

- wird

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

Profile – Auswertung

- Auswertung mit binärer oder linearer Suche
- Hier ist lineare Suche besser (Begründung gleich)
- Aus

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

- wird

for $i \leftarrow 0, \ldots, |P[v_{\text{arr}}(c)]| - 1$ **do**
  $(d, a) \leftarrow P[v_{\text{arr}}(c)][i]$
  if $\tau_{\text{arr}}(c) \leq d$ **then**
    $\tau_3 \leftarrow a$
    break

Profile – Einfügen

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

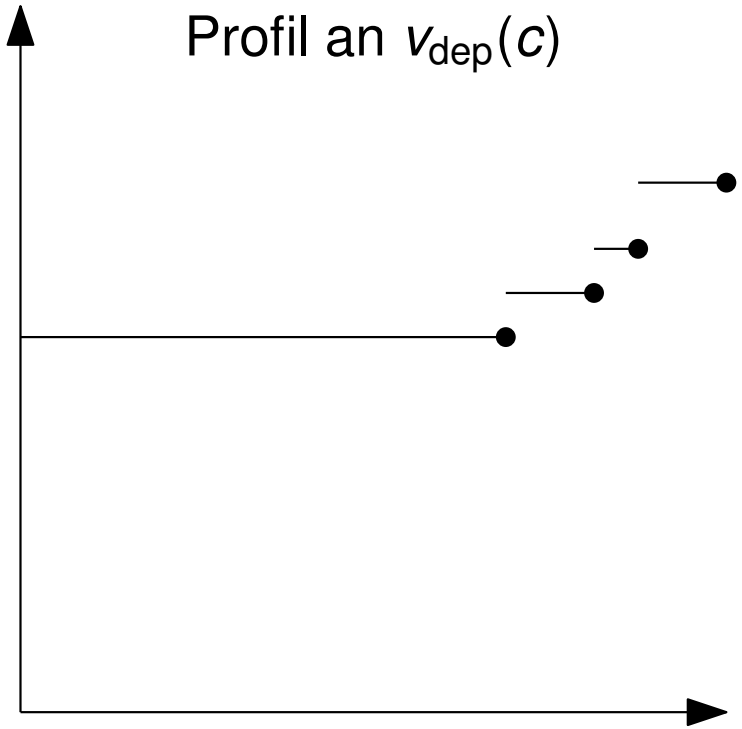
> Füge $(\tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c)), \tau_c)$ in $P[v_{\text{dep}}(c)]$ ein

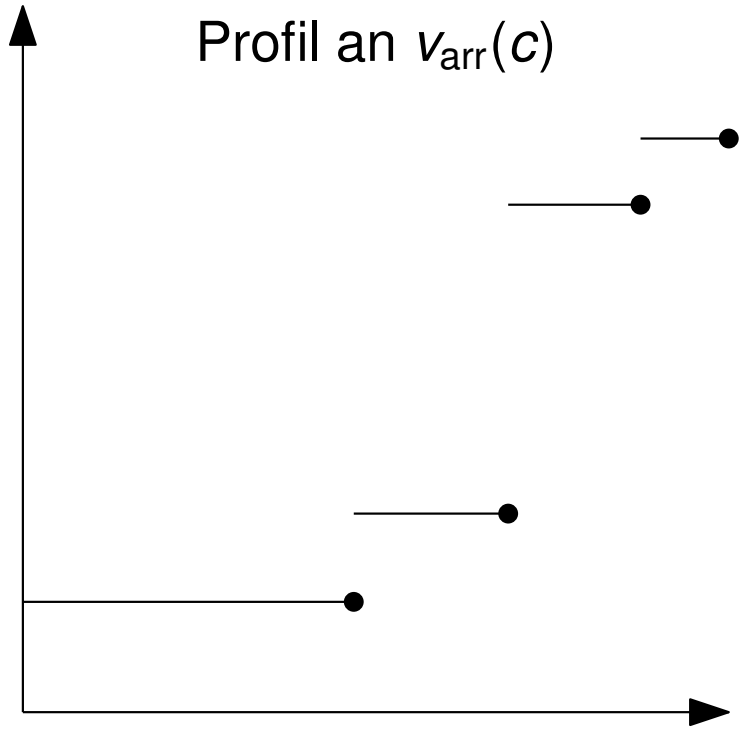
- wird

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

Profile Connection Scan

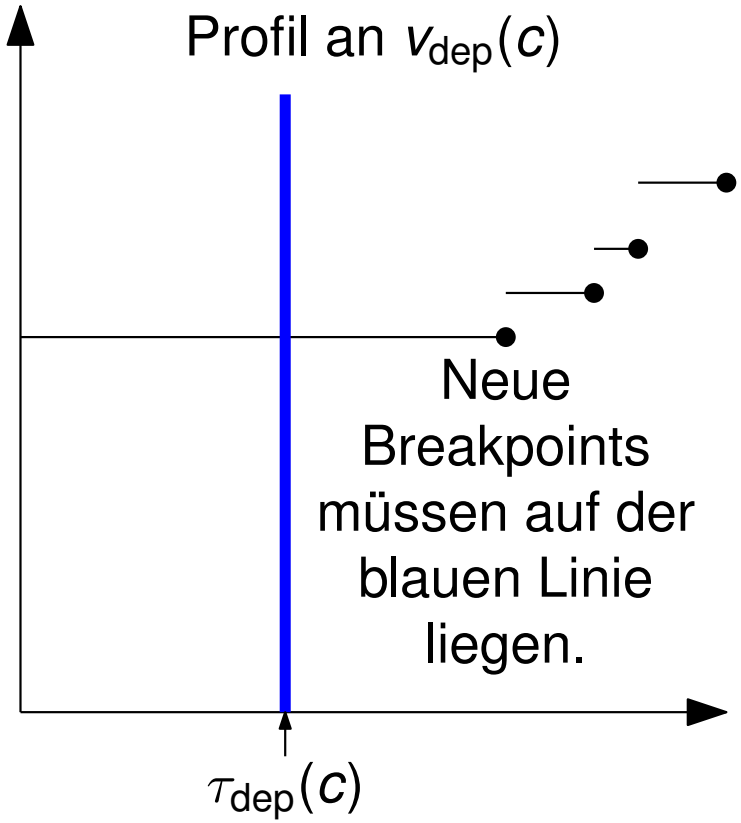
Für jede Connection c absteigend nach Abfahrtszeit:

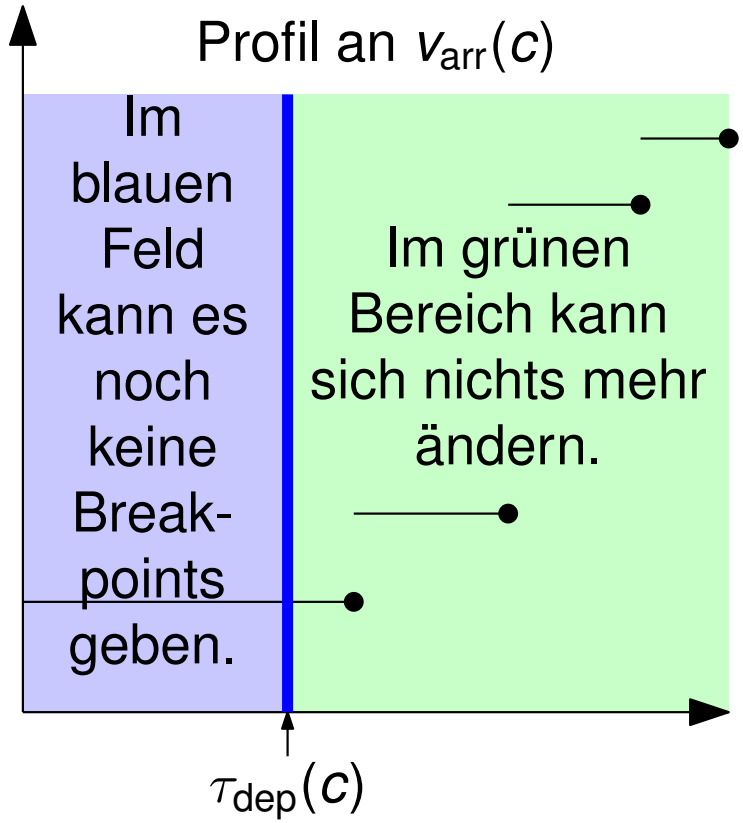




Profile Connection Scan

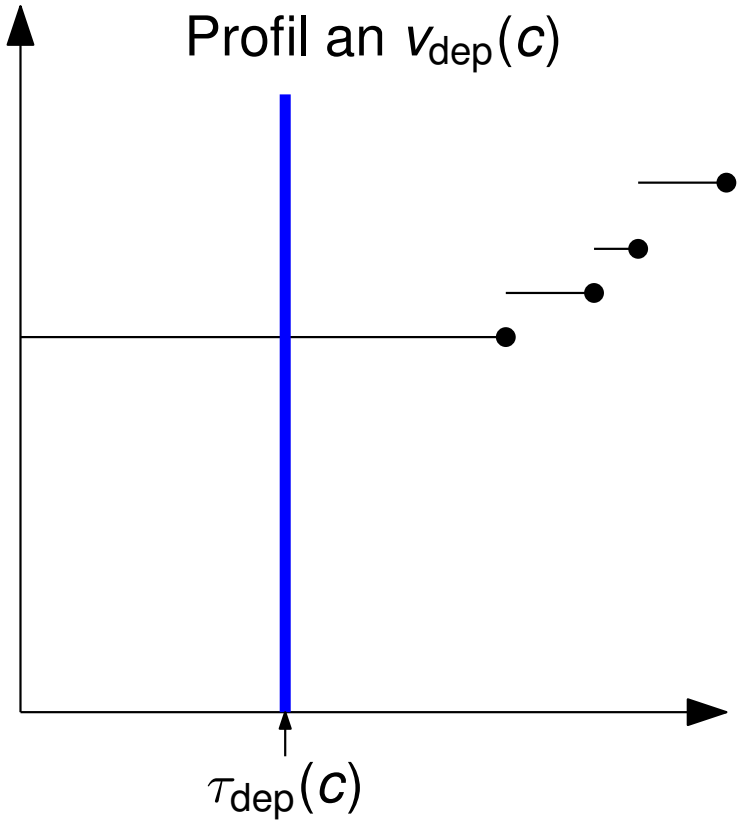
Für jede Connection c absteigend nach Abfahrtszeit:

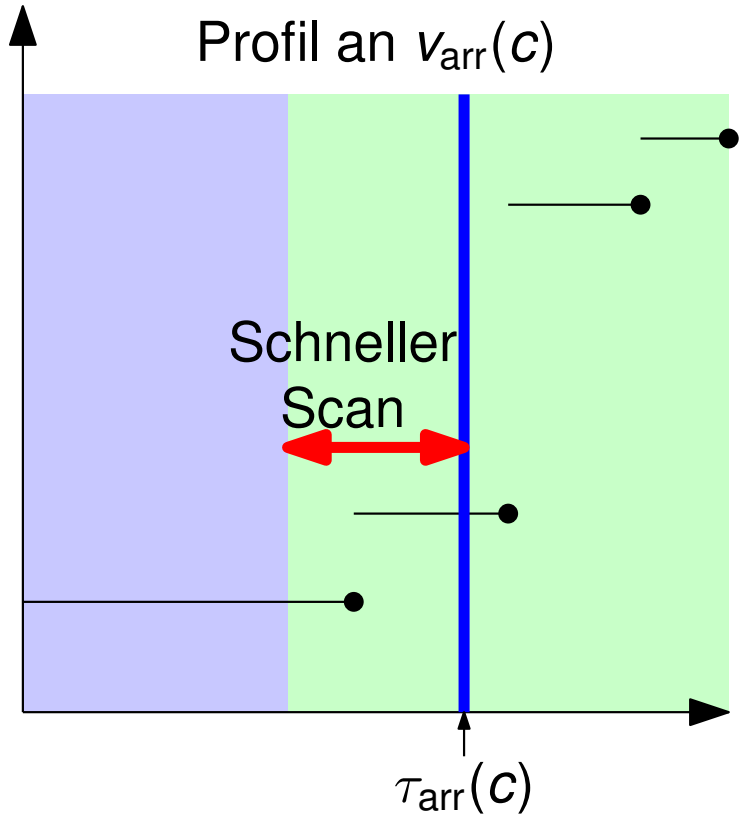




Profile Connection Scan

Für jede Connection c absteigend nach Abfahrtszeit:

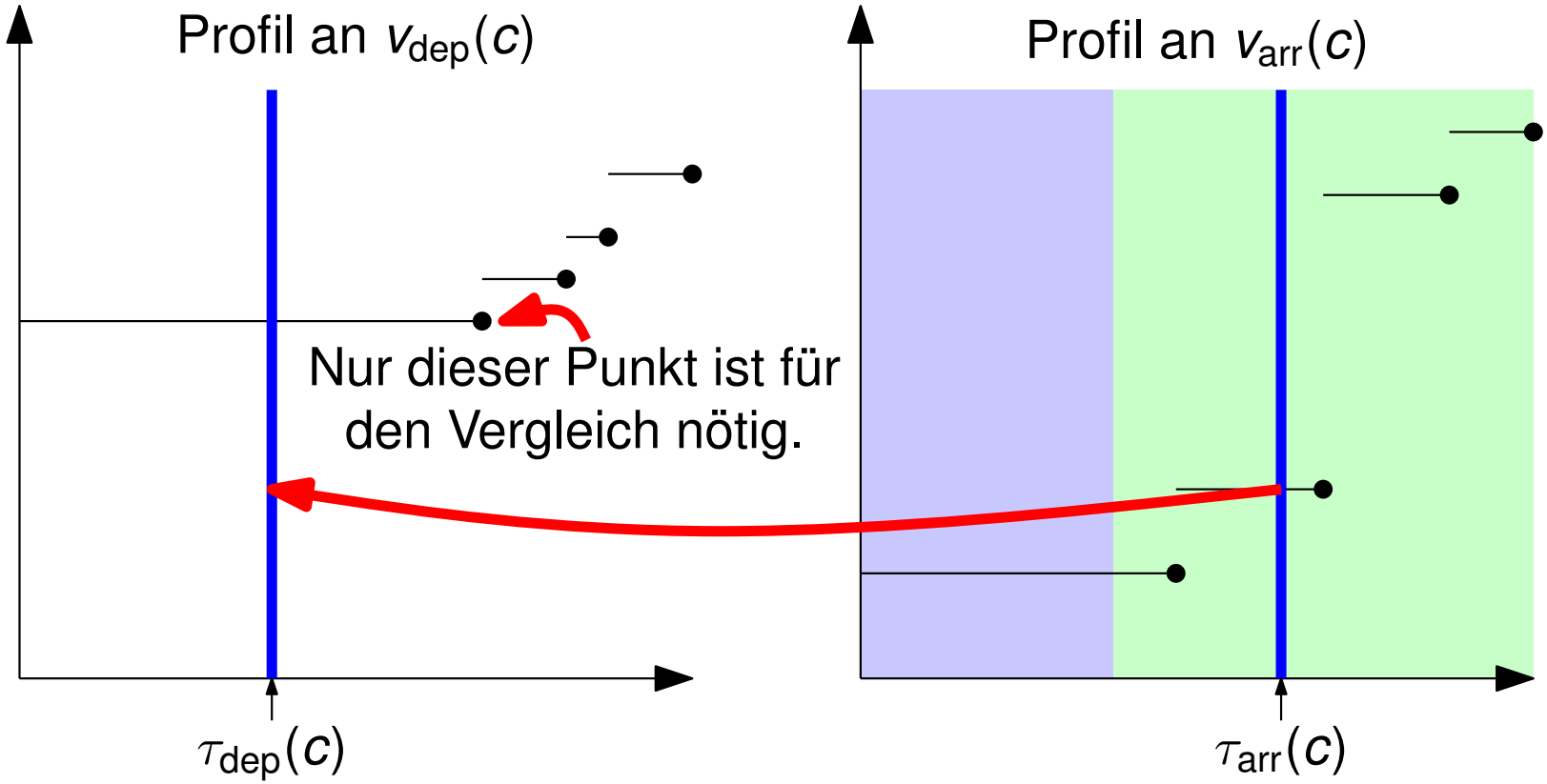




In der Praxis: Sehr kurzer linearer Scan $\rightarrow$ lineare Suche besser

Profile Connection Scan

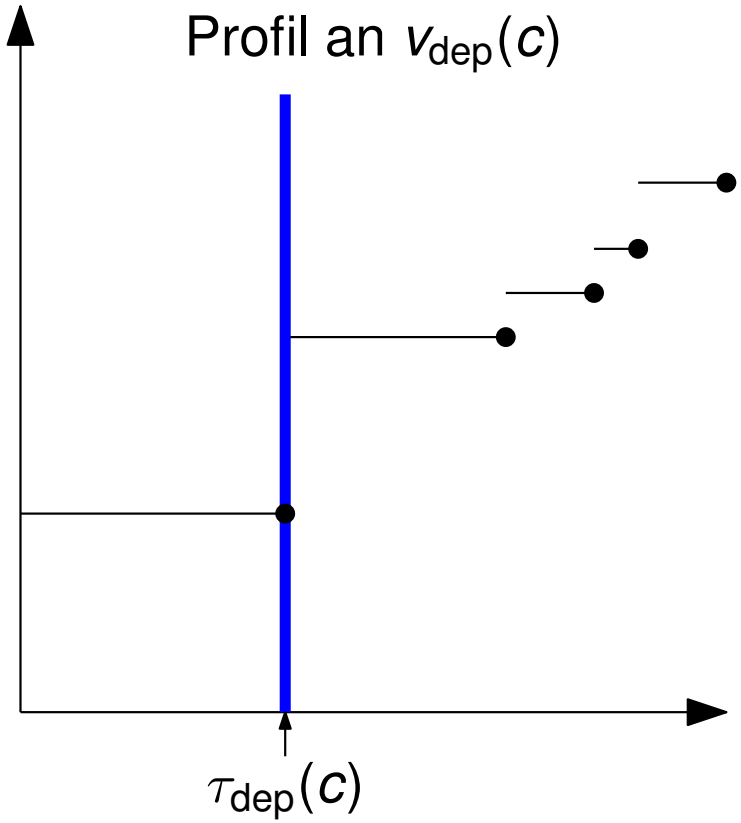
Für jede Connection c absteigend nach Abfahrtszeit:

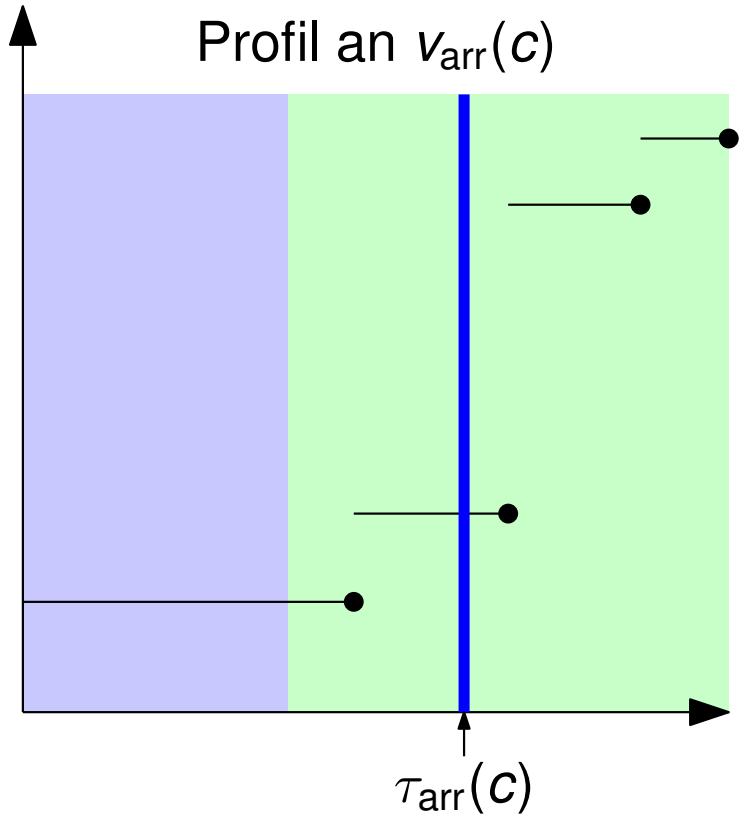


Teste, ob der neue Breakpoint unterhalb dem bereits existierenden ist.

Profile Connection Scan

Für jede Connection c absteigend nach Abfahrtszeit:





Neuen Breakpoint einfügen, wenn er unterhalb ist.

Alternative Profilauswertung

- Auswertung der Profile in der Praxis schnell
- Geht das auch beweisbar in $\mathcal{O}(1)$?

Alternative Profilauswertung

- Auswertung der Profile in der Praxis schnell
- Geht das auch beweisbar in $\mathcal{O}(1)$?

- Ja, mit leichter Modifikation
- (Geht nicht mit Fußwegen)

Alternative Profilauswertung

- **Idee:** Füge einen Breakpoint für jede Connection ein
- Aus

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

- wird

  $d \leftarrow \tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c)))$
  $x \leftarrow (d, \min\{\tau_c, P[v_{\text{dep}}(c)][0].a\})$
  Füge x am Anfang von $P[v_{\text{dep}}(c)]$ ein

Alternative Profilauswertung

Warum?

- Aufeinander folgende Breakpoints mit derselben Ankunftszeit möglich
- Mehr Breakpoints als nötig?
- Was bringt uns das?

Alternative Profilauswertung

Warum?

- Aufeinander folgende Breakpoints mit derselben Ankunftszeit möglich
- Mehr Breakpoints als nötig?
- Was bringt uns das?

Antwort:

- Auswertung von $P[v_{\text{arr}}(c)]$ zum Zeitpunkt $\tau_{\text{arr}}(c)$:
  - Finde erste erreichbare Connection c', die an $v_{\text{arr}}(c)$ abfährt
  - $P[v_{\text{arr}}(c)]$ hat einen Breakpoint für c'
  - Gib Ankunftszeit dieses Breakpoints aus
- Index des Breakpoints für c' ist unabhängig von t

$\rightarrow$ Berechne Index von c' für jede Connection c vor

$\rightarrow$ Profilauswertung in $\mathcal{O}(1)$

$\rightarrow$ Zeit für Profilsuche beweisbar linear in der Anzahl an Connections

τ_{min} und τ_{max} verwenden

- Profilanfrage fragt nur nach Journeys in Zeitspanne $[\tau_{\text{min}}, \tau_{\text{max}}]$
- Wie nutzen wir das aus?

τ_{min} und τ_{max} verwenden

- Profilanfrage fragt nur nach Journeys in Zeitspanne $[\tau_{\text{min}}, \tau_{\text{max}}]$
- Wie nutzen wir das aus?

τ_{min} und τ_{max} verwenden:

- Scanne nur Connections c mit $\tau_{\text{min}} \leq \tau_{\text{dep}}(c) \leq \tau_{\text{max}}$
- Finde späteste Connection c mit $\tau_{\text{dep}}(c) \leq \tau_{\text{max}}$ durch binäre Suche

Fußwege

- **Bisher:** Wir haben gesehen, wie man finale Fußwege behandelt.
- **Nun:** Wie geht man mit initialen und Fußwegen in der Mitte um?

Fußwege

- **Bisher:** Wir haben gesehen, wie man finale Fußwege behandelt.
- **Nun:** Wie geht man mit initialen und Fußwegen in der Mitte um?

Idee 1:

- Expandiere Fußwege zu Connections
- Für jede Connection, die per Fußweg erreichbar ist:
  Füge zeitabhängige Fußweg-Connection ein
- Können sehr viele Connections werden

Fußwege

- **Bisher:** Wir haben gesehen, wie man finale Fußwege behandelt.
- **Nun:** Wie geht man mit initialen und Fußwegen in der Mitte um?

Idee 1:

- Expandiere Fußwege zu Connections
- Für jede Connection, die per Fußweg erreichbar ist:
  Füge zeitabhängige Fußweg-Connection ein
- Können sehr viele Connections werden

Idee 2:

- Laufe Fußwege beim Einfügen von Breakpoints ab

Fußwege

- Aus

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

- wird

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

Fußwege

Problem:

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

Fußwege

Problem:

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

Idee:

- (d, a) soll eingefügt werden
- Verschiebe alle Breakpoints (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 $P[v]$ ein

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Fußwege

wird

```
Tmp ← {}
while P[v][0].d < d do
    Füge P[v][0] in Tmp ein
    Lösche P[v][0] aus P[v]
if a < P[v][0].a then
    if P[v][0].d = d then
        P[v][0].a ← a
    else
        Füge (d, a) am Anfang von P[v] ein
for alle (d', a') in Tmp absteigend in d' do
    if a' < P[v][0].a then
        Füge (d', a') am Anfang von P[v] ein
```

Fußwege

Optimierung:

- Wenn $P[v_{\text{dep}}(c)]$ durch das Einfügen des neuen Breakpoints nicht verändert wurde, muss man die Fußwege nicht ablaufen
- Gültig wegen transitiv abgeschlossener Fußwege mit Dreiecksungleichung
- Analog zu Fußwege-Optimierung bei Earliest Arrival Connection Scan

Hinweis

Diese Optimierung macht meistens einen signifikanten Unterschied

Anzahl Umstiege optimieren

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

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 $a[i]$ mit konstanter Länge
  - In den Beispielen Länge 8
  - Geht mit beliebigen Längen
- $a[i]$ ist Ankunftszeit an t, 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 Profil $P[v]$ pro Stop v
Initialisiere optimale Ankunftszeiten $d_{\text{trip}}[T]$ pro Trip T

for *alle 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 man sitzenbleibt (benutzt $d_{\text{trip}}[\text{trip}(c)]$)

$\tau_3 \leftarrow$ Ankunftszeit, wenn man umsteigt (benutzt $P[v_{\text{arr}}(c)]$)

`// τc: Optimale Ankunftszeit, falls c benutzt wird`

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

`// 2. Aktualisiere Stop- und Trip-Daten`

Füge τ_c zu $P[v_{\text{dep}}(c)]$ hinzu

Aktualisiere $d_{\text{trip}}[\text{trip}(c)]$ mit τ_c

- 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

  $\tau_1 \leftarrow \text{broadcast}(\tau_{\text{arr}}(c) + d_t[v_{\text{arr}}(c)])$

- Berechnung von $d_t[\cdot]$ wie bisher

Sitzenbleiben

Trip-Datenstruktur

- $d_{\text{trip}}[T]$ ist Vektor statt skalarer Ankunftszeit
- Aus

  Initialisiere optimale Ankunftszeiten $d_{\text{trip}}[T]$ pro Trip T

- wird

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

Sitzenbleiben

Trip-Datenstruktur

- $d_{\text{trip}}[T]$ ist Vektor statt skalarer Ankunftszeit
- Aus

  $\tau_2 \leftarrow$ Ankunftszeit, wenn man sitzenbleibt
- wird

  $\tau_2 \leftarrow d_{\text{trip}}[\text{trip}(c)]$
- Diesmal sind die Variablen aber Vektoren

Sitzenbleiben

Trip-Datenstruktur

- $d_{\text{trip}}[T]$ ist Vektor statt skalarer Ankunftszeit
- Aus

  Füge τ_c in Daten von Trip $\text{trip}(c)$ ein
- wird

  $d_{\text{trip}}[\text{trip}(c)] \leftarrow \tau_c$
- Diesmal sind die Variablen aber Vektoren

Umsteigen

Stop-Datenstruktur

- Profil bildet Abfahrtszeit auf Vektor von Ankunftszeiten ab
- Aus

  Initialisiere Profil $P[v]$ pro Stop v

- wird

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

Profile – Datenstruktur

- **Bisher:** $P[v]$ ist Array von Breakpoints (d, a)
- **Nun:** $P[v]$ ist Array von Breakpoints $(d, a = (a_1, \ldots, a_8))$
- Array ist nach wie vor sortiert nach Abfahrtszeit

Profile – Datenstruktur

- **Bisher:** $P[v]$ ist Array von Breakpoints (d, a)
- **Nun:** $P[v]$ ist Array von Breakpoints $(d, a = (a_1, \ldots, a_8))$
- Array ist nach wie vor sortiert nach Abfahrtszeit

Interpretation:

- $P[v]$ ist Profil von Stop v nach t
- $P[v]$ am Zeitpunkt τ_{dep} auswerten ergibt Vektor $(\tau^1_{\text{arr}}, \ldots, \tau^8_{\text{arr}})$
- In τ^i_{arr} steht, wann ich an t ankomme, wenn ich
  - um τ_{dep}
  - an v losfahre
  - und höchstens i mal aussteigen darf

Profile – Initialisierung

- Jedes Array endet mit Breakpoint $(\infty, \text{broadcast}(\infty))$
- Aus

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

- wird

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

Profile – Auswertung

- Pseudocode für Profilauswertung bleibt unverändert
- **Aber:** a ist nun ein Vektor
- Shift-Operation, um Anzahl Umstiege zu erhöhen
- Aus

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

- wird

for $i \leftarrow 0, \ldots, |P[v_{\text{arr}}(c)]| - 1$ **do**
  $(d, a) \leftarrow P[v_{\text{arr}}(c)][i]$
  if $\tau_{\text{arr}}(c) \leq d$ **then**
    $\tau_3 \leftarrow \text{shift}(a)$
    break

Profile – Einfügen

- Füge Breakpoint ein, wenn mindestens eine Komponente besser ist
- Neuer Breakpoint ist komponentenweises Minimum
- Aus

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

- wird

$d \leftarrow \tau_{\text{dep}}(c) - \tau_{\text{ch}}(v_{\text{dep}}(c))$
$a \leftarrow \min\{\tau_c, P[v_{\text{dep}}(c)][0].a\}$
if $a \neq P[v_{\text{dep}}(c)][0].a$ **then**
  if $d = P[v_{\text{dep}}(c)][0].d$ **then**
    $P[v_{\text{dep}}(c)][0].a \leftarrow a$
  else
    Füge (d, a) am Anfang von $P[v_{\text{dep}}(c)]$ ein

Umstiege Pareto-optimieren

- Vektor a der Länge n hat die Komponenten:

$$(a_1, a_2 \ldots a_n)$$

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

Umstiege Pareto-optimieren

- Vektor a der Länge n hat die Komponenten:

$$(a_1, a_2 \ldots a_n)$$

- Journeys werden gefunden bis maximal n Mal aussteigen
- $\rightarrow$ 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 a_n die früheste Ankunftszeit ohne beschränkte Umstiege ist.
- Die Bedeutung von a_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: $(\infty, x_1, x_2, x_3, x_5, x_6, \min\{x_7, x_8\})$

Weitere Optimierungen

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

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 $\tau_0, \tau_1, \ldots, \tau_n$, an denen ein Zug abfährt oder ankommt

- Indizes respektieren die zeitliche Ordnung, d.h., $\tau_i < \tau_j \iff i < j$
- 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 $\tau_0, \tau_1, \ldots, \tau_n$, an denen ein Zug abfährt oder ankommt

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

Fußwege

Nicht-triviale Interaktion mit Fußwegen (nicht in der Vorlesung)

Journey-Entpackung

Option 1:

- Speichere mit jeder Ankunftszeit im Profil das erste Leg der entsprechenden Journey
- Entpacke Journeys rekursiv

Journey-Entpackung

Option 1:

- Speichere mit jeder Ankunftszeit im Profil das erste Leg der entsprechenden Journey
- Entpacke Journeys rekursiv

Option 2:

- Traversiere Fahrplan DFS-mäßig von s 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

Earliest Arrival One-to-One:

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

Earliest Arrival 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 Ideen, die nicht in der Vorlesung vorkommen.)

Laufzeiten

Non-Pareto Profile All-to-One:

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

Pareto Profile All-to-One (mit höchstens 8 Trips pro Journey):

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

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.